SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning

Nishat Koti*, Mahak Pancholi*, Arpita Patra*, Ajith Suresh*

*Department of Computer Science and Automation, Indian Institute of Science, Bangalore India

{kotis, mahakp, arpita, ajith}@iisc.ac.in

***Abstract*—Performing ML computation on private data while maintaining data privacy aka Privacy-preserving Machine Learning (PPML) is an emergent field of research. Recently, PPML has seen a visible shift towards the adoption of Secure Outsourced Computation (SOC) paradigm, due to the heavy computation that it entails. In the SOC paradigm, computation is outsourced to a set of powerful and specially equipped servers that provide service on a pay-per-use basis. In this work, we propose SWIFT, a *robust* PPML framework for a range of ML algorithms in SOC setting, that guarantees output delivery to the users irrespective of any adversarial behaviour. Robustness, a highly desirable feature, evokes user participation without the fear of denial of service.**

At the heart of our framework lies a highly-efficient, maliciously-secure, three-party computation (3PC) over rings that provides guaranteed output delivery (GOD) in the honest-majority setting. To the best of our knowledge, SWIFT is the first robust and efficient PPML framework in the 3PC setting. SWIFT is as fast as the best-known 3PC framework BLAZE (Patra et al. NDSS'20) which only achieves fairness[1]. We extend our 3PC framework for four parties (4PC). In this regime, SWIFT is as fast as the best known *fair* 4PC framework Trident (Chaudhari et al. NDSS'20) and twice faster than the best-known *robust* 4PC framework FLASH (Byali et al. PETS'20).

We demonstrate the practical relevance of our framework by benchmarking two important applications– i) ML algorithms: Logistic Regression and Neural Network, and ii) Biometric matching, both over a 64-bit ring in WAN setting. Our readings reflect our claims as above.

I. Introduction

Privacy Preserving Machine Learning (PPML), a booming field of research, allows Machine Learning (ML) computations over private data of users while ensuring the privacy of the data. PPML finds applications in sectors that deal with sensitive/confidential data, e.g. healthcare, finance, and in cases where organisations are prohibited from sharing client information due to privacy laws such as CCPA and GDPR. However, PPML solutions make the already computationally heavy ML algorithms more compute-intensive. An average end-user who lacks the infrastructure required to run these tasks prefers to outsource the computation to a powerful set of specialized cloud servers and leverage their services on a pay-per-use basis. The Secure Outsourced Computation (SOC) paradigm is thus an apt fit for the need of the moment. The goal is to achieve *malicious* security against the collusion of an arbitrary number of users with some of the servers. Many recent works [1]–[9] exploit Secure Multiparty Computation (MPC) techniques to realize PPML in the SOC setting where the servers enact the role of the parties. Informally, MPC enables n mutually distrusting parties to compute a function over their private inputs, while ensuring the privacy of the same against an adversary controlling up to t parties. Both the training and prediction phases of PPML can be realized in the SOC setting. The common approach of outsourcing followed in the PPML literature, as well as by our work, requires the users to secret-share[2] their inputs between the set of hired (untrusted) servers, who jointly interact and compute the secret-shared output, and reconstruct it towards the users.

In a bid to improve practical efficiency, many recent works [6]–[17] cast their protocols into an *input-independent* preprocessing phase, and *input-dependent* online phase. Using this paradigm, the input-independent (yet function-dependent) computationally heavy tasks can be computed in the preprocessing phase in advance, resulting in a fast online phase. This paradigm suits scenario analogous to PPML setting, where functions (ML algorithms) typically need to be evaluated a large number of times, and the function description is known beforehand. To further enhance practical efficiency by leveraging CPU optimizations, recent works [15], [18]–[21] propose MPC protocols that work over 32 or 64 bit rings. Lastly, solutions for a small number of parties have received a huge momentum due to the many cost-effective customizations that they permit, for instance, a cheaper realisation of multiplication through custom-made secret sharing schemes [6]–[9], [22], [23] (that do not scale for a large number of parties).

We now motivate the need for robustness aka guaranteed output delivery (GOD) over fairness, or even abort security, in the domain of PPML. Robustness provides the guarantee of output delivery to all protocol participants, no matter how the adversary misbehaves. Robustness is extremely crucial for real-world deployment and usage of PPML techniques. Consider the following scenario wherein an ML model owner wishes to provide inference service. The model owner shares the model parameters between the servers, while the end-users share their queries. A protocol that provides security with abort or fairness will not suffice as in both the cases a malicious adversary can act in a way so that the protocol results in an abort which means that the user will not get the desired output. This leads to denial of service and heavy economic losses for the service provider. For data providers who want to collaboratively build a model on their data, more training data leads to a better, more accurate model, which enables them to provide better ML services and, consequently, attract more clients. A robust framework encourages active involvement from multiple data

[1]Fairness ensures either all or none receive the output, whereas GOD ensures guaranteed output delivery no matter what.

[2]The threshold of the secret-sharing is decided based on the number of corrupt servers so that privacy is preserved.

providers. Hence, for the seamless adoption of PPML solutions in the real-world, the robustness of the protocol is of utmost importance. The hall-mark result of [24] suggests that an honest-majority amongst the servers is necessary to achieve robustness (in fact, to achieve fairness which is a weaker goal than robustness).

Consequent to the discussion above, we focus on the honest-majority setting with a small set of parties, especially 3 and 4 parties with one corruption, both of which have drawn enormous attention recently [6]–[9], [22], [23], [25]–[30]. Our protocols work over rings, cast in input-independent and dependent phases, and achieves GOD.

Before we move on to discuss the related work, we state the challenges for stretching MPC techniques for PPML.

a) Challenges in PPML: MPC protocols, while a potential solution for SOC, cannot be directly plugged in to achieve the goal of PPML. This is because, ML algorithms involve decimal values that need to be embedded into rings, resulting in doubling of the fractional part after every multiplication. This was handled in previous works via secure truncation [1], [4], [6], [8], [9]. Secondly, ML algorithms require techniques to efficiently alternate between boolean and arithmetic computations, as some operations like secure comparison are better realized in boolean representation, while operations like dot product are better realized in arithmetic representation. These challenges call for customized MPC protocols for truncation, comparison, dot product, and many others.

b) Related Work: We restrict the relevant works with a small number of parties and honest-majority, focusing first on MPC, followed by PPML. MPC protocols for a small population can be cast into orthogonal domains of low latency protocols [27], [31], [32], and high throughput protocols [6], [9], [18], [22], [23], [26], [28], [30], [33]–[35]. In the 3PC setting, [6], [22] provide efficient semi-honest protocols wherein ASTRA [6] improved upon [22] by casting the protocols in the preprocessing model and provided a fast online phase. ASTRA further provided security with fairness in the malicious setting with an improved online phase compared to [23]. Later, a maliciously-secure 3PC protocol based on distributed zero-knowledge techniques was proposed by Boneh et al. [29] providing abort security. Further, building on [29] and enhancing the security to GOD, Boyle et al. [30] proposed a concretely efficient 3PC protocol with an amortized communication cost of 3 field elements (can be extended to work over rings) per multiplication gate. Concurrently, BLAZE [9] provided a fair protocol in the preprocessing model, which required communicating 3 ring elements in each phase. However, BLAZE eliminated the reliance on the computationally intensive distributed zero-knowledge system (whose efficiency kicks in for large circuit or many multiplication gates) from the online phase and pushed it to the preprocessing phase. This resulted in a faster online phase compared to [30].

In the regime of 4PC, Gordon et al. [36] presented protocols achieving abort security and GOD. However, [36] relied on expensive public-key primitives and broadcast channels to achieve GOD. Trident [8] improved over the abort protocol of [36], providing a fast online phase achieving security with fairness, and presented a framework for mixed world computations [20]. A robust 4PC protocol was provided in FLASH [7] which requires communicating 6 ring elements, each, in the preprocessing and online phases.

In the PPML domain, MPC has been used for various ML algorithms such as Decision Trees [37], Linear Regression [38], [39], k-means clustering [40], [41], SVM Classification [42], [43], Logistic Regression [44]. In the 3PC SOC setting, the works of ABY3 [4] and SecureNN [5], provide security with abort. This was followed by ASTRA [6], which improves upon ABY3 and achieves security with fairness. ASTRA presents primitives to build protocols for Linear Regression and Logistic Regression inference. Recently, BLAZE improves over the efficiency of ASTRA and additionally tackles training for the above ML tasks, which requires building additional PPML building blocks, such as truncation and bit to arithmetic conversions. In the 4PC setting, the first robust framework for PPML was provided by FLASH [7] which proposed efficient building blocks for ML such as dot product, truncation, MSB extraction, and bit conversion. The works of [1], [4]–[9] work over rings to garner practical efficiency. In terms of efficiency, BLAZE and respectively FLASH and Trident are the closest competitors of this work in 3 and 4 party settings. We now present our contributions and compare them with these works.

A. Our Contributions

We propose a robust maliciously-secure framework for PPML in the SOC setting, **SWIFT**, with a set of 3 and 4 servers having an honest-majority. At the heart of our framework lies highly-efficient, maliciously-secure, 3PC and 4PC over rings (both $\mathbb{Z}_{2^\ell}$ and $\mathbb{Z}_{2^1}$) that provide GOD in the honest-majority setting. We cast our protocols in the preprocessing model which helps to push the computationally intensive tasks into the preprocessing phase, resulting in a fast online phase. Apart from PPML, our framework also supports biometric matching.

To the best of our knowledge, SWIFT is the first robust and efficient PPML framework in the 3PC setting and is as fast as the best known *fair* 3PC framework BLAZE [9]. We extend our 3PC framework for 4 parties. In this regime, SWIFT is as fast as the best known *fair* 4PC framework Trident [8] and twice faster than best known *robust* 4PC framework FLASH [7]. We next detail our framework, followed by an overview of technical novelties:

a) Robust 3/4PC frameworks: The framework consists of a range of primitives realized in a privacy-preserving way which is ensured via running computation in a secret-shared fashion. Secret-sharing tolerating up to one malicious corruption is the basis for all our constructions. We use the sharing over both $\mathbb{Z}_{2^\ell}$ and its special instantiation $\mathbb{Z}_{2^1}$ and refer them as *arithmetic* and respectively *boolean* sharing. Our framework consists of realizations for all primitives needed for general MPC and PPML such as multiplication, dot-product, truncation, bit extraction (given arithmetic sharing of a value v, this is used to generate boolean sharing of the most significant bit (msb) of the value), bit to arithmetic sharing conversion (converts the boolean sharing of a single bit value to its arithmetic sharing), bit injection (computes the arithmetic sharing of $\mathsf{b} \cdot \mathsf{v}$, given the boolean sharing of a bit b and the arithmetic sharing of a ring element v) and above all input sharing and output reconstruction. The performance comparison in

Building Blocks	3PC					4PC				
	Ref.	Pre.	Online		Security	Ref.	Pre.	Online		Security
		Comm. (ℓ)	Rounds	Comm. (ℓ)			Comm. (ℓ)	Rounds	Comm. (ℓ)	
Multiplication	[29]	1	1	2	Abort					
	[30]	-	3	3	GOD	Trident	3	1	3	Fair
	BLAZE	3	1	3	Fair	FLASH	6	1	6	GOD
	SWIFT	**3**	**1**	**3**	GOD	**SWIFT**	**3**	**1**	**3**	GOD
Dot Product						Trident	3	1	3	Fair
	BLAZE	$3\mathsf{n}$	1	3	Fair	FLASH	6	1	6	GOD
	SWIFT	$\mathbf{3n}$	**1**	**3**	GOD	**SWIFT**	**3**	**1**	**3**	GOD
Dot Product with Truncation						Trident	6	1	3	Fair
	BLAZE	$3\mathsf{n}+2$	1	3	Fair	FLASH	8	1	6	GOD
	SWIFT	$\mathbf{3n+2}$	**1**	**3**	GOD	**SWIFT**	**4**	**1**	**3**	GOD
Bit Extraction						Trident	≈ 8	$\log \ell + 1$	≈ 7	Fair
	BLAZE	9	$1 + \log \ell$	9	Fair	FLASH	14	$\log \ell$	14	GOD
	SWIFT	**9**	$\mathbf{1 + \log \ell}$	**9**	GOD	**SWIFT**	$\approx \mathbf{7}$	$\log \ell$	$\approx \mathbf{7}$	GOD
Bit to Arithmetic						Trident	≈ 3	1	3	Fair
	BLAZE	9	1	4	Fair	FLASH	6	1	8	GOD
	SWIFT	**9**	**1**	**4**	GOD	**SWIFT**	$\approx \mathbf{3}$	**1**	**3**	GOD
Bit Injection						Trident	≈ 6	1	3	Fair
	BLAZE	12	2	7	Fair	FLASH	8	2	10	GOD
	SWIFT	**12**	**2**	**7**	GOD	**SWIFT**	$\approx \mathbf{6}$	**1**	**3**	GOD

– Notations: ℓ - size of ring in bits, n - size of vectors for dot product, κ - computational security parameter.

TABLE I: 3PC and 4PC: Comparison of SWIFT with its closest competitors in terms of Communication and Round Complexity

terms of concrete cost for communication and round, for the preprocessing and online phase of PPML primitives, for both 3PC and 4PC, appear in Table I. Akin to our earlier claim, SWIFT is neck to neck with BLAZE for each primitive (while improving security from fairness to robustness). The same is true for SWIFT and Trident in the 4-party case. On the other hand, SWIFT is doubly faster than FLASH.

We now point out a few lucrative features that our framework offers, some of which are shared with the earlier works on PPML. First, by resorting to the preprocessing model, we achieve dot product with online cost completely independent of vector size. This brings a massive gain since dot product is one of the most invoked primitives in PPML. The other primitives also show improved performance in the online phase in the preprocessing paradigm. The multiplication protocol gives a technical basis for our dot product protocol. Instead of using the multiplication of [30] (which has the same overall communication cost as that of our online phase), we build a new protocol that gives rise to a dot product with the above feature. Also, the multiplication protocol of [30] involves distributed zero-knowledge protocols. The cost of this heavy machinery gets amortized over, only for large circuits having millions of gates, which is very unlikely for inferences, and moderately heavy training in the PPML domain. Second, extending to the 4-party setting brings to the table a flurry of performance improvements over 3PC. Most prominent of all is a dot product with cost totally independent of vector size, which remains as a challenging open question in 3PC setting. Third, the only two tasks, input sharing and output reconstruction, carried out by the users of our framework are very light-weight, in spite of offering GOD. As a final remark, we note that the roles of the servers in our framework are asymmetric and consequently, we only require active participation from two of the servers, while the remaining server(s) is(are) brought in just for the verification towards the end of each phase. In a cloud setting, this may provide additional benefit in terms of monetary cost [45].

We demonstrate the practicality of our protocols by benchmarking Biometric Matching and PPML. For the latter, Logistic Regression (training and inference) and Neural Networks (inference) are considered. The NN training requires mixed-world conversions [4], [8], [20], which we leave as future work. Our PPML blocks can be used to perform training and inference of Linear Regression, Support Vector Machines, and Binarized Neural Networks (as demonstrated in [6]–[9]).

b) Overview of Techniques: Robustness is known to be desirable, yet a costly goal. We work against the pre-assumption on cost, at least for our concerned setting. Starting from BLAZE [9], we overcome a series of hurdles to achieve GOD.

Relay primitive with rate-1 communication. We introduce a new primitive called Joint Message Passing (jmp) that allows two servers to relay a common message to the third server such that either the relay is successful or an honest server (or a conflicting pair) is identified. The striking feature of jmp is that it offers a rate-1 communication i.e. for a message of ℓ elements, it only incurs a communication of ℓ elements (in an amortized sense). Without any extra cost, it allows us to replace several pivotal private communications, that may lead to abort, either because the malicious sender does not send anything or sends a wrong message. Being two-sender primitive, it has one guaranteed honest sender who (in some sense) guards the correct send. Using this primitive, we create a win-win scenario as below. Either the send is successful (and consequently the computation) or an honest server is identified. All our primitives invoke jmp and as a result, the final protocol, either for a general 3PC or a PPML task, requires invocations of many jmp primitives. To leverage amortization, the primitive is designed to have two phases, *send* and *verify*, where the *send* phase is carried out on the flow of the protocol and the *verify* phase takes place once and for all in the end. If the verification goes through (meaning all sends via jmp are successful) the computation carried out already achieves GOD. In the latter case, our computation completes by labelling the honest server as a trusted third party and allowing it to simply carry out the computation centrally. To shoot for GOD, our next step is to conform to all the protocols to be able to use jmp primitive.

Conforming protocols. Among all our protocols, the major challenge came from the multiplication protocol of BLAZE (which is our starting point). Our approach is to manipulate and transform some of the protocol steps so that the information required by a server in a round can be locally computed by two other servers. But this transformation is not straight forward since BLAZE was constructed with a focus towards providing only fairness. We demonstrate with an example below. We consider the online phase of multiplication in BLAZE at a very high level. Let P_0, P_1, P_2 be the three servers. P_1, P_2 need to locally compute additive shares of value v, denoted by $\mathsf{v}_1, \mathsf{v}_2$ and mutually exchange the shares to obtain v in clear. To prevent P_1, P_2 from cheating, P_0 is equipped with preprocessing data that allows it to compute $\mathsf{v}^\star = \mathsf{v} + \delta$ locally. Here δ is a common value possessed by both P_1, P_2 and is unknown to P_0. P_0 then sends $\mathsf{v}^\star$ in clear to both P_1, P_2 who abort in the case of any inconsistency. It is only P_0 who has enough information to compute $\mathsf{v}^\star$, and such an arrangement makes it difficult to achieve GOD in BLAZE. Hence, we modify the online phase in such a way that the pairs (P_0, P_1) and (P_0, P_2) can locally compute one of the two additive shares of $\mathsf{v}^\star$. Servers then communicate the missing share using jmp primitive. Upon obtaining $\mathsf{v}^\star$, P_1, P_2 locally compute v using δ which is known to them. This transformation in the online phase of the multiplication protocol demands a fresh preprocessing phase from scratch. Thus, it is the combination of jmp primitive, a new preprocessing phase, along with the restructuring of the online phase that helps us to obtain higher security guarantee of GOD without affecting the communication cost.

Improved jmp *primitive for 4PC.* In a 4-party setting, we provide an improved instantiation of the jmp primitive, which forgoes the broadcast channel, while retaining the rate-1 property. Whereas, our 3-party instantiation uses a broadcast. We note that, in the 4PC case, our jmp primitive achieves a goal similar to the "bi-convey primitive" of FLASH [7]. However, jmp is more efficient. Further, it allows identifying an honest server, as opposed to two honest servers locally identifying each other in bi-convey, helping to craft a clean and swift completion after this event. We defer other details to §IV. Using jmp, we present better/simpler protocols than 3PC counterparts, specifically the multiplication and dot product.

Robust and Improved Input Sharing and Output Reconstruction. We provide robust protocols for the input sharing and output reconstruction phase in the SOC setting, wherein a user shares its input with the servers and the output is reconstructed towards a user. The need for robustness and light-weightless together makes these slightly non-trivial. As a highlight, we introduce a super-fast online phase for the reconstruction protocol, which gives $4\times$ improvement in terms of rounds (apart from improvement in communication as well) compared to BLAZE. We make sure that a user neither takes part in a jmp protocol nor in a broadcast, both of which need several rounds of communication and are relatively expensive than *atomic* point-to-point communication.

B. Organisation of the paper

The rest of the paper is organized as follows. In §II we describe the system model, preliminaries and notations used. §III and §IV detail our constructs in the 3PC and respectively 4PC setting. These are followed by the applications and benchmarking in §V. The appendix §A elaborates on the preliminaries. Protocols to complete the PPML framework and detailed cost analysis for all the 3PC and 4PC protocols are provided in appendix §B and §C respectively. The security proofs for our constructions follow in appendix §D.

II. Preliminaries

We consider a set of three servers $\mathcal{P} = \{P_0, P_1, P_2\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, malicious adversary that can corrupt at most one server. We use a broadcast channel for 3PC alone, which is inevitable [46]. For ML training, several data-owners who wish to jointly train a model, secret share (using the sharing semantics which appear in the latter part of the paper) their data among the servers. For ML inference, a model-owner and client secret share the model and the query, respectively, among the servers. Once the inputs are available in the shared format, the servers perform computations and obtain the output in the shared form. In the case of training, the output model is reconstructed towards the data-owners, whereas for inference, the prediction result is reconstructed towards the client. We assume that an arbitrary number of data-owners may collude with a corrupt server for training, whereas for the case of prediction, we assume that either the model-owner or the client can collude with a corrupt server. We prove the security of our protocols using standard real-world / ideal-world paradigm. We also explore the above model for the four server setting with $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$. The aforementioned setting has been explored extensively [1], [4], [6]–[9].

Our constructions achieve the strongest security guarantee of GOD. A protocol is said to be *robust* or achieve GOD if all parties obtain the output of the protocol regardless of how the adversary behaves. In our model, this translates to all the data owners obtaining the trained model for the case of ML training, while the client obtains the query output for ML inference. All our protocols are cast into: *input-independent* preprocessing phase and *input-dependent* online phase.

For 3/4PC, the function to be computed is expressed as a circuit ckt, whose topology is public, and is evaluated over an arithmetic ring $\mathbb{Z}_{2^\ell}$ or boolean ring $\mathbb{Z}_{2^1}$. For PPML, we consider computation over the same algebraic structure. To deal with floating-point values, we use Fixed-Point Arithmetic (FPA) [1], [4], [6]–[9] representation in which 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 $\ell = 64$ and $x = 13$, leaving $\ell - x - 1$ bits for the integral part.

The servers use a one-time key setup, modelled as a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 14), to establish pre-shared random keys for pseudo-random functions (PRF) between them. A similar setup is used in [3], [4], [6], [9], [23], [26], [30] for 3 server case, and in [7], [8] for 4 server setting. The key-setup can be instantiated using any standard MPC protocol in the respective setting. Further, our protocols make use of a *collision-resistant* hash function, denoted by $\mathsf{H}()$, and a commitment scheme, denoted by $\mathsf{Com}()$. The formal details of key setup, hash function, and the commitment scheme are deferred to §A.

Notation II.1. The i^{th} element of a vector $\vec{\mathbf{x}}$ is denoted as x_i. The dot product of two n length vectors, $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, is computed as $\vec{\mathbf{x}} \odot \vec{\mathbf{y}} = \sum_{i=1}^{\mathsf{n}} \mathsf{x}_i \mathsf{y}_i$. For two matrices $\mathbf{X}, \mathbf{Y}$, the operation $\mathbf{X} \circ \mathbf{Y}$ denotes the matrix multiplication. The i^{th} bit of an ℓ-bit value v is denoted by $\mathsf{v}[i]$.

Notation II.2. For a bit $\mathsf{b} \in \{0,1\}$, we use b^{R} to denote the equivalent value of b over the ring $\mathbb{Z}_{2^\ell}$. b^{R} will have its least significant bit set to b, while all other bits will be set to zero.

III. Robust 3PC and PPML

In this section, we first introduce the sharing semantics for three servers. Then, we introduce our new Joint Message Passing (jmp) primitive, which plays a crucial role in obtaining the strongest security guarantee of GOD, followed by our protocols in the three server setting.

a) Secret Sharing Semantics: We use the following secret-sharing semantics.

- $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared among P_1, P_2, if P_s for $s \in \{1,2\}$ holds $[\mathsf{v}]_s \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = [\mathsf{v}]_1 + [\mathsf{v}]_2$.
- $\langle\cdot\rangle$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $\langle\cdot\rangle$-shared among $\mathcal{P}$, if
  - there exists $\mathsf{v}_0, \mathsf{v}_1, \mathsf{v}_2 \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$.
  - P_s holds $(\mathsf{v}_s, \mathsf{v}_{(s+1)\%3})$ for $s \in \{0,1,2\}$.
- $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}$, if
  - there exists $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared among P_1, P_2.
  - there exists $\beta_{\mathsf{v}}, \gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ and P_0 holds $([\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2, \beta_{\mathsf{v}} + \gamma_{\mathsf{v}})$ while P_s for $s \in \{1,2\}$ holds $([\alpha_{\mathsf{v}}]_s, \beta_{\mathsf{v}}, \gamma_{\mathsf{v}})$.

b) Arithmetic and Boolean Sharing: Arithmetic sharing refers to sharing over $\mathbb{Z}_{2^\ell}$ while *boolean* sharing, denoted as $[\![\cdot]\!]^{\mathbf{B}}$, refers to sharing over $\mathbb{Z}_{2^1}$.

c) Linearity of the Secret Sharing Scheme: Given the $[\cdot]$-shares of $\mathsf{v}_1, \mathsf{v}_2$, and public constants c_1, c_2, servers can locally compute the $[\cdot]$-share of $c_1\mathsf{v}_1 + c_2\mathsf{v}_2$ as $c_1[\mathsf{v}_1] + c_2[\mathsf{v}_2]$. It is trivial to see that the linearity property is satisfied by $\langle\cdot\rangle$ and $[\![\cdot]\!]$-sharing as well.

A. Joint Message Passing primitive

The jmp primitive allows two servers to relay a common message to the third server such that either the relay is successful or an honest server (or a conflicting pair) is identified. The striking feature of jmp is that it offers a rate-1 communication i.e. for a message of ℓ elements, it only incurs a communication of ℓ elements (in an amortized sense).

The task of jmp is captured in an ideal functionality (Fig. 1) below and the protocol (Fig. 2) realizing the functionality appears subsequently followed by an overview.

Functionality $\mathcal{F}_{\mathsf{jmp}}$

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

Step 1: $\mathcal{F}_{\mathsf{jmp}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from P_s for $s \in \{i,j\}$, while it receives $(\mathsf{Select}, \mathsf{ttp})$ from $\mathcal{S}$. Here ttp denotes the server that $\mathcal{S}$ wants to choose as the TTP. Let $P^\star \in \mathcal{P}$ denote the server corrupted by $\mathcal{S}$.

Step 2: If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{ttp} = \perp$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \perp, \mathsf{msg}_k = \mathsf{v}_i$ and go to **Step 5**.

Step 3: If $\mathsf{ttp} \in \mathcal{P} \setminus \{P^\star\}$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{ttp}$.

Step 4: Else, TTP is set to be the honest party with smallest index. Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k =$ TTP

Step 5: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{0,1,2\}$.

Fig. 1: 3PC: Ideal functionality for jmp primitive

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

- Each server P_s for $s \in \{i,j,k\}$ initializes bit $\mathsf{b}_s = 0$.
- P_i sends v to P_k, while P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.
- P_k broadcasts "(accuse, P_i)", if P_i is silent and TTP = P_j. Analogously for P_j. If P_k accuses both P_i, P_j, then TTP = P_i. Otherwise, P_k receives some $\tilde{\mathsf{v}}$ and either sets $\mathsf{b}_k = 0$ when the value and the hash are consistent or sets $\mathsf{b}_k = 1$. P_k then sends b_k to P_i, P_j and terminates if $\mathsf{b}_k = 0$.
- If P_i does not receive a bit from P_k, it broadcasts "(accuse, P_k)" and TTP = P_j. Analogously for P_j. If both P_i, P_j accuse P_k, then TTP = P_i. Otherwise, P_s for $s \in \{i,j\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.
- P_i, P_j exchange their bits to each other. If P_i does not receive b_j from P_j, it broadcasts "(accuse, P_j)" and TTP = P_k. Analogously for P_j. Otherwise, P_i resets its bit to $\mathsf{b}_i \vee \mathsf{b}_j$ and likewise P_j resets its bit to $\mathsf{b}_j \vee \mathsf{b}_i$.
- P_s for $s \in \{i,j,k\}$ broadcasts $\mathsf{H}_s = \mathsf{H}(\mathsf{v}^*)$ if $b_s = 1$, where $\mathsf{v}^* = \mathsf{v}$ for $s \in \{i,j\}$ and $\mathsf{v}^* = \tilde{\mathsf{v}}$ otherwise. If P_k does not broadcast, terminate. If either P_i or P_j does not broadcast, then TTP = P_k. Otherwise,
  - If $\mathsf{H}_i \neq \mathsf{H}_j$: TTP = P_k.
  - Else if $\mathsf{H}_i \neq \mathsf{H}_k$: TTP = P_j.
  - Else if $\mathsf{H}_i = \mathsf{H}_j = \mathsf{H}_k$: TTP = P_i.

Fig. 2: 3PC: Joint Message Passing Protocol

Given two servers P_i, P_j possessing a common value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, protocol Π_{jmp} proceeds as follows. First, P_i sends v to P_k while P_j sends a hash of v to P_k. The communication of v is done once and for all from P_i to P_k. In the simplest case, P_k receives consistent (value, hash) pair, and the protocol terminates. In all other cases, a TTP is identified as follows without having to communicate v again. Importantly, this part can be run once and for all instances of Π_{jmp} with P_i, P_j, P_k in the same roles, invoked in the final 3PC protocol. Consequently, the cost relevant to this part vanishes in an amortized sense, making the construction rate-1.

Each P_s for $s \in \{i,j,k\}$ maintains a bit b_s initialized to 0, as an indicator for inconsistency. When P_k receives inconsistent (value, hash) pair, it sets $\mathsf{b}_k = 1$ and sends the bit to both P_i, P_j, who cross-check with each other by exchanging the bit and turn on their inconsistency bit if the bit received from either P_k or its fellow sender is turned on. A party broadcasts a hash of its value when its inconsistency bit is on;[3] P_k's value is the one it receives from P_i. At this stage, there are a bunch of possible cases and a detailed analysis determines an eligible TTP in each case.

When P_k is silent, the protocol is understood to be complete. This is fine irrespective of the status of P_k– an honest P_k never skips this broadcast with inconsistency bit on and a corrupt P_k implies honest senders. If either P_i or P_j is silent, then P_k is picked as TTP which is surely honest. A corrupt P_k could not make one of $\{P_i, P_j\}$ speak as the senders (honest in this case) are on agreement on their inconsistency bit (due to their mutual exchange of inconsistency bit). When all of them speak and (i) the senders' hashes do not match, P_k is picked as TTP; (ii) one of the senders conflicts with P_k, the other sender is picked as TTP; and lastly (iii) if there is no conflict, P_i is picked as TTP. The first two cases are self-explanatory. In the last case, either P_j or P_k is corrupt. Because a corrupt P_i can have honest P_k speak (and hence turn on its inconsistency bit), by sending a v' whose hash is not same as that of v and so inevitably the hashes of honest P_j and P_k will conflict.

As a final touch, we ensure that, in each step, a party raises a public alarm (via broadcast) accusing a party who is silent when it is not supposed to be, and the protocol terminates immediately by labelling the party as TTP who is neither the complainer nor the accused.

Notation III.1. We say that P_i, P_j $\mathsf{jmp\text{-}send}$ v to P_k when they invoke $\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{v})$.

Using jmp *in protocols.* As mentioned in the introduction, the protocol for jmp needs to be viewed consisting of two phases (*send, verify*), where *send* phase consists of P_i sending v to P_k and the rest goes to *verify* phase. Looking ahead, most of our protocols use jmp, and consequently our final construction, either of general MPC or any PPML task will have several calls to jmp. To leverage amortization, the *send* phase will be executed in all protocols invoking jmp on the flow, while the *verify* for a fixed ordered pair of senders will be executed once and for all in the end. The *verify* phase will determine if all the sends were correct. If not, a TTP is identified, as explained, and the computation completes with the help of TTP, just as in the ideal-world.

B. 3PC Protocols

We now describe the protocols for 3 parties/servers.

a) Sharing Protocol: Protocol Π_{sh} (Fig. 3) allows a server P_i to generate $[\![\cdot]\!]$-shares of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. In the preprocessing phase, P_0, P_j for $j \in \{1,2\}$ along with P_i sample a random $[\alpha_{\mathsf{v}}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2, P_i sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$. This allows P_i to know both α_{v} and γ_{v} in clear. During the online phase, if $P_i = P_0$, then P_0 sends $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to P_1. P_0, P_1 then $\mathsf{jmp\text{-}send}$ β_{v} to P_2 to complete the secret sharing. If $P_i = P_1$, P_1 sends $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to P_2. Then P_1, P_2 $\mathsf{jmp\text{-}send}$ $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0. The case for $P_i = P_2$ proceeds similar to that of P_1. The correctness of the shares held by each server is assured by the guarantees of Π_{jmp}.

[3]This hash can be computed on a combined message across many calls of jmp.

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

Preprocessing:

- If $P_i = P_0$: P_0, P_j, for $j \in \{1,2\}$, together sample random $[\alpha_{\mathsf{v}}]_j \in \mathbb{Z}_{2^\ell}$, while $\mathcal{P}$ together sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$.
- If $P_i = P_1$: P_0, P_1 together sample random $[\alpha_{\mathsf{v}}]_1 \in \mathbb{Z}_{2^\ell}$, while $\mathcal{P}$ together sample a random $[\alpha_{\mathsf{v}}]_2 \in \mathbb{Z}_{2^\ell}$. Also, P_1, P_2 together sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$.
- If $P_i = P_2$: Symmetric to the case when $P_i = P_1$.

Online:

- If $P_i = P_0$: P_0 computes $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ and sends β_{v} to P_1. P_1, P_0 $\mathsf{jmp\text{-}send}$ β_{v} to P_2.
- If $P_i = P_j$, for $j \in \{1,2\}$: P_j computes $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$, sends β_{v} to P_{3-j}. P_1, P_2 $\mathsf{jmp\text{-}send}$ $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0.

Fig. 3: 3PC: Generating $[\![\mathsf{v}]\!]$-shares by server P_i

b) Joint Sharing Protocol: Protocol Π_{jsh} (Fig. 16) allows two servers P_i, P_j to jointly generate a $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is known to both. Towards this, servers execute the preprocessing of Π_{sh} (Fig. 3) to generate $[\alpha_{\mathsf{v}}]$ and γ_{v}. If $(P_i, P_j) = (P_1, P_0)$, then P_1, P_0 $\mathsf{jmp\text{-}send}$ $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to P_2. The case when $(P_i, P_j) = (P_2, P_0)$ proceeds similarly. The case for $(P_i, P_j) = (P_1, P_2)$ is optimized further as follows: servers locally set $[\alpha_{\mathsf{v}}]_1 = [\alpha_{\mathsf{v}}]_2 = 0$. P_1, P_2 together sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$, set $\beta_{\mathsf{v}} = \mathsf{v}$ and $\mathsf{jmp\text{-}send}$ $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0. We defer the formal details of Π_{jsh} to §B-C.

c) Addition Protocol: Given the $[\![\cdot]\!]$-shares on input wires x, y, servers can use the linearity property of the sharing scheme to locally compute $[\![\cdot]\!]$-shares of the output of addition gate, $\mathsf{z} = \mathsf{x} + \mathsf{y}$ as $[\![\mathsf{z}]\!] = [\![\mathsf{x}]\!] + [\![\mathsf{y}]\!]$.

d) Multiplication Protocol: Protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$ (Fig. 4) enables the servers in $\mathcal{P}$ to compute $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{xy}$, given the $[\![\cdot]\!]$-sharing of x and y. We build on the protocol of BLAZE [9] and discuss along the way the differences and resemblances. We begin with a protocol for the semi-honest setting, which is also the starting point of BLAZE. During the preprocessing phase, P_0, P_j for $j \in \{1,2\}$ sample random $[\alpha_{\mathsf{z}}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample random $\gamma_{\mathsf{z}} \in \mathbb{Z}_{2^\ell}$. In addition, P_0 locally computes $\Gamma_{\mathsf{xy}} = \alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$ and generates $[\cdot]$-sharing of the same between P_1, P_2. Since

$$\begin{aligned}\beta_{\mathsf{z}} &= \mathsf{z} + \alpha_{\mathsf{z}} = \mathsf{xy} + \alpha_{\mathsf{z}} = (\beta_{\mathsf{x}} - \alpha_{\mathsf{x}})(\beta_{\mathsf{y}} - \alpha_{\mathsf{y}}) + \alpha_{\mathsf{z}} \\ &= \beta_{\mathsf{x}}\beta_{\mathsf{y}} - \beta_x\alpha_{\mathsf{y}} - \beta_y\alpha_{\mathsf{x}} + \Gamma_{\mathsf{xy}} + \alpha_{\mathsf{z}} \qquad (1)\end{aligned}$$

holds, servers P_1, P_2 locally compute $[\beta_{\mathsf{z}}]_j = (j-1)\beta_{\mathsf{x}}\beta_{\mathsf{y}} - \beta_x [\alpha_{\mathsf{y}}]_j - \beta_y [\alpha_{\mathsf{x}}]_j + [\Gamma_{\mathsf{xy}}]_j + [\alpha_{\mathsf{z}}]_j$ during the online phase and mutually exchange their shares to reconstruct β_{z}. P_1 then sends $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0, completing the semi-honest protocol. The correctness that asserts $\mathsf{z} = \mathsf{xy}$ or in other words $\beta_{\mathsf{z}} - \alpha_{\mathsf{z}} = \mathsf{xy}$ holds due to Eq. 1.

The following issues arise in the above protocol when a malicious adversary is considered:

1) When P_0 is corrupt, the $[\cdot]$-sharing of Γ_{xy} performed by P_0 might not be correct, i.e. $\Gamma_{\mathsf{xy}} \neq \alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$.
2) When P_1 (or P_2) is corrupt, the $[\cdot]$-share of β_{z} handed over to the fellow honest evaluator during the online phase might not be correct, causing reconstruction of an incorrect β_{z}.
3) When P_1 is corrupt, the value $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ that is sent to P_0 during the online phase may not be correct.

All the three issues are common with BLAZE (copied verbatim), but we differ from BLAZE in handling them. We begin with solving the last issue first. We simply make P_1, P_2 jmp-send $\beta_z + \gamma_z$ to P_0 (after β_z is computed). This either leads to success or a TTP selection. Due to jmp's rate-1 communication, P_1 alone sending the value to P_0 remains as costly as using jmp in amortized sense. Whereas in BLAZE, the malicious version simply makes P_2 to send a hash of $\beta_z+\gamma_z$ to P_0 (in addition to P_1's communication of $\beta_z + \gamma_z$ to P_0), who aborts if the received values are inconsistent.

For the remaining two issues, similar to BLAZE, we reduce both to a multiplication (on values unrelated to inputs) in the preprocessing phase. However, our method leads to either success or TTP selection, with no additional cost.

We start with the second issue. To solve it, where a corrupt P_1 (or P_2) sends an incorrect $[\cdot]$-share of β_z, BLAZE makes use of server P_0 to compute a version of β_z for verification, based on β_x and β_y, as follows. Using $\beta_x + \gamma_x$ and $\beta_y + \gamma_y$, which are already available to P_0 as a part of $[\![x]\!], [\![y]\!]$, P_0 computes:

$$\begin{aligned}\beta_z^\star &= -(\beta_x+\gamma_x)\alpha_y - (\beta_y+\gamma_y)\alpha_x + 2\Gamma_{xy} + \alpha_z \\ &= (\beta_z - \beta_x\beta_y) - (\gamma_x\alpha_y + \gamma_y\alpha_x - \Gamma_{xy} + \psi) + \psi \qquad \text{[by Eq. 1]} \\ &= (\beta_z - \beta_x\beta_y + \psi) - \chi \quad \text{[where } \chi = \gamma_x\alpha_y + \gamma_y\alpha_x - \Gamma_{xy} + \psi]\end{aligned}$$

Assuming that (a) $\psi \in \mathbb{Z}_{2^\ell}$ is a random value sampled together by P_1 and P_2 (and unknown to P_0) for securing the β values from P_0 and (b) P_0 knows the value χ, P_0 can send $\beta_z^\star + \chi$ to P_1 and P_2 who using the knowledge of β_x, β_y and ψ can verify the correctness of β_z by computing $\beta_z - \beta_x\beta_y + \psi$ and checking against the value $\beta_z^\star + \chi$ received from P_0. The rest of the logic in BLAZE goes on to discuss how to enforce P_0– (a) to compute a correct χ (when honest) and (b) to share correct Γ_{xy} (when corrupt) via a single multiplication of two values in the preprocessing phase. Tying the ends together, one of their innovations goes in identifying the precise shared multiplication triple and mapping its components to χ and Γ_{xy} so that these are correct by the virtue of the correctness of the multiplication relation.

Protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, [\![x]\!], [\![y]\!])$

Preprocessing:

- P_0, P_j for $j \in \{1, 2\}$ together sample random $[\alpha_z]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample random $\gamma_z \in \mathbb{Z}_{2^\ell}$.
- Servers in $\mathcal{P}$ locally compute $\langle\cdot\rangle$-sharing of $d = \gamma_x + \alpha_x$ and $e = \gamma_y + \alpha_y$ by setting the shares as follows (ref. Table II):

  $(d_0{=}[\alpha_x]_2, d_1{=}[\alpha_x]_1, d_2{=}\gamma_x), \quad (e_0{=}[\alpha_y]_2, e_1{=}[\alpha_y]_1, e_2{=}\gamma_y)$
- Servers in $\mathcal{P}$ execute $\Pi_{\mathsf{mulPre}}(\mathcal{P}, d, e)$ to generate $\langle f\rangle = \langle de\rangle$.
- P_0, P_1 locally set $[\chi]_1 = f_1$, while P_0, P_2 locally set $[\chi]_2 = f_0$. P_1, P_2 locally compute $\psi = f_2 - \gamma_x\gamma_y$.

Online:

- P_0, P_j, for $j \in \{1, 2\}$, compute $[\beta_z^\star]_j = -(\beta_x + \gamma_x)[\alpha_y]_j - (\beta_y + \gamma_y)[\alpha_x]_j + [\alpha_z]_j + [\chi]_j$.
- P_0, P_1 jmp-send $[\beta_z^\star]_1$ to P_2 and P_0, P_2 jmp-send $[\beta_z^\star]_2$ to P_1.
- P_1, P_2 compute $\beta_z^\star = [\beta_z^\star]_1 + [\beta_z^\star]_2$ and set $\beta_z = \beta_z^\star + \beta_x\beta_y + \psi$.
- P_1, P_2 jmp-send $\beta_z + \gamma_z$ to P_0.

Fig. 4: 3PC: Multiplication Protocol ($z = x \cdot y$)

We differ from BLAZE in several ways. First, we do not simply rely on P_0 for the verification information $\beta_z^\star + \chi$, as this may inevitably lead to abort when P_0 is corrupt. Instead, we find (a slightly different) $\beta_z^\star$ that, instead of entirely available to P_0, will be available in $[\cdot]$-shared form between the two teams $\{\{P_0, P_i\}\}_{i\in\{1,2\}}$, with both servers in $\{P_0, P_i\}$ holding ith share $[\beta_z^\star]_i$. With this edit, the ith team can jmp-send the ith share of $\beta_z^\star$ to the third party which computes $\beta_z^\star$. Due to the presence of one honest party in each team, this $\beta_z^\star$ is correct and P_1, P_2 directly use it to compute β_z, with the knowledge of ψ, β_x, β_y. This means, departing from BLAZE and the starting semi-honest construction, P_1 and P_2 compute β_z from $\beta_z^\star$. Whereas, BLAZE suggests computing β_z from the exchange P_1, P_2's respective share of β_z and use $\beta_z^\star$ for verification. The outcome is a win-win situation i.e. either success or TTP selection. Our new $\beta_z^\star$ and χ are:

$$\begin{aligned}\chi &= \gamma_x\alpha_y + \gamma_y\alpha_x + \Gamma_{xy} - \psi \quad \text{and} \\ \beta_z^\star &= -(\beta_x+\gamma_x)\alpha_y - (\beta_y+\gamma_y)\alpha_x + \alpha_z + \chi \\ &= (-\beta_x\alpha_y - \beta_y\alpha_x + \Gamma_{xy} + \alpha_z) - \psi = \beta_z - \beta_x\beta_y - \psi\end{aligned}$$

Clearly, both P_0 and P_i can compute $[\beta_z^\star]_i = -(\beta_x + \gamma_x)[\alpha_y]_i - (\beta_y + \gamma_y)[\alpha_x]_i + [\alpha_z]_i + [\chi]_i$ given $[\chi_i]$. The rest of our discussion explains how (a) ith share of $[\chi]$ can be made available to $\{P_0, P_i\}$ and (b) ψ can be derived by P_1, P_2, from a multiplication triple. Similar to BLAZE, yet for a different triple, we observe that (d, e, f) is a multiplication triple, where $d = (\gamma_x + \alpha_x), e = (\gamma_y + \alpha_y), f = (\gamma_x\gamma_y + \psi) + \chi$ if and only if χ and Γ_{xy} are correct. Indeed,

$$\begin{aligned}de &= (\gamma_x+\alpha_x)(\gamma_y+\alpha_y) = \gamma_x\gamma_y + \gamma_x\alpha_y + \gamma_y\alpha_x + \Gamma_{xy} \\ &= (\gamma_x\gamma_y + \psi) + (\gamma_x\alpha_y + \gamma_y\alpha_x + \Gamma_{xy} - \psi) \\ &= (\gamma_x\gamma_y + \psi) + \chi = f\end{aligned}$$

Based on this observation, we compute the above multiplication triple using a multiplication protocol and extract out the values for ψ and χ from the shares of f which are bound to be correct. This can be executed entirely in the preprocessing phase. Specifically, the servers (a) locally obtain $\langle\cdot\rangle$-shares of d, e as in Table II, (b) compute $\langle\cdot\rangle$-shares of $f(= de)$, say denoted by f_0, f_1, f_2, using an efficient, robust 3-party multiplication protocol, say Π_{mulPre} (abstracted in a functionality Fig. 17) and finally (c) extract out the required preprocessing data *locally* as in Eq. 2. We switch to $\langle\cdot\rangle$-sharing in this part to be able to use the best robust multiplication protocol of [30] that supports this form of secret sharing and requires communication of just 3 elements. Fortunately, the switch does not cost anything, as both the first and third steps involve local computation and the cost simply reduces to a single run of a multiplication protocol.

$\langle v\rangle$	P_0 (v_0, v_1)	P_1 (v_1, v_2)	P_2 (v_2, v_0)
$\langle d\rangle$	$([\alpha_x]_2, [\alpha_x]_1)$	$([\alpha_x]_1, \gamma_x)$	$(\gamma_x, [\alpha_x]_2)$
$\langle e\rangle$	$([\alpha_y]_2, [\alpha_y]_1)$	$([\alpha_y]_1, \gamma_y)$	$(\gamma_y, [\alpha_y]_2)$

TABLE II: The $\langle\cdot\rangle$-sharing of values d and e

$$[\chi]_2 \leftarrow f_0, \quad [\chi]_1 \leftarrow f_1, \quad \gamma_x\gamma_y + \psi \leftarrow f_2. \qquad (2)$$

According to $\langle\cdot\rangle$-sharing, both P_0 and P_1 obtain f_1 and hence obtain $[\chi]_1$. Similarly, P_0, P_2 obtain f_0 and hence $[\chi]_2$. Finally,

P_1, P_2 obtain f_2 from which they compute $\psi = \mathsf{f}_2 - \gamma_{\mathsf{x}}\gamma_{\mathsf{y}}$. This completes the informal discussion.

We note that to facilitate a fast online phase for multiplication, our preprocessing phase leverages a robust multiplication protocol [30] in a black-box manner to derive the necessary preprocessing information. A similar black-box approach is also taken for the dot product protocol in the preprocessing phase. This leaves room for further improvements in communication cost, which can be obtained by instantiating the black-box with an efficient robust dot product protocol, coupled with the fast online phase.

e) Reconstruction Protocol: Protocol Π_{rec} (Fig. 5) allows servers to robustly reconstruct value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ from its $[\![\cdot]\!]$-shares. Note that each server misses one share of v which is held by the other two servers. Consider the case of P_0 who requires γ_{v} to compute v. During the preprocessing, P_1, P_2 compute a commitment of γ_{v}, denoted by $\mathsf{Com}(\gamma_{\mathsf{v}})$ and jmp-send the same to P_0. Similar steps are performed for the values $[\alpha_{\mathsf{v}}]_2$ and $[\alpha_{\mathsf{v}}]_1$ that are required by servers P_1 and P_2 respectively. During the online phase, servers open their commitments to the intended server who accepts the opening that is consistent with the agreed upon commitment.

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

Preprocessing:

- P_0, P_j, for $j \in \{1, 2\}$, compute $\mathsf{Com}([\alpha_{\mathsf{v}}]_j)$, while P_1, P_2 compute $\mathsf{Com}(\gamma_{\mathsf{v}})$.
- P_1, P_2 jmp-send $\mathsf{Com}(\gamma_{\mathsf{v}})$ to P_0, while P_0, P_1 jmp-send $\mathsf{Com}([\alpha_{\mathsf{v}}]_1)$ to P_2, and P_0, P_2 jmp-send $\mathsf{Com}([\alpha_{\mathsf{v}}]_2)$ to P_1,

Online:

- P_0, P_1 open $\mathsf{Com}([\alpha_{\mathsf{v}}]_1)$ to P_2. P_0, P_2 open $\mathsf{Com}([\alpha_{\mathsf{v}}]_2)$ to P_1. P_1, P_2 open $\mathsf{Com}(\gamma_{\mathsf{v}})$ to P_0.
- Each of the servers accept the opening that is consistent with the agreed upon commitment. P_1, P_2 compute $\mathsf{v} = \beta_{\mathsf{v}} - [\alpha_{\mathsf{v}}]_1 - [\alpha_{\mathsf{v}}]_2$, while P_0 computes $\mathsf{v} = (\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}) - [\alpha_{\mathsf{v}}]_1 - [\alpha_{\mathsf{v}}]_2 - \gamma_{\mathsf{v}}$.

Fig. 5: 3PC: Reconstruction of v among the servers

f) The Complete 3PC: For the sake of completeness and to demonstrate how GOD is achieved, we show how to compile the above primitives for a general 3PC. The main purpose is to explain the usage of jmp in a complete computation. A similar approach will be taken for 4PC and each PPML task (both training and inference) and we will avoid repetition. In order to compute an arithmetic circuit over $\mathbb{Z}_{2^\ell}$, we first invoke the key-setup functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 14) for key distribution and preprocessing of Π_{sh}, Π_{mult} and Π_{rec}, as per the given circuit. During the online phase, $P_i \in \mathcal{P}$ shares its input x_i by executing online steps of Π_{sh} (Fig. 3) protocol. This is followed by the circuit evaluation phase, where severs evaluate the gates in the circuit in the topological order, with addition gates (and multiplication-by-a-constant gates) being computed locally, and multiplication gates being computed via online of Π_{mult} (Fig. 4). Finally, servers run the online steps of Π_{rec} protocol (Fig. 5) on the output wires to reconstruct the function output. All the building blocks above invoke jmp, except the online phase of reconstruction. To leverage amortization, only *send* phases of all the jumps are run on the flow. At the end of preprocessing, and right before the reconstruction in the online phase, the *verify* phase for all possible ordered pair of senders are run. We carry on computation in the online phase only when the *verify* phases in the preprocessing are successful. Otherwise, the servers simply send their inputs to the elected TTP who computes the function and returns the result to all the servers. Similarly, depending on the output of the *verify* phase at the end of the online phase, either the reconstruction is carried out or a TTP is identified. In the latter case, computation completes as mentioned before.

C. Building Blocks for PPML using 3PC

This section provides details on robust realizations of the following building blocks for PPML in 3-server setting– i) Dot Product, ii) Truncation, iii) Dot Product with Truncation, iv) Secure Comparison, and v) Non-linear Activation functions– Sigmoid and ReLU. We begin by providing details of input sharing and reconstruction in the SOC setting.

a) Input Sharing and Output Reconstruction in the SOC Setting: Protocol $\Pi^{\mathsf{SOC}}_{\mathsf{sh}}$ (Fig. 6) extends input sharing to the SOC setting and allows a user U to generate the $[\![\cdot]\!]$-shares of its input v among the three servers. Note that the necessary commitments to facilitate the sharing are generated in the preprocessing phase by the servers which are then communicated to U, along with the opening, in the online phase. U selects the commitment forming the majority (for each share) owing to the presence of an honest majority among the servers, and accepts the corresponding shares. Analogously, protocol $\Pi^{\mathsf{SOC}}_{\mathsf{rec}}$ (Fig. 6) allows the servers to reconstruct a value v towards user U. In either of the protocols, if at any point, a TTP is identified, then servers signal the TTP's identity to U. U selects the TTP as the one forming a majority and sends its input in clear to the TTP, who computes the function output and sends it back to U.

Protocol $\Pi^{\mathsf{SOC}}_{\mathsf{sh}}(\mathsf{U}, \mathsf{v})$ and $\Pi^{\mathsf{SOC}}_{\mathsf{rec}}(\mathsf{U}, [\![\mathsf{v}]\!])$

Input Sharing:

- P_0, P_s, for $s \in \{1, 2\}$, together sample random $[\alpha_{\mathsf{v}}]_s \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 together sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$.
- P_0, P_1 jmp-send $\mathsf{Com}([\alpha_{\mathsf{v}}]_1)$ to P_2, while P_0, P_2 jmp-send $\mathsf{Com}([\alpha_{\mathsf{v}}]_2)$ to P_1, and P_1, P_2 jmp-send $\mathsf{Com}(\gamma_{\mathsf{v}})$ to P_0.
- Each of the servers sends $(\mathsf{Com}([\alpha_{\mathsf{v}}]_1), \mathsf{Com}([\alpha_{\mathsf{v}}]_2), \mathsf{Com}(\gamma_{\mathsf{v}}))$ to U who accepts the values that form the majority. Also, P_0, P_s, for $s \in \{1, 2\}$, open $[\alpha_{\mathsf{v}}]_s$ towards U while P_1, P_2 open γ_{v} towards U.
- U accepts the consistent opening, recovers $[\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2, \gamma_{\mathsf{v}}$, computes $\beta_{\mathsf{v}} = \mathsf{v} + [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$, and sends $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to all three servers.
- Servers broadcast the received value and accept the majority value if it exists, and a default value, otherwise. P_1, P_2 locally compute β_{v} from $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ using γ_{v} to complete the sharing of v.

Output Reconstruction:

- Servers execute the preprocessing of $\Pi_{\mathsf{rec}}(\mathcal{P}, [\![\mathsf{v}]\!])$ to agree upon commitments of $[\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2$ and γ_{v}.
- Each of the servers send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ as well as commitments on $[\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2$ and γ_{v} to U, who accepts the values forming majority.
- Now, P_0, P_1 open $[\alpha_{\mathsf{v}}]_1$ to U, P_0, P_2 open $[\alpha_{\mathsf{v}}]_2$, while P_1, P_2 open γ_{v} to U.
- U accepts the consistent opening and computes $\mathsf{v} = (\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}) - [\alpha_{\mathsf{v}}]_1 - [\alpha_{\mathsf{v}}]_2 - \gamma_{\mathsf{v}}$.

Fig. 6: 3PC: Input Sharing and Output Reconstruction in SOC Setting

b) MSB Extraction, Bit to Arithmetic Conversion and Bit Injection Protocols: Here we provide a high-level overview of three protocols that involve working over arithmetic and boolean rings in a mixed fashion and are used for the PPML primitives. The *Bit Extraction Protocol*, Π_{bitext} allows servers to compute the *boolean* sharing of the most significant bit (msb) of a value v given its arithmetic sharing $[\![\mathsf{v}]\!]$. The *Bit2A Protocol*, Π_{bit2A}, given the boolean sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, allows servers to compute the arithmetic sharing $[\![\mathsf{b}^{\mathsf{R}}]\!]$. Here b^{R} denotes the equivalent value of b over ring $\mathbb{Z}_{2^\ell}$ (see Notation II.2). Lastly, the *Bit Injection Protocol*, Π_{BitInj}, allows the servers to compute the arithmetic sharing $[\![\mathsf{bv}]\!]$ given the boolean sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$ and the arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$.

The core techniques used in these protocols follow from BLAZE [9], where the multiplication calls are instantiated with Π_{mult}, and several private communications are replaced with jmp-send to ensure either successful run or TTP selection. The PPML building blocks above can be understood without the details of the constructs and hence they are moved to §B-F.

c) Dot Product: Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol Π_{dotp} (Fig. 7) allows servers to generate $[\![\cdot]\!]$-sharing of $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ in a robust fashion. By $[\![\cdot]\!]$-sharing of a vector $\vec{\mathbf{x}}$ of size n, we mean that each element $\mathsf{x}_i \in \mathbb{Z}_{2^\ell}$ in the vector, for $i \in [\mathsf{n}]$, is $[\![\cdot]\!]$-shared. We borrow ideas from BLAZE for obtaining an online communication cost *independent* of n and use jmp primitive to ensure either success or TTP selection. Similar to our multiplication protocol that offloads one call to a robust multiplication protocol in the preprocessing, our dot product does the same for a robust dot product.

Protocol $\Pi_{\mathsf{dotp}}(\mathcal{P}, \{[\![\mathsf{x}_i]\!], [\![\mathsf{y}_i]\!]\}_{i \in [\mathsf{n}]})$

Preprocessing:

- P_0, P_j, for $j \in \{1,2\}$, together sample random $[\alpha_{\mathsf{z}}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample random $\gamma_{\mathsf{z}} \in \mathbb{Z}_{2^\ell}$.
- Servers locally compute $\langle\cdot\rangle$-sharing of $\vec{\mathbf{d}}, \vec{\mathbf{e}}$ with $\mathsf{d}_i = \gamma_{\mathsf{x}_i} + \alpha_{\mathsf{x}_i}$ and $\mathsf{e}_i = \gamma_{\mathsf{y}_i} + \alpha_{\mathsf{y}_i}$ for $i \in [\mathsf{n}]$ as follows:

$$(\langle \mathsf{d}_i \rangle_0 = ([\alpha_{\mathsf{x}_i}]_2, [\alpha_{\mathsf{x}_i}]_1), \langle \mathsf{d}_i \rangle_1 = ([\alpha_{\mathsf{x}_i}]_1, \gamma_{\mathsf{x}_i}), \langle \mathsf{d}_i \rangle_2 = (\gamma_{\mathsf{x}_i}, [\alpha_{\mathsf{x}_i}]_2))$$
$$(\langle \mathsf{e}_i \rangle_0 = ([\alpha_{\mathsf{y}_i}]_2, [\alpha_{\mathsf{y}_i}]_1), \langle \mathsf{e}_i \rangle_1 = ([\alpha_{\mathsf{y}_i}]_1, \gamma_{\mathsf{y}_i}), \langle \mathsf{e}_i \rangle_2 = (\gamma_{\mathsf{y}_i}, [\alpha_{\mathsf{y}_i}]_2))$$

- Servers execute $\Pi_{\mathsf{dotpPre}}(\mathcal{P}, \langle \vec{\mathbf{d}} \rangle, \langle \vec{\mathbf{e}} \rangle)$ to generate $\langle \mathsf{f} \rangle = \langle \vec{\mathbf{d}} \odot \vec{\mathbf{e}} \rangle$.
- P_0, P_1 locally set $[\chi]_1 = \mathsf{f}_1$, while P_0, P_2 locally set $[\chi]_2 = \mathsf{f}_0$. P_1, P_2 locally compute $\psi = \mathsf{f}_2 - \sum_{i=1}^{\mathsf{n}} \gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i}$.

Online:

- P_0, P_j, for $j \in \{1,2\}$, compute $[\beta^\star_{\mathsf{z}}]_j = -\sum_{i=1}^{\mathsf{n}}((\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})[\alpha_{\mathsf{y}_i}]_j + (\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})[\alpha_{\mathsf{x}_i}]_j) + [\alpha_{\mathsf{z}}]_j + [\chi]_j$.
- P_0, P_1 jmp-send $[\beta^\star_{\mathsf{z}}]_1$ to P_2 and P_0, P_2 jmp-send $[\beta^\star_{\mathsf{z}}]_2$ to P_1.
- P_1, P_2 locally compute $\beta^\star_{\mathsf{z}} = [\beta^\star_{\mathsf{z}}]_1 + [\beta^\star_{\mathsf{z}}]_2$ and set $\beta_{\mathsf{z}} = \beta^\star_{\mathsf{z}} + \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i}) + \psi$.
- P_1, P_2 jmp-send $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0.

Fig. 7: 3PC: Dot Product Protocol ($\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$)

To begin with, $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ can be viewed as n parallel multiplication instances of the form $\mathsf{z}_i = \mathsf{x}_i\mathsf{y}_i$ for $i \in \{1, \ldots, \mathsf{n}\}$, followed by adding up the results. Let $\beta^\star_{\mathsf{z}} = \sum_{i=1}^{\mathsf{n}} \beta^\star_{\mathsf{z}_i}$. Then,

$$\beta^\star_{\mathsf{z}} = -\sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})\alpha_{\mathsf{y}_i} - \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})\alpha_{\mathsf{x}_i} + \alpha_{\mathsf{z}} + \chi \quad (3)$$

where $\chi = \sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\alpha_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i}\alpha_{\mathsf{x}_i} + \Gamma_{\mathsf{x}_i\mathsf{y}_i} - \psi_i)$.

Apart from the aforementioned modification, the online phase for dot product proceeds similar to that of multiplication protocol. P_0, P_1 locally compute $[\beta^\star_{\mathsf{z}}]_1$ as per Eq. 3 and jmp-send $[\beta^\star_{\mathsf{z}}]_1$ to P_2. P_1 obtains $[\beta^\star_{\mathsf{z}}]_2$ in a similar fashion. P_1, P_2 reconstruct $\beta^\star_{\mathsf{z}} = [\beta^\star_{\mathsf{z}}]_1 + [\beta^\star_{\mathsf{z}}]_2$ and compute $\beta_{\mathsf{z}} = \beta^\star_{\mathsf{z}} + \sum_{i=1}^{\mathsf{n}} \beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i} + \psi$. Here, the value ψ has to be correctly generated in the preprocessing phase satisfying Eq. 3. Finally, P_1, P_2 jmp-send $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0.

We now provide the details for preprocessing phase that enable servers to obtain the required values (χ, ψ) with the invocation of a dot product protocol in a black-box way. Towards this, let $\vec{\mathbf{d}} = [\mathsf{d}_1, \ldots, \mathsf{d}_{\mathsf{n}}]$ and $\vec{\mathbf{e}} = [\mathsf{e}_1, \ldots, \mathsf{e}_{\mathsf{n}}]$, where $\mathsf{d}_i = \gamma_{\mathsf{x}_i} + \alpha_{\mathsf{x}_i}$ and $\mathsf{e}_i = \gamma_{\mathsf{y}_i} + \alpha_{\mathsf{y}_i}$ for $i \in [\mathsf{n}]$, as in the case of multiplication. Then for $\mathsf{f} = \vec{\mathbf{d}} \odot \vec{\mathbf{e}}$,

$$\begin{aligned}
\mathsf{f} = \vec{\mathbf{d}} \odot \vec{\mathbf{e}} &= \sum_{i=1}^{\mathsf{n}} \mathsf{d}_i\mathsf{e}_i = \sum_{i=1}^{\mathsf{n}} (\gamma_{\mathsf{x}_i} + \alpha_{\mathsf{x}_i})(\gamma_{\mathsf{y}_i} + \alpha_{\mathsf{y}_i}) \\
&= \sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i} + \psi_i) + \sum_{i=1}^{\mathsf{n}} \chi_i = \sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i} + \psi_i) + \chi \\
&= \sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i} + \psi_i) + [\chi]_1 + [\chi]_2 = \mathsf{f}_2 + \mathsf{f}_1 + \mathsf{f}_0.
\end{aligned}$$

where $\mathsf{f}_2 = \sum_{i=1}^{\mathsf{n}}(\gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i} + \psi_i), \mathsf{f}_1 = [\chi]_1$ and $\mathsf{f}_0 = [\chi]_2$.

Using the above relation, the preprocessing phase proceeds as follows: P_0, P_j for $j \in \{1,2\}$ sample a random $[\alpha_{\mathsf{z}}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample random γ_{z}. Servers locally prepare $\langle \vec{\mathbf{d}} \rangle, \langle \vec{\mathbf{e}} \rangle$ similar to that of multiplication protocol. Servers then execute a robust 3PC dot product protocol, denoted by Π_{dotpPre}, that takes $\langle \vec{\mathbf{d}} \rangle, \langle \vec{\mathbf{e}} \rangle$ as input and compute $\langle \mathsf{f} \rangle$ with $\mathsf{f} = \vec{\mathbf{d}} \odot \vec{\mathbf{e}}$. Given $\langle \mathsf{f} \rangle$, the ψ and $[\chi]$ values are extracted as follows (ref. Eq. 4):

$$\psi = \mathsf{f}_2 - \sum_{i=1}^{\mathsf{n}} \gamma_{\mathsf{x}_i}\gamma_{\mathsf{y}_i}, \quad [\chi]_1 = \mathsf{f}_1, \quad [\chi]_2 = \mathsf{f}_0, \qquad (4)$$

It is easy to see from the semantics of $\langle\cdot\rangle$-sharing that both P_1, P_2 obtain f_2 and hence ψ. Similarly, both P_0, P_1 obtain f_1 and hence $[\chi]_1$, while P_0, P_2 obtain $[\chi]_2$.

In this work, we instantiate Π_{dotpPre} using n black-box invocations of Π_{mulPre}. In the i^{th} invocation, servers compute $\langle \mathsf{d}_i, \mathsf{e}_i \rangle$ and obtain the respective ψ_i and $[\chi_i]$ values. Finally, servers locally set $\psi = \sum_{i=1}^{\mathsf{n}} \psi_i$ and $[\chi] = \sum_{i=1}^{\mathsf{n}} [\chi_i]$. The aforementioned method results in communication of $3\mathsf{n}$ elements in the preprocessing phase. A protocol for Π_{dotpPre} with better cost, when plugged into our Π_{dotp}, would bring down the preprocessing cost further while maintaining a fast online phase. We defer the formal details of Π_{dotp} to §B-G.

d) Truncation: Working over fixed-point values, repeated multiplications using FPA arithmetic can lead to an overflow resulting in loss of significant bits of information. This put forth the need for truncation [1], [4], [6], [7], [9] that re-adjusts the shares after multiplication so that FPA semantics are maintained. As shown in SecureML [1], the method of truncation would result in loss of information on the least significant bits and affect the accuracy by a very minimal amount only.

For truncation, servers execute Π_{trgen} (Fig. 8) to generate a random pair of the form $([\mathsf{r}], [\![\mathsf{r}^d]\!])$. Here, r denotes a random ring element, while r^d represents the truncated value of r. By truncated value, we mean that the value is right-shifted by d bit positions, where d is the number of bits allocated for the fractional part in the FPA representation. Given $(\mathsf{r}, \mathsf{r}^d)$, the truncated value of v denoted by v^d can be computed from v as $\mathsf{v}^d = (\mathsf{v} - \mathsf{r})^d + \mathsf{r}^d$. As shown in ABY3 [4], this method ensures the same correctness of SecureML and the accuracy is affected by a very minimal amount.

To generate $([\mathsf{r}], [\![\mathsf{r}^d]\!])$, servers proceed as follows: P_0, P_j for $j \in \{1,2\}$ sample random $R_j \in \mathbb{Z}_{2^\ell}$. P_0 locally computes $\mathsf{r} = R_1 + R_2$ and truncates r to obtain r^d. P_0 then executes Π_{sh} on r^d to generate $[\![\mathsf{r}^d]\!]$. As shown in BLAZE, the correctness of sharing performed by P_0 is checked using the relation $\mathsf{r} = 2^d \mathsf{r}^d + \mathsf{r}_d$, where r_d denotes the ring element r with the higher order $\ell - d$ bit positions set to 0. In detail, P_0, P_j for $j \in \{1,2\}$ locally computes $[\mathsf{a}_j]$ for $\mathsf{a} = (\mathsf{r} - 2^d \mathsf{r}^d + \mathsf{r}_d)$. P_0, P_1 then jmp-send $\mathsf{H}([\mathsf{a}]_1)$ to P_2. P_2 checks if the received hash value matches with $\mathsf{H}(-[\mathsf{a}]_2)$. In case of any inconsistency, P_2 accuses P_0 and then P_1 is identified as the TTP. The correctness of Π_{trgen} follows from BLAZE.

Protocol $\Pi_{\mathsf{trgen}}(\mathcal{P})$

- P_0, P_j for $j \in \{1,2\}$ together sample random $R_j \in \mathbb{Z}_{2^\ell}$. P_0 sets $\mathsf{r} = R_1 + R_2$ while P_j sets $[\mathsf{r}]_j = R_j$. P_j sets $[\mathsf{r}_d]_j$ as the ring element that has last d bits of r_j in the last d positions and 0 elsewhere.
- P_0 locally truncates r to obtain r^d and executes $\Pi_{\mathsf{sh}}(P_0, \mathsf{r}^d)$ to generate $[\![\mathsf{r}^d]\!]$.
- P_0, P_1 set $[\mathsf{r}^d]_1 = \beta_{\mathsf{r}^d} - [\alpha_{\mathsf{r}^d}]_1$, while P_0, P_2 set $[\mathsf{r}^d]_2 = -[\alpha_{\mathsf{r}^d}]_2$.
- P_0, P_1 compute $\mathsf{u} = [\mathsf{r}]_1 - 2^d [\mathsf{r}^d]_1 - [\mathsf{r}_d]_1$. P_0, P_1 jmp-send $\mathsf{H}(\mathsf{u})$) to P_2.
- P_2 locally computes $\mathsf{v} = 2^d [\mathsf{r}^d]_2 + [\mathsf{r}_d]_2 - [\mathsf{r}]_2$. If $\mathsf{H}(\mathsf{u}) \neq \mathsf{H}(\mathsf{v})$, P_2 `broadcast "(accuse,` P_0`)"` and P_1 is chosen as the TTP.

Fig. 8: 3PC: Generating Random Truncated Pair $(\mathsf{r}, \mathsf{r}^d)$

e) Dot Product with Truncation: Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$, protocol Π_{dotpt} (Fig. 20) allows servers to generate $[\![\mathsf{z}^d]\!]$, where z^d denotes the truncated value of $\mathsf{z} = \vec{\mathsf{x}} \odot \vec{\mathsf{y}}$. One naive way is to compute the dot product using Π_{dotp} first, followed by performing the truncation using the $(\mathsf{r}, \mathsf{r}^d)$ pair. Instead, we follow the optimization of BLAZE where the online phase of Π_{dotp} is modified to integrate the truncation using $(\mathsf{r}, \mathsf{r}^d)$ at no additional cost.

The preprocessing phase now consists of the execution of one instance of Π_{trgen} (Fig. 8) and the preprocessing corresponding to Π_{dotp} (Fig. 7). At a high level, the online phase proceeds as follows: P_0, P_j for $j \in \{1,2\}$ locally compute $[\mathsf{z}^\star - \mathsf{r}]_j$ (instead of $[\beta^\star_{\mathsf{z}}]_j$ as in Π_{dotp}) where $\mathsf{z}^\star = \beta^\star_{\mathsf{z}} - \alpha_{\mathsf{z}}$. P_0, P_1 jmp-send $[\mathsf{z}^\star - \mathsf{r}]_1$ to P_2 while P_0, P_2 jmp-send $[\mathsf{z}^\star - \mathsf{r}]_2$ to P_1. Both P_1, P_2 then compute $(\mathsf{z} - \mathsf{r})$ locally, truncate it to obtain $(\mathsf{z} - \mathsf{r})^d$ and execute Π_{jsh} to generate $[\![(\mathsf{z} - \mathsf{r})^d]\!]$. Finally, servers locally compute the result as $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z} - \mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$. We defer the formal details of the protocol Π_{dotpt} to §B-I.

f) Secure Comparison: Secure comparison allows servers to check whether $\mathsf{x} < \mathsf{y}$, given their $[\![\cdot]\!]$-shares. In FPA representation, checking $\mathsf{x} < \mathsf{y}$ is equivalent to checking the msb of $\mathsf{v} = \mathsf{x} - \mathsf{y}$. Towards this, servers locally compute $[\![\mathsf{v}]\!] = [\![\mathsf{x}]\!] - [\![\mathsf{y}]\!]$ and extract the msb of v using Π_{bitext} (§B-F1). In case an arithmetic sharing is desired, servers can apply Π_{bit2A} (Fig. 18) protocol on the outcome of Π_{bitext} protocol.

g) Activation Functions: We now elaborate on two of the most prominently used activation functions: i) Rectified Linear Unit (ReLU) and (ii) Sigmoid (Sig).

– ReLU: The ReLU function, $\mathsf{relu}(\mathsf{v}) = \mathsf{max}(0, \mathsf{v})$, can be viewed as $\mathsf{relu}(\mathsf{v}) = \overline{\mathsf{b}} \cdot \mathsf{v}$, where the bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. Here $\overline{\mathsf{b}}$ denotes the complement of b. Given $[\![\mathsf{v}]\!]$, servers first execute Π_{bitext} on $[\![\mathsf{v}]\!]$ to generate $[\![\mathsf{b}]\!]^{\mathbf{B}}$. The $[\![\cdot]\!]^{\mathbf{B}}$-sharing of $\overline{\mathsf{b}}$ is then locally computed by setting $\beta_{\overline{\mathsf{b}}} = 1 \oplus \beta_{\mathsf{b}}$. Servers then execute Π_{BitInj} protocol on $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$ to obtain the desired result.

– Sig: In this work, we use the MPC-friendly variant of the Sigmoid function [1], [4], [6] (ref. §B-J). Note that $\mathsf{sig}(\mathsf{v}) = \overline{\mathsf{b}_1}\mathsf{b}_2(\mathsf{v} + 1/2) + \overline{\mathsf{b}_2}$, where $\mathsf{b}_1 = 1$ if $\mathsf{v} + 1/2 < 0$ and $\mathsf{b}_2 = 1$ if $\mathsf{v} - 1/2 < 0$. To compute $[\![\mathsf{sig}(\mathsf{v})]\!]$, servers proceed in a similar fashion as the ReLU, and hence, we skip the formal details.

IV. Robust 4PC and PPML

In this section, we extend our 3PC results to the 4-party case and observe substantial efficiency gain. First, the use of broadcast is eliminated. Second, the preprocessing of multiplication becomes substantially computationally light, eliminating the multiplication protocol altogether. Third and the most striking of all, we achieve a dot product protocol with communication cost *completely independent* of the size of the vector, as opposed to its 3PC counterpart (cf. Π_{dotp} (Fig. 7)). At the heart of our 4PC constructions lies an efficient 4-party jmp primitive, denoted as jmp4, that allows two servers to robustly send a common value to a third server. We start with the secret-sharing semantics for 4 parties. We only use an extended version of $[\![\cdot]\!]$-sharing defined below.

a) Secret Sharing Semantics: For a value v, the shares for P_0, P_1 and P_2 remain the same as that of 3PC case. That is, P_0 holds $([\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2, \beta_{\mathsf{v}} + \gamma_{\mathsf{v}})$ while P_i for $i \in \{1,2\}$ holds $([\alpha_{\mathsf{v}}]_i, \beta_{\mathsf{v}}, \gamma_{\mathsf{v}})$. The shares for the fourth server P_3 is defined as $([\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2, \gamma_{\mathsf{v}})$. Clearly, the secret is defined as $\mathsf{v} = \beta_{\mathsf{v}} - [\alpha_{\mathsf{v}}]_1 - [\alpha_{\mathsf{v}}]_2$.

b) 4PC Joint Message Passing Primitive: The jmp4 primitive enables two servers P_i, P_j to send a common value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to a third server P_k, or identify a TTP in case of any inconsistency. This primitive is analogous to jmp (Fig. 2) in spirit but is significantly optimized and free from broadcast calls. Similar to the 3PC counterpart, each party maintains a bit and P_i sends the value, and P_j the hash of it to P_k. P_k sets its inconsistency bit to 1 when the (value, hash) pair is inconsistent. This is followed by relaying the bit back to the senders who exchange it to reach an agreement on the consistency bit. During this execution, if a party remains silent, then it triggers the recipient to turn on its bit. In case of any inconsistency, the fourth server, who was not a part of the computation, can be employed as the TTP. However, reaching agreement on whether the TTP is established or the protocol terminates successfully needs extra care. For this, P_i, P_j, P_k send their bits to P_l who accepts to be a TTP when at least two parties' bits are turned on including P_k's. P_l relays a confirmation to all and a server accepts her if its inconsistency bit is on and it receives the confirmation from P_l. Notice that an honest P_l when it relays a confirmation will always be accepted and a corrupt P_l's deliberate attempt to become TTP will always be rejected.

Protocol $\Pi_{\mathsf{jmp4}}(P_i, P_j, P_k, \mathsf{v}, P_l)$

– P_s for $s \in \{i, j, k\}$ initializes an inconsistency bit $\mathsf{b}_s = 0$.
– P_i sends v to P_k and 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 either P_i or P_j remained silent.
– P_k sends b_k to P_i, P_j. P_i sets b_i to b_k and to 1 when nothing is received from P_k. Similarly, for P_j.
– P_i, P_j mutually exchange their bits. P_i resets $\mathsf{b}_i = \mathsf{b}_i \vee \mathsf{b}_j$ where b_j is set to 1 when P_j remains silent. Analogously for P_j.
– P_s for $s \in \{i, j, k\}$ sends b_s to P_l. If P_l receives 1 from at least two servers among which one is P_k, P_l sets $\mathsf{b}_l = 1$ and to 0 otherwise. It sends b_l to all.
– P_s, for $s \in \{i, j, k\}$, sets $\mathsf{TTP} = P_l$ if $\mathsf{b}_s \wedge \mathsf{b}_l = 1$, terminates otherwise.

Fig. 9: 4PC: Joint Message Passing Primitive

Notation IV.1. We say that P_i, P_j jmp4-send v to P_k when they invoke $\Pi_{\mathsf{jmp4}}(P_i, P_j, P_k, \mathsf{v}, P_l)$.

We note that the end goal of jmp4 primitive relates closely to the bi-convey primitive of FLASH [7]. Bi-convey allows two servers S_1, S_2 to convey a value to a server R, and in case of an inconsistency, a pair of honest servers mutually identify each other, followed by exchanging their internal randomness to recover the clear inputs, computing the circuit, and sending the output to all. Note, however, that jmp4 primitive is more efficient and differs significantly in techniques from the bi-convey primitive. Unlike in bi-convey, in case of an inconsistency, jmp4 enables servers to unanimously learn the TTP's identity. Moreover, bi-convey demands that honest servers, identified during an inconsistency, exchange their internal randomness (which comprises of the shared keys established during the key-setup phase) to proceed with the computation. This enforces the need for a fresh key-setup every time inconsistency is detected. In the efficiency front, jmp4 simply halves the communication cost of bi-convey, giving a $2\times$ improvement.

A. 4PC Protocols

In this section, we revisit the protocols from 3PC (§III) and suggest optimizations. While we provide details for the protocols that vary significantly from their 3PC counterpart in this section, the details for other protocols are deferred to §C.

a) Sharing Protocol: To enable P_i to share a value v, protocol Π_{sh4} (Fig. 10) proceeds similar to that of 3PC case with the addition that P_3 also samples the values $[\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2, \gamma_{\mathsf{v}}$ using the shared randomness with the respective servers. On a high level, P_i computes $\beta_{\mathsf{v}} = \mathsf{v} + [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$ and sends β_{v} (or $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$) to another server and they together jmp4-send this information to the intended servers.

Protocol $\Pi_{\mathsf{sh4}}(P_i, \mathsf{v})$

Preprocessing:

– If $P_i = P_0$: P_0, P_3, P_j, for $j \in \{1, 2\}$, together sample random $[\alpha_{\mathsf{v}}]_j \in \mathbb{Z}_{2^\ell}$, while $\mathcal{P}$ sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$.
– If $P_i = P_1$: P_0, P_3, P_1 together sample random $[\alpha_{\mathsf{v}}]_1 \in \mathbb{Z}_{2^\ell}$, while $\mathcal{P}$ sample a random $[\alpha_{\mathsf{v}}]_2 \in \mathbb{Z}_{2^\ell}$. Also, P_1, P_2, P_3 sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$.
– If $P_i = P_2$: Analogous to the case when $P_i = P_1$.
– If $P_i = P_3$: P_0, P_3, P_j, for $j \in \{1, 2\}$, sample random $[\alpha_{\mathsf{v}}]_j \in \mathbb{Z}_{2^\ell}$. P_1, P_2, P_3 together sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$.

Online:

– If $P_i = P_0$: P_0 computes $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ and sends β_{v} to P_1. P_0, P_1 jmp4-send β_{v} to P_2.
– If $P_i = P_j$, for $j \in \{1, 2\}$: P_j computes $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$, sends β_{v} to P_{3-j}. P_1, P_2 jmp4-send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0.
– If $P_i = P_3$: P_3 sends $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0. P_3, P_0 jmp4-send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to both P_1 and P_2.

Fig. 10: 4PC: Generating $[\![\mathsf{v}]\!]$-shares by server P_i

b) Multiplication Protocol: Given the $[\![\cdot]\!]$-shares of x and y, protocol Π_{mult4} (Fig. 11) allows servers to compute $[\![\mathsf{z}]\!]$ with $\mathsf{z} = \mathsf{xy}$. When compared with the state-of-the-art 4PC GOD protocol of FLASH [7], our solution improves communication in both, the preprocessing and online phase, from 6 to 3 ring elements. Moreover, our communication cost matches with the state-of-the-art 4PC protocol of Trident [8] that provides security with fairness only.

Protocol $\Pi_{\mathsf{mult4}}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$

Preprocessing:

– P_0, P_3, P_j, for $j \in \{1, 2\}$, sample random $[\alpha_{\mathsf{z}}]_j \in \mathbb{Z}_{2^\ell}$, while P_0, P_1, P_3 sample random $[\Gamma_{\mathsf{xy}}]_1 \in \mathbb{Z}_{2^\ell}$.
– P_1, P_2, P_3 sample random $\gamma_{\mathsf{z}}, \psi, \mathsf{r} \in \mathbb{Z}_{2^\ell}$ and set $[\psi]_1 = \mathsf{r}$, $[\psi]_2 = \psi - \mathsf{r}$.
– P_0, P_3 set $[\Gamma_{\mathsf{xy}}]_2 = \Gamma_{\mathsf{xy}} - [\Gamma_{\mathsf{xy}}]_1$, where $\Gamma_{\mathsf{xy}} = \alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$. P_0, P_3 jmp4-send $[\Gamma_{\mathsf{xy}}]_2$ to P_2.
– P_3, P_j, for $j \in \{1, 2\}$, set $[\chi]_j = \gamma_{\mathsf{x}} [\alpha_{\mathsf{y}}]_j + \gamma_{\mathsf{y}} [\alpha_{\mathsf{x}}]_j + [\Gamma_{\mathsf{xy}}]_j - [\psi]_j$. P_1, P_3 jmp4-send $[\chi]_1$ to P_0, while P_2, P_3 jmp4-send $[\chi]_2$ to P_0.

Online:

– P_0, P_j, for $j \in \{1, 2\}$, compute $[\beta^\star_{\mathsf{z}}]_j = -(\beta_{\mathsf{x}} + \gamma_{\mathsf{x}}) [\alpha_{\mathsf{y}}]_j - (\beta_{\mathsf{y}} + \gamma_{\mathsf{y}}) [\alpha_{\mathsf{x}}]_j + [\alpha_{\mathsf{z}}]_j + [\chi]_j$.
– P_1, P_0 jmp4-send $[\beta^\star_{\mathsf{z}}]_1$ to P_2, while P_2, P_0 jmp4-send $[\beta^\star_{\mathsf{z}}]_2$ to P_1.
– P_j, for $j \in \{1, 2\}$, computes $\beta^\star_{\mathsf{z}} = [\beta^\star_{\mathsf{z}}]_1 + [\beta^\star_{\mathsf{z}}]_2$ and sets $\beta_{\mathsf{z}} = \beta^\star_{\mathsf{z}} + \beta_{\mathsf{x}}\beta_{\mathsf{y}} + \psi$.
– P_1, P_2 jmp4-send $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0.

Fig. 11: 4PC: Multiplication Protocol ($\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$)

Recall that the goal of preprocessing in 3PC multiplication was to enable P_1, P_2 obtain ψ, and P_0, P_i for $i \in \{1, 2\}$ obtain $[\chi]_i$ where $\chi = \gamma_{\mathsf{x}}\alpha_{\mathsf{y}} + \gamma_{\mathsf{y}}\alpha_{\mathsf{x}} + \Gamma_{\mathsf{xy}} - \psi$. Here ψ is a random value known to both P_1, P_2. With the help of P_3, we let the servers obtain the respective preprocessing data as follows: P_0, P_3, P_1 together samples random $[\Gamma_{\mathsf{xy}}]_1 \in \mathbb{Z}_{2^\ell}$. P_0, P_3 locally compute $\Gamma_{\mathsf{xy}} = \alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$, set $[\Gamma_{\mathsf{xy}}]_2 = \Gamma_{\mathsf{xy}} - [\Gamma_{\mathsf{xy}}]_1$ and jmp4-send $[\Gamma_{\mathsf{xy}}]_2$ to P_2. P_1, P_2, P_3 locally sample ψ, r and generate $[\cdot]$-shares of ψ by setting $[\psi]_1 = r$ and $[\psi]_2 = \psi - \mathsf{r}$. Then P_j, P_3 for $j \in \{1, 2\}$ compute $[\chi]_j = \gamma_{\mathsf{x}} [\alpha_{\mathsf{y}}]_j + \gamma_{\mathsf{y}} [\alpha_{\mathsf{x}}]_j + [\Gamma_{\mathsf{xy}}]_j - [\psi]_j$ and jmp4-send $[\chi]_j$ to P_0. The online phase is similar to that of 3PC, apart from Π_{jmp4} being used instead of Π_{jmp} for communication. Since P_3 is not involved in the online computation phase, we can safely assume P_3 to serve as the TTP for the Π_{jmp4} executions in the online phase.

c) Reconstruction Protocol: Given $[\![\mathsf{v}]\!]$, protocol Π_{rec4} (Fig. 12) enables servers to robustly reconstruct the value v among the servers. Note that every server lacks one share for reconstruction and the same is available with three other servers. Hence, they communicate the missing share among themselves, and the majority value is accepted. As an optimization, two among the three servers can send the missing share while the third one can send a hash of the same for verification. Notice that, as opposed to the 3PC case, this protocol does not require commitments.

Protocol $\Pi_{\mathsf{rec4}}(\mathcal{P}, [\![\mathsf{v}]\!])$

Online

- P_0 receives γ_{v} from P_1, P_2 and $\mathsf{H}(\gamma_{\mathsf{v}})$ from P_3.
- P_1 receives $[\alpha_{\mathsf{v}}]_2$ from P_2, P_3 and $\mathsf{H}([\alpha_{\mathsf{v}}]_2)$ from P_0.
- P_2 receives $[\alpha_{\mathsf{v}}]_1$ from P_0, P_3 and $\mathsf{H}([\alpha_{\mathsf{v}}]_1)$ from P_1.
- P_3 receives $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ from P_0, P_1 and $\mathsf{H}(\beta_{\mathsf{v}} + \gamma_{\mathsf{v}})$ from P_2.
- $P_i \in \mathcal{P}$ selects the missing share forming the majority among the values received and reconstructs the output.

Fig. 12: 4PC: Reconstruction of v among the servers

d) Input Sharing and Output Reconstruction in SOC Setting: We extend input sharing and reconstruction in the SOC setting as follows. To generate $[\![\cdot]\!]$-shares for its input v, U receives each of the shares $[\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2$, and γ_{v} from three out of the four servers as well as a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ sampled together by P_0, P_1, P_2 and accepts the values that form the majority. U locally computes $\mathsf{u} = \mathsf{v} + [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2 + \gamma_{\mathsf{v}} + \mathsf{r}$ and sends u to all the servers. Servers then execute a two round byzantine agreement (BA) [47] to agree on u (or $\perp$). On successful completion of BA, P_0 computes $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ from u while P_1, P_2 compute β_{v} from u locally. For the reconstruction of a value v, servers send their $[\![\cdot]\!]$-shares of v to U, who selects the majority value for each share and reconstructs the output. At any point, if a TTP is identified, the servers proceed as follows. All servers send their $[\![\cdot]\!]$-share of the input to the TTP. TTP picks the majority value for each share and computes the function output. It then sends this output to U. U also receives the identity of the TTP from all servers and accepts the output received from the TTP forming majority.

e) Dot Product: Given $[\![\cdot]\!]$-shares of two n-sized vectors $\vec{\mathbf{x}}, \vec{\mathbf{y}}$, protocol Π_{dotp4} (Fig. 26) enables servers to compute $[\![\mathsf{z}]\!]$ with $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$. The protocol is essentially similar to n instances of multiplications of the form $\mathsf{z}_i = \mathsf{x}_i\mathsf{y}_i$ for $i \in [\mathsf{n}]$. But instead of communicating values corresponding to each of the n instances, servers locally sum up the shares and communicate a single value. This technique helps to obtain a communication cost independent of the size of the vectors.

During the preprocessing phase, similar to the multiplication protocol P_0, P_1, P_3 sample a random $[\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_1$. P_0, P_3 compute $\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}} = \sum_{i=1}^{\mathsf{n}} \alpha_{\mathsf{x}_i}\alpha_{\mathsf{y}_i}$ and jmp4-send $[\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_2 = \Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}} - [\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_1$ to P_2. P_1, P_2, P_3 sample a random ψ, and generate its $[\cdot]$-shares locally. Servers P_3, P_j for $j \in \{1, 2\}$ then compute $[\chi]_j = \sum_{i=1}^{\mathsf{n}} (\gamma_{\mathsf{x}_i}[\alpha_{\mathsf{y}_i}]_j + \gamma_{\mathsf{y}_i}[\alpha_{\mathsf{x}_i}]_j) + [\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_j - [\psi]_j$, and jmp4-send $[\chi]_j$ to P_0.

During the online phase, P_0, P_1 first compute $[\beta_{\mathsf{z}}^\star]_1$ where $\beta_{\mathsf{z}}^\star = \sum_{i=1}^{\mathsf{n}} \beta_{\mathsf{z}_i}^\star$ directly as $[\beta_{\mathsf{z}}^\star]_1 = -\sum_{i=1}^{\mathsf{n}}((\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})[\alpha_{\mathsf{y}_i}]_1 + (\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})[\alpha_{\mathsf{x}_i}]_1) + [\alpha_{\mathsf{z}}]_1 + [\chi]_1$. Following this, P_0, P_1 jmp4-send $[\beta_{\mathsf{z}}^\star]_1$ to P_2. P_0, P_2 proceed similarly to enable P_1 obtain $[\beta_{\mathsf{z}}^\star]_2$. Finally, P_1, P_2 compute $\beta_{\mathsf{z}}^\star = [\beta_{\mathsf{z}}^\star]_1 + [\beta_{\mathsf{z}}^\star]_2$ followed by computing $\beta_{\mathsf{z}} = \beta_{\mathsf{z}}^\star + \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i}) + \psi$. P_1, P_2 then jmp4-send $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0.

V. Applications and Benchmarking

In this section, we empirically show the practicality of our protocols for two widely used applications: Biometric Matching and PPML.

a) Benchmarking Environment: We use a 64-bit ring ($\mathbb{Z}_{2^{64}}$). The benchmarking is performed over a WAN that was instantiated using n1-standard-8 instances of Google Cloud[4], 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 2.3 GHz Intel Xeon E5 v3 (Haswell) processors supporting hyper-threading, with 8 vCPUs, and 30 GB of RAM Memory and with a bandwidth of 50 Mbps. The average round-trip time (rtt) was taken as the time for communicating 1 KB of data between a pair of parties, and the rtt values were

P_0-P_1	P_0-P_2	P_0-P_3	P_1-P_2	P_1-P_3	P_2-P_3
$151.40ms$	$59.95ms$	$275.02ms$	$92.94ms$	$173.93ms$	$219.37ms$

b) Software Details: We implement our protocols[5] using the publicly available ENCRYPTO library [48] in C++17. We obtained the code of BLAZE and FLASH from the respective authors and executed them in our environment. The collision-resistant hash function was instantiated using SHA-256. We have used multi-threading and our machines were capable of handling a total of 32 threads. Each experiment is run for 20 times and the average values are reported.

A. Biometric Matching

Biometric computation is central to many real-world tasks such as face recognition [49], [50] and fingerprint-matching [51], [52]. The objective is, given a database $\mathbf{D}$ of m biometric samples stored as vectors $(\vec{\mathbf{s}}_1, \ldots, \vec{\mathbf{s}}_m)$ each of size n, and a user with its own sample $\vec{\mathbf{u}}$, identify the "closest" sample to $\vec{\mathbf{u}}$ in $\mathbf{D}$. This task can be accomplished by considering various distance metrics, the most prominent of which is the Euclidean Distance (ED). In this work, we consider ED as the metric, and hence the problem boils down to identifying a sample vector in $\mathbf{D}$ which has the least ED for $\vec{\mathbf{u}}$. Note that, in our setting, each entry in the database $\mathbf{D}$ is $[\![\cdot]\!]$-shared among the servers. The client with an input query $\vec{\mathbf{u}}$ generates $[\![\cdot]\!]$- shares of the same along with the servers.

Let x_i denote the ith element in the vector $\vec{\mathbf{x}}$. As was introduced in [1], ED between two n length vectors $\vec{\mathbf{x}}$, $\vec{\mathbf{y}}$ is computed as $\mathbf{ED}_{\vec{\mathbf{x}}\vec{\mathbf{y}}} = \sum_{i=1}^{i=\mathsf{n}}(\mathsf{x}_i - \mathsf{y}_i)^2 = \vec{\mathbf{z}} \odot \vec{\mathbf{z}}$ where $\vec{\mathbf{z}} = ((\mathsf{x}_1 - \mathsf{y}_1), \ldots, (\mathsf{x}_{\mathsf{n}} - \mathsf{y}_{\mathsf{n}}))$. Hence, the servers first compute $[\![\cdot]\!]$-shares for vector $\vec{\mathbf{z}}$ locally, as $[\![\mathsf{z}_i]\!] = [\![\mathsf{x}_i]\!] - [\![\mathsf{y}_i]\!]$ for $i \in [\mathsf{n}]$, followed by an execution of Π_{dotp} on $[\![\vec{\mathbf{z}}]\!], [\![\vec{\mathbf{z}}]\!]$. For biometric computation, the servers create a distance vector $\mathbf{DV}$

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

[5]The link to our code is not provided respecting the double-blinded submission policy. The code will be made publicly available once the work sees formal acceptance.

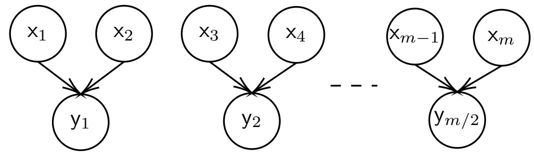

Fig. 13: Minimum Value - An example

by computing the ED between $\vec{\mathbf{u}}$ and every sample vector $\vec{\mathbf{s}}_i$ in $\mathbf{D}$, i.e $\mathbf{DV}_i = \mathbf{ED}_{\vec{\mathbf{u}}\vec{\mathbf{s}}_i}$ for $i \in [m]$. The next task now is to find the minimum among the m values in $\mathbf{DV}$.

Minimum among m values: Consider vector $\vec{\mathbf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_m)$ of size m, where each element is $[\![\cdot]\!]$-shared among the servers. We follow the standard tree based approach to compute the minimum element. This is as follows. First the elements of the vector are grouped into pairs, which are then securely compared to find the pairwise minimum. For instance, $[\![\cdot]\!]$-shares of $(\mathsf{x}_1, \mathsf{x}_2)$, $(\mathsf{x}_3, \mathsf{x}_4)$, ..., $(\mathsf{x}_{m-1}, \mathsf{x}_m)$ are compared to obtain $[\![\cdot]\!]$-shares of $\mathsf{y}_1, \ldots, \mathsf{y}_{m/2}$. Let $\vec{\mathbf{y}} = (\mathsf{y}_1, \mathsf{y}_2, \ldots, \mathsf{y}_{m/2})$. This process is recursively applied on $\vec{\mathbf{y}}$, until a single element is obtained. This requires $O(\log(m))$ rounds of recursion to obtain the minimum value in $\vec{\mathbf{x}}$. Note that the minimum of any two elements, say $\mathsf{x}_1, \mathsf{x}_2$ can be computed as $\mathsf{y}_1 = \mathsf{b} \cdot (\mathsf{x}_1 - \mathsf{x}_2) + \mathsf{x}_2$, where $\mathsf{b} = 0$ if $\mathsf{x}_1 > \mathsf{x}_2$, or 1, otherwise. This can be achieved using one invocation of bit extraction protocol Π_{bitext} on $(\mathsf{x}_1 - \mathsf{x}_2)$ to obtain $[\![\cdot]\!]^{\mathbf{B}}$-shares of b, followed by one execution of bit injection Π_{BitInj} on $\mathsf{b}^{\mathbf{B}}$ and $(\mathsf{x}_1 - \mathsf{x}_2)$.

Setting	Ref.	$m = 1024$			$m = 16384$		
		Pre.	Online		Pre.	Online	
		Com [KB]	R	Com [KB]	C [KB]	R	Com [KB]
3PC	BLAZE	1127.1	102	151.1	18036.0	142	2419.9
	SWIFT	1128.3	103	151.9	18037.8	143	2420.7
4PC	FLASH	223.9	71	239.8	3583.8	99	3839.8
	SWIFT	127.1	71	103.2	2035.9	99	1651.9

TABLE III: Minimum ED distance. The values are reported for biometric samples of size 40.

Table III presents the benchmarking for biometric matching over 3PC and 4PC setting. Following SecureML [1], we chose the size of the biometric sample n to be 40. As is evident from the Table III, in 3PC, we incur a minimal loss in performance over BLAZE but guarantee the security of GOD instead of fairness. For the case of 4PC, we observe $\approx 2\times$ improvement over the state-of-the-art protocol of FLASH [7] in terms of communication cost.

B. Privacy-preserving Machine Learning

We consider training and inference for Linear Regression and Logistic Regression and inference for Neural Networks (NN). As pointed out in BLAZE, NN training requires additional tools to allow mixed world computations, which we leave as future work. We refer readers to SecureML [1], ABY3 [4], and BLAZE [9] for a detailed description of the training and inference steps for the aforementioned ML algorithms. All our benchmarking is done over the publicly available MNIST [53] dataset that has $\mathsf{n} = 784$ features. For training, we used a batch size of $B = 128$.

In 3PC, we compare our results against the best-known framework BLAZE in this setting that provides fairness. Our results imply that we get GOD at no additional cost compared to BLAZE. For 4PC, we compare our results with two best-known works FLASH [7] (which is robust) and Trident [8] (which is fair). Our results halve the cost of FLASH and are on par with Trident.

1) Benchmarking Parameter: We use *throughput* (TP) as the benchmarking parameter following BLAZE and ABY3 [4] as it would help to analyse the effect of improved communication and round complexity in a single shot. Here, TP denotes the number of operations ("iterations" for the case of training and "queries" for the case of inference) that can be performed in unit time. We consider minute as the unit time since most of our protocols over WAN requires more than a second to complete. An *iteration* in ML training consists of a *forward propagation* phase followed by a *backward propagation* phase. In the former phase, servers compute the output from the inputs while in the latter, the model parameters are adjusted according to the difference in the computed output and the actual output. The inference can be viewed as one forward propagation of the algorithm alone.

2) Logistic Regression: In Logistic Regression, one iteration comprises updating the weight vector $\vec{\mathbf{w}}$ using the gradient descent algorithm (GD). It is updated according to the function given below: $\vec{\mathbf{w}} = \vec{\mathbf{w}} - \frac{\alpha}{B}\mathbf{X}_i^T \circ (\mathsf{sig}(\mathbf{X}_i \circ \vec{\mathbf{w}}) - \mathbf{Y}_i)$. where α and $\mathbf{X}_i$ denote the learning rate, and a subset of batch size B, randomly selected from the entire dataset in the ith iteration, respectively. The forward propagation comprises of computing the value $\mathbf{X}_i \circ \vec{\mathbf{w}}$ followed by an application of a sigmoid function on it. The weight vector is updated in the backward propagation, which internally requires the computation of a series of matrix multiplications, and can be achieved using a dot product. The update function can be computed using $[\![\cdot]\!]$ shares as: $[\![\vec{\mathbf{w}}]\!] = [\![\vec{\mathbf{w}}]\!] - \frac{\alpha}{B}[\![\mathbf{X}_j^T]\!] \circ (\mathsf{sig}([\![\mathbf{X}_j]\!] \circ [\![\vec{\mathbf{w}}]\!]) - [\![\mathbf{Y}_j]\!])$. We summarize our results in Table IV.

Setting	Ref.	Pre.	Online (TP in $\times 10^3$)		
		Com [KB]	Latency (s)	Com [KB]	TP
3PC Training	BLAZE	4757.11	1.17	50.23	2525.36
	SWIFT	4757.29	1.23	50.31	2393.38
3PC Inference	BLAZE	18.69	1.08	0.25	2728.65
	SWIFT	18.71	1.08	0.28	2727.38
4PC Training	FLASH	99.09	1.22	88.84	1158.65
	SWIFT	51.36	1.22	41.23	2407.64
4PC Inference	FLASH	0.39	1.05	0.41	2044.01
	SWIFT	0.21	1.05	0.18	2806.09

TABLE IV: Logistic Regression training and inference. TP is given in (#it/min) for training and (#queries/min) for inference.

We observe that the online TP, for the case of 3PC, is slightly lower compared to that of BLAZE, though the amortized online communication cost is the same for both. This is because the total number of rounds for both training and inference phase of Logistic Regression is slightly higher in our case due to the additional rounds introduced by the verification mechanism (aka *verify* phase which also needs broadcast). This gap becomes less evident for protocols with more number of rounds, as is demonstrated in the case of NN (presented next), where verification for several iterations is clubbed together, making the overhead for verification insignificant.

For the case of 4PC, our solution outperforms FLASH in terms of communication as well as throughput. For the case of logistic regression inference, the improvement over FLASH is not $2\times$ as claimed theoretically. This stems from the limited processing power in our benchmarking environment and can be addressed by increasing the processing capacity of the servers. Hence, to be fair, we limit the bandwidth to 34 Mbps and obtain a $2\times$ improvement in TP over FLASH. At 34 Mbps, the TP of FLASH turns out to be 1389.93×10^3 #queries/min. This highlights the efficiency improvements with reduced communication over lower bandwidths. We now compare with Trident [8]. Here, we observe a drop of 12.49% in TP for inference and a drop of 10.91% in TP for training. This is due to the extra rounds required for verification to achieve GOD. We point out that this drop becomes less significant for protocols involving more number of rounds, as will be evident from the comparisons for NN inference. Note, however, that the loss in TP is traded off with stronger security and the gain in saving the runtime of one server by $\approx 73\%$ owing to the presence of 2 active servers in our case, as opposed to 3 in Trident, thereby resulting in monetary gains in the cloud setting.

3) NN Inference: In this work, we consider a NN with two hidden layers, each consisting of 128 nodes each and an output layer with 10 nodes [4], [9]. Each of the layers is fully connected. Inference in NN requires several dot product calls followed by an application of the ReLU function. This process will be carried out for each layer in a sequential manner. Table V summarises our benchmarking results for NN inference.

Setting	Ref.	Pre.	Online (TP in $\times 10^3$)		
		Com [MB]	Latency (s)	Com [MB]	TP
3PC Inference	BLAZE	351.70	2.81	4.91	26.40
	SWIFT	351.70	2.91	4.91	26.40
4PC Inference	FLASH	7.79	2.71	7.79	14.21
	SWIFT	4.39	2.71	3.35	36.96

TABLE V: NN Inference. TP is given in (#queries/min).

As illustrated in Table V, the performance of our 3PC framework is comparable to BLAZE. In the 4PC setting, when compared to Trident [8], we observe a minimal loss of 0.36% in the online throughput. The drop surfaces due to the extra rounds involved in our verification. However, as pointed out earlier, the difference in TP closes in due to the large number of rounds required for computing NN inference, which results in amortizing the extra rounds required for verification. As mentioned earlier, this loss is traded-off with the stronger security guarantee and saving in the runtime of one server by $\approx 85\%$ owing to the presence of 2 active servers. Moreover, we outperform FLASH in every aspect. This establishes the practical relevance of our work.

VI. Conclusion

In this work, we presented an efficient framework for PPML that achieves the strongest security of GOD or robustness. Our 3PC protocol builds upon the recent work of BLAZE [9] and achieves almost similar performance albeit improving the security guarantee. For the case of 4PC, we outperform the best-known– (a) robust protocol of FLASH [7] by $2\times$ performance-wise and (b) fair protocol of Trident [8] by uplifting its security.

We leave the problem of extending our framework to support mixed-world conversions as well as to design protocols to support algorithms like Decision Trees, k-means Clustering etc. as open problem. The problem of making the communication cost of dot product entirely independent of the feature size is another challenging question.

References

[1] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *IEEE S&P*, 2017, pp. 19–38.

[2] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "EPIC: efficient private image classification (or: Learning from the masters)," in *CT-RSA*, 2019, pp. 473–492.

[3] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *AsiaCCS*, 2018, pp. 707–721.

[4] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM CCS*, 2018, pp. 35–52.

[5] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *PoPETs*, pp. 26–49, 2019.

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

[7] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: fast and robust framework for privacy-preserving machine learning," *PETS*, 2020. [Online]. Available: https://eprint.iacr.org/2019/1365

[8] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning," *NDSS*, 2020. [Online]. Available: https://arxiv.org/abs/1912.02631

[9] A. Patra and A. Suresh, "BLAZE: Blazing Fast Privacy-Preserving Machine Learning," *NDSS*, 2020. [Online]. Available: https://eprint.iacr.org/2020/042

[10] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO*, 2012, pp. 643–662.

[11] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits," in *ESORICS*, 2013, pp. 1–18.

[12] M. Keller, P. Scholl, and N. P. Smart, "An architecture for practical actively secure MPC with dishonest majority," in *ACM CCS*, 2013, pp. 549–560.

[13] M. Keller, E. Orsini, and P. Scholl, "MASCOT: faster malicious arithmetic secure computation with oblivious transfer," in *ACM CCS*, 2016, pp. 830–842.

[14] C. Baum, I. Damgård, T. Toft, and R. W. Zakarias, "Better preprocessing for secure multiparty computation," in *ACNS*, 2016, pp. 327–345.

[15] I. Damgård, C. Orlandi, and M. Simkin, "Yet another compiler for active security or: Efficient MPC over arbitrary rings," in *CRYPTO*, 2018, pp. 799–829.

[16] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority," in *CRYPTO*, 2018, pp. 769–798.

[17] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *EUROCRYPT*, 2018, pp. 158–189.

[18] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *ESORICS*, 2008, pp. 192–206.

[19] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz, "Efficient multi-party computation over rings," in *EUROCRYPT*, 2003, pp. 596–613.

[20] D. Demmler, T. Schneider, and M. Zohner, "ABY - A framework for efficient mixed-protocol secure two-party computation," in *NDSS*, 2015.

[21] 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," *IEEE S&P*, 2019.

[22] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *ACM CCS*, 2016, pp. 805–817.

[23] 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 *IEEE S&P*, 2017, pp. 843–862.

[24] R. Cleve, "Limits on the security of coin flips when half the processors are faulty (extended abstract)," in *ACM STOC*, 1986, pp. 364–369.

[25] P. Mohassel, M. Rosulek, and Y. Zhang, "Fast and secure three-party computation: The garbled circuit approach," in *ACM CCS*, 2015, pp. 591–602.

[26] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *EUROCRYPT*, 2017, pp. 225–255.

[27] M. Byali, A. Joseph, A. Patra, and D. Ravi, "Fast secure computation for small population over the internet," in *ACM CCS*, 2018, pp. 677–694.

[28] P. S. Nordholt and M. Veeningen, "Minimising communication in honest-majority MPC by batchwise multiplication verification," in *ACNS*, 2018, pp. 321–339.

[29] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-knowledge proofs on secret-shared data via fully linear pcps," in *CRYPTO*, 2019, pp. 67–97.

[30] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs," in *ACM CCS*, 2019, pp. 869–886.

[31] A. Patra and D. Ravi, "On the exact round complexity of secure three-party computation," in *CRYPTO*, 2018, pp. 425–458.

[32] M. Byali, C. Hazay, A. Patra, and S. Singla, "Fast actively secure five-party computation with security beyond abort," in *ACM CCS*, 2019, pp. 1573–1590.

[33] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority MPC for malicious adversaries," in *CRYPTO*, 2018, pp. 34–64.

[34] M. Abspoel, A. P. K. 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.

[35] H. Eerikson, M. Keller, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin, "Use your brain! arithmetic 3pc for any modulus with active security," Cryptology ePrint Archive, Report 2019/164, 2019, https://eprint.iacr.org/2019/164.

[36] S. D. Gordon, S. Ranellucci, and X. Wang, "Secure computation with low communication from cross-checking," in *ASIACRYPT*, 2018, pp. 59–85.

[37] Y. Lindell and B. Pinkas, "Privacy preserving data mining," *J. Cryptology*, pp. 177–206, 2002.

[38] W. Du and M. J. Atallah, "Privacy-preserving cooperative scientific computations," in *IEEE CSFW-14*, 2001, pp. 273–294.

[39] A. P. Sanil, A. F. Karr, X. Lin, and J. P. Reiter, "Privacy preserving regression modelling via distributed computation," in *ACM SIGKDD*, 2004, pp. 677–682.

[40] G. Jagannathan and R. N. Wright, "Privacy-preserving distributed k-means clustering over arbitrarily partitioned data," in *ACM SIGKDD*, 2005, pp. 593–599.

[41] P. Bunn and R. Ostrovsky, "Secure two-party k-means clustering," in *ACM CCS*, 2007, pp. 486–497.

[42] H. Yu, J. Vaidya, and X. Jiang, "Privacy-preserving SVM classification on vertically partitioned data," in *PAKDD*, 2006, pp. 647–656.

[43] J. Vaidya, H. Yu, and X. Jiang, "Privacy-preserving SVM classification," *Knowl. Inf. Syst.*, pp. 161–178, 2008.

[44] A. B. Slavkovic, Y. Nardi, and M. M. Tibbits, "Secure logistic regression of horizontally and vertically partitioned distributed databases," in *ICDM*, 2007, pp. 723–728.

[45] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "Spot-light: Lightweight private set intersection from sparse OT extension," in *CRYPTO*, 2019, pp. 401–431.

[46] R. Cohen, I. Haitner, E. Omri, and L. Rotem, "Characterization of secure multiparty computation without broadcast," *J. Cryptology*, pp. 587–609, 2018.

[47] M. C. Pease, R. E. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, pp. 228–234, 1980.

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

[49] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, "Privacy-preserving face recognition," in *PETS*, 2009, pp. 235–253.

[50] W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: tool for automating secure two-party computations," in *ACM CCS*, 2010, pp. 451–462.

[51] M. Blanton and P. Gasti, "Secure and efficient protocols for iris and fingerprint identification," in *ESORICS*, 2011, pp. 190–209.

[52] W. Henecka and T. Schneider, "Faster secure two-party computation with less memory," in *ASIA CCS*, 2013, pp. 437–446.

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

[54] G. Asharov and Y. Lindell, "A full proof of the BGW protocol for perfectly secure multiparty computation," *J. Cryptology*, pp. 58–151, 2017.

APPENDIX A
PRELIMINARIES

A. 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 set of keys established between the servers for 3PC is as follows:

- One key shared between every pair– k_{01}, k_{02}, k_{12} for the parties $(P_0, P_1), (P_0, P_2)$and(P_1, P_2), respectively.
- One shared key known to all the servers– $k_{\mathcal{P}}$.

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 $\mathcal{F}_{\mathsf{setup}}$

$\mathcal{F}_{\mathsf{setup}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys k_{ij} for $i, j \in \{0,1,2\}$ and $k_{\mathcal{P}}$. Let y_s denote the keys corresponding to server P_s. Then

- $\mathsf{y}_s = (k_{01}, k_{02}$ and $k_{\mathcal{P}})$ when $P_s = P_0$.
- $\mathsf{y}_s = (k_{01}, k_{12}$ and $k_{\mathcal{P}})$ when $P_s = P_1$.
- $\mathsf{y}_s = (k_{02}, k_{12}$ and $k_{\mathcal{P}})$ when $P_s = P_2$.

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

Fig. 14: 3PC: Ideal functionality for shared-key setup

The key setup is modelled via a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 14) that can be realised using any secure MPC protocol. Analogously, the key setup functionality for 4PC is given in Fig. 15.

Functionality $\mathcal{F}_{\text{setup4}}$

$\mathcal{F}_{\text{setup4}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\text{setup4}}$ picks random keys k_{ij} and k_{ijk} for $i, j, k \in \{0, 1, 2\}$ 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 $P_s \in \mathcal{P}$.

Fig. 15: 4PC: Ideal functionality for shared-key setup

B. Collision Resistant Hash Function

Consider a hash function family $\mathsf{H} = \mathcal{K} \times \mathcal{L} \rightarrow \mathcal{Y}$. The hash function H is said to be collision resistant if, for all probabilistic polynomial-time adversaries $\mathcal{A}$, given the description of H_k where $k \in_R \mathcal{K}$, there exists a negligible function negl() such that $\Pr[(x_1, x_2) \leftarrow \mathcal{A}(k) : (x_1 \neq x_2) \wedge \mathsf{H}_k(x_1) = \mathsf{H}_k(x_2)] \leq \mathsf{negl}(\kappa)$, where $m = \mathsf{poly}(\kappa)$ and $x_1, x_2 \in_R \{0, 1\}^m$.

C. Commitment Scheme

Let $\mathsf{Com}(x)$ denote the commitment of a value x. The commitment scheme $\mathsf{Com}(x)$ possesses two properties; *hiding* and *binding*. The former ensures privacy of the value v given just its commitment $\mathsf{Com}(\mathsf{v})$, while the latter prevents a corrupt party from opening the commitment to a different value $x' \neq x$. The practical realization of a commitment scheme is via a hash function $\mathcal{H}()$ given below, whose security can be proved in the random-oracle model (ROM)– for $(c, o) = (\mathcal{H}(x||r), x||r) = \mathsf{Com}(x; r)$.

APPENDIX B
3PC PROTOCOLS

In this section, we provide a detailed communication cost analysis for our protocols in the 3PC setting. Also detailed information regarding some of the protocols are provided.

A. Joint Message Passing

Lemma B.1 (Communication)**.** *Protocol Π_{jmp} (Fig. 2) requires 1 round and an amortized communication of ℓ bits overall.*

Proof: Server P_i sends value v to P_k while P_j sends hash of the same to P_k. This accounts for one round and communication of ℓ bits. P_k then sends back its inconsistency bit to P_i, P_j, who then exchange it; this takes another two rounds. This is followed by servers broadcasting hashes on their values and selecting a TTP based on it, which takes one more round. All except the first round can be combined for several instances of Π_{jmp} protocol and hence the cost gets amortized. ■

B. Sharing Protocol

Lemma B.2 (Communication)**.** *Protocol Π_{sh} (Fig. 3) is non-interactive in the preprocessing phase and requires 2 rounds and an amortized communication of 2ℓ bits in the online phase.*

Proof: During the preprocessing phase, servers non-interactively sample the $[\cdot]$-shares of α_{v} and γ_{v} values using the shared key setup. In the online phase, when $P_i = P_0$, it computes β_{v} and sends it to P_1, resulting in one round and ℓ bits communicated. They then jmp-send β_{v} to P_2, which requires additional one round in an amortized sense, and ℓ bits to be communicated. For the case when $P_i = P_1$, it sends β_{v} to P_2, resulting in one round and a communication of ℓ bits. Then, P_1, P_2 jmp-send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0. This again requires an additional one round and ℓ bits. The analysis is similar in the case of $P_i = P_2$. ■

C. Joint Sharing Protocol

The formal details for Π_{jsh} protocol appears in Fig. 16.

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

Preprocessing:

– If $(P_i, P_j) = (P_1, P_0)$: Servers execute the preprocessing of $\Pi_{\mathsf{sh}}(P_1, \mathsf{v})$ and then locally set $\gamma_{\mathsf{v}} = 0$.
– If $(P_i, P_j) = (P_2, P_0)$: Similar to the case above.
– If $(P_i, P_j) = (P_1, P_2)$: P_1, P_2 together sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$. Servers locally set $[\alpha_{\mathsf{v}}]_1 = [\alpha_{\mathsf{v}}]_2 = 0$.

Online:

– If $(P_i, P_j) = (P_1, P_0)$: P_0, P_1 compute $\beta_{\mathsf{v}} = \mathsf{v} + [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$. P_0, P_1 jmp-send β_{v} to P_2.
– If $(P_i, P_j) = (P_2, P_0)$: Similar to the case above.
– If $(P_i, P_j) = (P_1, P_2)$: P_1, P_2 locally set $\beta_{\mathsf{v}} = \mathsf{v}$. P_1, P_2 jmp-send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0.

Fig. 16: 3PC: $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ jointly by P_i, P_j

When the value v is available to both P_i, P_j in the preprocessing phase, protocol Π_{jsh} can be made non-interactive in the following way: $\mathcal{P}$ sample a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and locally set their share according to Table VI.

	(P_1, P_2)	(P_1, P_0)	(P_2, P_0)
	$[\alpha_{\mathsf{v}}]_1 = 0,\ [\alpha_{\mathsf{v}}]_2 = 0$ $\beta_{\mathsf{v}} = \mathsf{v},\ \gamma_{\mathsf{v}} = \mathsf{r} - \mathsf{v}$	$[\alpha_{\mathsf{v}}]_1 = -\mathsf{v},\ [\alpha_{\mathsf{v}}]_2 = 0$ $\beta_{\mathsf{v}} = 0,\ \gamma_{\mathsf{v}} = \mathsf{r}$	$[\alpha_{\mathsf{v}}]_1 = 0,\ [\alpha_{\mathsf{v}}]_2 = -\mathsf{v}$ $\beta_{\mathsf{v}} = 0,\ \gamma_{\mathsf{v}} = \mathsf{r}$
P_0	$(0,\ 0,\ \mathsf{r})$	$(-\mathsf{v},\ 0,\ \mathsf{r})$	$(0,\ -\mathsf{v},\ \mathsf{r})$
P_1	$(0,\ \mathsf{v},\ \mathsf{r} - \mathsf{v})$	$(-\mathsf{v},\ 0,\ \mathsf{r})$	$(0,\ 0,\ \mathsf{r})$
P_2	$(0,\ \mathsf{v},\ \mathsf{r} - \mathsf{v})$	$(0,\ 0,\ \mathsf{r})$	$(0,\ -\mathsf{v},\ \mathsf{r})$

TABLE VI: The columns depict the three distinct possibility of input contributing pairs. The first row shows the assignment to various components of the sharing. The last row, along with three sub-rows, specify the shares held by the three servers.

Lemma B.3 (Communication)**.** *Protocol Π_{jsh} (Fig. 16) is non-interactive in the preprocessing phase and requires 1 round and an amortized communication of ℓ bits in the online phase.*

Proof: In this protocol, servers execute Π_{jmp} protocol once. Hence the overall cost follows from that of an instance of the Π_{jmp} protocol (Lemma B.1). ■

D. Multiplication Protocol

Lemma B.4 (Communication)**.** *Protocol Π_{mult} (Fig. 4) requires an amortized cost of 3ℓ bits in the preprocessing phase, and 1 round and amortized cost of 3ℓ bits in the online phase.*

Proof: In the preprocessing phase, generation of α_{z} and γ_{z} are non-interactive. This is followed by one execution of Π_{mulPre}, which requires an amortized communication cost of 3ℓ bits. During the online phase, P_0, P_1 jmp-send $[\beta^\star_{\mathsf{z}}]_1$ to P_2, while P_0, P_2 jmp-send $[\beta^\star_{\mathsf{z}}]_2$ to P_1. This requires one round and a communication of 2ℓ bits. Following this, P_1, P_2 jmp-send $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0, which requires one round and a communication of ℓ bits. However, jmp-send of $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ can be delayed till the end of the protocol, and will require only one round for the entire circuit and can be amortized. ■

The ideal functionality for Π_{mulPre} appears in Fig. 17.

Functionality $\mathcal{F}_{\mathsf{MulPre}}$

$\mathcal{F}_{\mathsf{MulPre}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{MulPre}}$ receives $\langle\cdot\rangle$-shares of d, e from the servers where P_s, for $s \in \{0, 1, 2\}$, holds $\langle \mathsf{d} \rangle_s = (\mathsf{d}_s, \mathsf{d}_{(s+1)\%3})$ and $\langle \mathsf{e} \rangle_s = (\mathsf{e}_s, \mathsf{e}_{(s+1)\%3})$ such that $\mathsf{d} = \mathsf{d}_0 + \mathsf{d}_1 + \mathsf{d}_2$ and $\mathsf{e} = \mathsf{e}_0 + \mathsf{e}_1 + \mathsf{e}_2$. Let P_i denotes the server corrupted by $\mathcal{S}$. $\mathcal{F}_{\mathsf{MulPre}}$ receives $\langle \mathsf{f} \rangle_i = (\mathsf{f}_i, \mathsf{f}_{(i+1)\%3})$ from $\mathcal{S}$ where $\mathsf{f} = \mathsf{de}$. $\mathcal{F}_{\mathsf{MulPre}}$ proceeds as follows:

- Reconstructs d, e using the shares received from honest servers and compute $\mathsf{f} = \mathsf{de}$.
- Compute $\mathsf{f}_{(i+2)\%3} = \mathsf{f} - \mathsf{f}_i - \mathsf{f}_{(i+1)\%3}$ and set the output shares as $\langle \mathsf{f} \rangle_0 = (\mathsf{f}_0, \mathsf{f}_1), \langle \mathsf{f} \rangle_1 = (\mathsf{f}_1, \mathsf{f}_2), \langle \mathsf{f} \rangle_2 = (\mathsf{f}_2, \mathsf{f}_0)$.
- Send $(\mathsf{Output}, \langle \mathsf{f} \rangle_s)$ to server $P_s \in \mathcal{P}$.

Fig. 17: 3PC: Ideal functionality for Π_{mulPre} protocol

E. Reconstruction Protocol

Lemma B.5 (Communication)**.** *Protocol Π_{rec} (Fig. 5) requires 1 round and a communication of 6ℓ bits in the online phase.*

Proof: The preprocessing phase consists of communication of commitment values using the Π_{jmp} protocol. The hash-based commitment scheme allows generation of a single commitment for several values and hence the cost gets amortised away for multiple instances. During the online phase, each server receives an opening for the commitment from other two servers, which requires one round and an overall communication of 6ℓ bits. ■

F. Special protocols

Here we provide details regarding the special protocols - i) Bit Extraction, ii) Bit2A, and iii) Bit Injection.

1) Bit Extraction protocol: Protocol Π_{bitext} allows servers to compute the *boolean* sharing of the most significant bit (msb) of a value v given its arithmetic sharing $[\![\mathsf{v}]\!]$. To compute the msb, we use the optimized 2-input Parallel Prefix Adder (PPA) boolean circuit proposed by ABY3 [4]. The PPA circuit consists of $2\ell - 2$ AND gates and has a multiplicative depth of $\log \ell$. Let $\mathsf{v}_0 = \beta_{\mathsf{v}}, \mathsf{v}_1 = -[\alpha_{\mathsf{v}}]_1$ and $\mathsf{v}_2 = -[\alpha_{\mathsf{v}}]_2$.

	P_0	P_1	P_2
$[\![\mathsf{v}_0[i]]\!]^{\mathbf{B}}$	$(0, 0, 0)$	$(0, \mathsf{v}_0[i], \mathsf{v}_0[i])$	$(0, \mathsf{v}_0[i], \mathsf{v}_0[i])$
$[\![\mathsf{v}_1[i]]\!]^{\mathbf{B}}$	$(\mathsf{v}_1[i], 0, 0)$	$(\mathsf{v}_1[i], 0, 0)$	$(0, 0, 0)$
$[\![\mathsf{v}_2[i]]\!]^{\mathbf{B}}$	$(0, \mathsf{v}_2[i], 0)$	$(0, 0, 0)$	$(0, \mathsf{v}_2[i], 0)$

TABLE VII: The $[\![\cdot]\!]^{\mathbf{B}}$-sharing corresponding to i^{th} bit of $\mathsf{v}_0 = \beta_{\mathsf{v}}, \mathsf{v}_1 = -[\alpha_{\mathsf{v}}]_1$ and $\mathsf{v}_2 = -[\alpha_{\mathsf{v}}]_2$. Here $i \in \{0, \ldots, \ell - 1\}$.

Then $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$. Servers first locally compute the boolean shares corresponding to each bit of the values $\mathsf{v}_0, \mathsf{v}_1$ and v_2 according to Table VII. It has been shown in ABY3 that $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$ can also be expressed as $\mathsf{v} = 2c + s$ where $\mathsf{FA}(\mathsf{v}_0[i], \mathsf{v}_1[i], \mathsf{v}_2[i]) \rightarrow (c[i], s[i])$ for $i \in \{0, \ldots, \ell - 1\}$. Here FA denotes a Full Adder circuit while s and c denote the sum and carry bits respectively. To summarize, servers execute ℓ instances of FA in parallel to compute $[\![c]\!]^{\mathbf{B}}$ and $[\![s]\!]^{\mathbf{B}}$. The FA's are executed independently and require one round of communication. The final result is then computed as $\mathsf{msb}(2[\![c]\!]^{\mathbf{B}} + [\![s]\!]^{\mathbf{B}})$ using the optimized PPA circuit.

Lemma B.6 (Communication)**.** *Protocol Π_{bitext} requires a communication cost of $9\ell - 6$ bits in the preprocessing phase and require $\log \ell + 1$ rounds and an amortized communication of $9\ell - 6$ bits in the online phase.*

Proof: In Π_{bitext}, first round comprises of ℓ Full Adder (FA) circuits executing in parallel, each comprising of single AND gate. This is followed by the execution of the optimized PPA circuit of ABY3 [4], which comprises of $2\ell - 2$ AND gates and has a multiplicative depth of $\log \ell$. Hence the communication cost follows from the multiplication for $3\ell - 2$ AND gates. ■

2) Bit2A Conversion protocol: Given the boolean sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, protocol Π_{bit2A} (Fig. 18) allows servers to compute the arithmetic sharing $[\![\mathsf{b}^{\mathsf{R}}]\!]$. Here b^{R} denotes the equivalent value of b over ring $\mathbb{Z}_{2^\ell}$ (see Notation II.2). As pointed out in BLAZE, $\mathsf{b}^{\mathsf{R}} = (\beta_{\mathsf{b}} \oplus \alpha_{\mathsf{b}})^{\mathsf{R}} = \beta_{\mathsf{b}}^{\mathsf{R}} + \alpha_{\mathsf{b}}^{\mathsf{R}} - 2\beta_{\mathsf{b}}^{\mathsf{R}}\alpha_{\mathsf{b}}^{\mathsf{R}}$. Also $\alpha_{\mathsf{b}}^{\mathsf{R}} = ([\alpha_{\mathsf{b}}]_1 \oplus [\alpha_{\mathsf{b}}]_2)^{\mathsf{R}} = [\alpha_{\mathsf{b}}]_1^{\mathsf{R}} + [\alpha_{\mathsf{b}}]_2^{\mathsf{R}} - 2\,[\alpha_{\mathsf{b}}]_1^{\mathsf{R}}\,[\alpha_{\mathsf{b}}]_2^{\mathsf{R}}$. During the preprocessing phase, P_0, P_j for $j \in \{1, 2\}$ execute Π_{jsh} on $[\alpha_{\mathsf{b}}]_j^{\mathsf{R}}$ to generate $[\![[\alpha_{\mathsf{b}}]_j^{\mathsf{R}}]\!]$. Servers then execute Π_{mult} on $[\![[\alpha_{\mathsf{b}}]_1^{\mathsf{R}}]\!]$ and $[\![[\alpha_{\mathsf{b}}]_2^{\mathsf{R}}]\!]$ to generate $[\![[\alpha_{\mathsf{b}}]_1^{\mathsf{R}}\,[\alpha_{\mathsf{b}}]_2^{\mathsf{R}}]\!]$ followed by locally computing $[\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$. During the online phase, P_1, P_2 execute Π_{jsh} on $\beta_{\mathsf{b}}^{\mathsf{R}}$ to jointly generate $[\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!]$. Servers then execute Π_{mult} protocol on $[\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!]$ and $[\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$ to compute $[\![\beta_{\mathsf{b}}^{\mathsf{R}}\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$ followed by locally computing b^{R}. The formal details for Π_{bit2A} protocol appears in Fig. 18.

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

Preprocessing:

- P_0, P_j for $j \in \{1, 2\}$ execute Π_{jsh} on $[\alpha_{\mathsf{b}}]_j^{\mathsf{R}}$ to generate $[\![[\alpha_{\mathsf{b}}]_j^{\mathsf{R}}]\!]$.
- Servers execute $\Pi_{\mathsf{mult}}(\mathcal{P}, [\alpha_{\mathsf{b}}]_1^{\mathsf{R}}, [\alpha_{\mathsf{b}}]_2^{\mathsf{R}})$ to generate $[\![\mathsf{u}]\!]$ where $\mathsf{u} = [\alpha_{\mathsf{b}}]_1^{\mathsf{R}}\,[\alpha_{\mathsf{b}}]_2^{\mathsf{R}}$, followed by locally computing $[\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!] = [\![[\alpha_{\mathsf{b}}]_1^{\mathsf{R}}]\!] + [\![[\alpha_{\mathsf{b}}]_2^{\mathsf{R}}]\!] - 2[\![\mathsf{u}]\!]$.
- Servers in $\mathcal{P}$ execute the preprocessing phase of $\Pi_{\mathsf{mult}}(\mathcal{P}, \beta_{\mathsf{b}}^{\mathsf{R}}, \alpha_{\mathsf{b}}^{\mathsf{R}})$ where $\mathsf{v} = \beta_{\mathsf{b}}^{\mathsf{R}}\alpha_{\mathsf{b}}^{\mathsf{R}}$.

Online:

- P_1, P_2 execute $\Pi_{\mathsf{jsh}}(P_1, P_2, \beta_{\mathsf{b}}^{\mathsf{R}})$ to generate $[\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!]$.
- Servers execute online phase of $\Pi_{\mathsf{mult}}(\mathcal{P}, \beta_{\mathsf{b}}^{\mathsf{R}}, \alpha_{\mathsf{b}}^{\mathsf{R}})$ to generate $[\![\mathsf{v}]\!]$ where $\mathsf{v} = \beta_{\mathsf{b}}^{\mathsf{R}}\alpha_{\mathsf{b}}^{\mathsf{R}}$, followed by locally computing $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!] + [\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!] - 2[\![\mathsf{v}]\!]$.

Fig. 18: 3PC: Bit2A Protocol

Lemma B.7 (Communication)**.** *Protocol Π_{bit2A} (Fig. 18) requires an amortized communication cost of 9ℓ bits in the preprocessing phase and requires 1 round and an amortized communication of 4ℓ bits in the online phase.*

Proof: In the preprocessing phase, servers run two instances of Π_{jsh}, which can be done non-interactively (ref. § B-C). This is followed by an execution of entire multiplication protocol, which requires 6ℓ bits to be communicated (Lemma B.4). Parallelly, the servers execute the preprocessing phase of Π_{mult}, resulting in an additional 3ℓ bits of communication (Lemma B.4). During the online phase, P_1, P_2 execute Π_{jsh} once, which requires one round and ℓ bits to be communicated. In Π_{jsh}, the communication towards P_0 can be deferred till the end, thereby requiring a single round for multiple instances. This is followed by an execution of the online phase of Π_{mult}, which requires one round and a communication of 3ℓ bits. ■

3) Bit Injection protocol: Given the binary sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, and the arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, protocol Π_{BitInj} computes $[\![\cdot]\!]$-sharing of bv. Towards this, servers first execute Π_{bit2A} on $[\![\mathsf{b}]\!]^{\mathbf{B}}$ to generate $[\![\mathsf{b}]\!]$. This is followed by servers computing $[\![\mathsf{bv}]\!]$ by executing Π_{mult} protocol on $[\![\mathsf{b}]\!]$ and $[\![\mathsf{v}]\!]$.

Lemma B.8 (Communication)**.** *Protocol Π_{BitInj} requires an amortized communication cost of 12ℓ bits in the preprocessing phase and requires 2 rounds and an amortized communication of 7ℓ bits in the online phase.*

Proof: Protocol Π_{BitInj} is essentially an execution of Π_{bit2A} (Lemma B.6) followed by one invocation of Π_{mult} (Lemma B.4) and the costs follow. ■

G. Dot Product Protocol

The ideal world functionality for realizing Π_{dotpPre} is presented in Fig. 19.

Functionality $\mathcal{F}_{\mathsf{DotPPre}}$

$\mathcal{F}_{\mathsf{DotPPre}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{DotPPre}}$ receives $\langle\cdot\rangle$-shares of vectors $\vec{\mathbf{d}} = (\mathsf{d}_1, \ldots, \mathsf{d}_\mathsf{n}), \vec{\mathbf{e}} = (\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{n})$ from the servers. Server P_s, for $s \in \{0,1,2\}$, holds $\langle \mathsf{d}_j \rangle_s = ((\mathsf{d}_j)_s, (\mathsf{d}_j)_{(s+1)\%3})$ and $\langle \mathsf{e}_j \rangle_s = ((\mathsf{e}_j)_s, (\mathsf{e}_j)_{(s+1)\%3})$ such that $\mathsf{d}_j = (\mathsf{d}_j)_0 + (\mathsf{d}_j)_1 + (\mathsf{d}_j)_2$ and $\mathsf{e}_j = (\mathsf{e}_j)_0 + (\mathsf{e}_j)_1 + (\mathsf{e}_j)_2$ where $j \in [\mathsf{n}]$. Let P_i denotes the server corrupted by $\mathcal{S}$. $\mathcal{F}_{\mathsf{MulPre}}$ receives $\langle \mathsf{f} \rangle_i = (\mathsf{f}_i, \mathsf{f}_{(i+1)\%3})$ from $\mathcal{S}$ where $\mathsf{f} = \vec{\mathbf{d}} \odot \vec{\mathbf{e}}$. $\mathcal{F}_{\mathsf{DotPPre}}$ proceeds as follows:

- Reconstructs $\mathsf{d}_j, \mathsf{e}_j$, for $j \in [\mathsf{n}]$, using the shares received from honest servers and compute $\mathsf{f} = \sum_{j=1}^{n} \mathsf{d}_j \mathsf{e}_j$.
- Compute $\mathsf{f}_{(i+2)\%3} = \mathsf{f} - \mathsf{f}_i - \mathsf{f}_{(i+1)\%3}$ and set the output shares as $\langle \mathsf{f} \rangle_0 = (\mathsf{f}_0, \mathsf{f}_1), \langle \mathsf{f} \rangle_1 = (\mathsf{f}_1, \mathsf{f}_2), \langle \mathsf{f} \rangle_2 = (\mathsf{f}_2, \mathsf{f}_0)$.
- Send $(\mathsf{Output}, \langle \mathsf{f} \rangle_s)$ to server $P_s \in \mathcal{P}$.

Fig. 19: 3PC: Ideal functionality for Π_{dotpPre} protocol

Lemma B.9 (Communication)**.** *Protocol Π_{dotp} (Fig. 7) requires an amortized communication of $3\mathsf{n}\ell$ bits in the preprocessing phase and requires 1 round and an amortized communication of 3ℓ bits in the online phase, where n denotes the size of the underlying vectors.*

Proof: During the preprocessing phase, servers execute the preprocessing phase of Π_{mult} corresponding to each of the n multiplications in parallel, resulting in a communication of $3\mathsf{n}\ell$ bits (Lemma B.4). The online phase follows similarly to that of Π_{mult}, the only difference being that servers combine their shares corresponding to all the n multiplications into one and then exchange. This requires one round and an amortized communication of 3ℓ bits. ■

H. Truncation

Lemma B.10 (Communication)**.** *Protocol Π_{trgen} (Fig. 8) requires 2 rounds and an amortized communication of 2ℓ bits.*

Proof: In Π_{trgen}, the additive shares of r are sampled non-interactively. P_0 then executes Π_{sh} protocol on r^d, which requires two rounds and a communication of 2ℓ bits (Lemma B.2). P_0, P_1 then jmp-send the hash of u, followed by a broadcast from P_2. Note that the cost of broadcast, and Π_{jmp} (as it involves sending a hash), gets amortized over multiple instances. ■

I. Dot Product with Truncation

The formal details for Π_{dotpt} protocol appears in Fig. 20.

Protocol $\Pi_{\mathsf{dotpt}}(\mathcal{P}, \{[\![\mathsf{x}_i]\!], [\![\mathsf{y}_i]\!]\}_{i \in [\mathsf{n}]})$

<u>**Preprocessing:**</u>

- Servers execute the preprocessing of $\Pi_{\mathsf{dotp}}(\mathcal{P}, \{[\![\mathsf{x}_i]\!], [\![\mathsf{y}_i]\!]\}_{i \in [\mathsf{n}]})$.
- In parallel, servers execute $\Pi_{\mathsf{trgen}}(\mathcal{P})$ to generate the truncation pair $([\mathsf{r}], [\![\mathsf{r}^d]\!])$. Also, P_0 obtains both the values $[\mathsf{r}]_1$ and $[\mathsf{r}]_2$.

<u>**Online:**</u>

- P_0, P_j, for $j \in \{1,2\}$, compute $[\Psi]_j = -\sum_{i=1}^{\mathsf{n}}((\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})[\alpha_{\mathsf{y}_i}]_j + (\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})[\alpha_{\mathsf{x}_i}]_j) - [\mathsf{r}]_j$ and set $[(\mathsf{z}-\mathsf{r})^\star]_j = [\Psi]_j + [\chi]_j$.
- P_1, P_0 jmp-send $[(\mathsf{z}-\mathsf{r})^\star]_1$ to P_2 and P_2, P_0 jmp-send $[(\mathsf{z}-\mathsf{r})^\star]_2$ to P_1.
- P_1, P_2 locally compute $(\mathsf{z}-\mathsf{r})^\star = [(\mathsf{z}-\mathsf{r})^\star]_1 + [(\mathsf{z}-\mathsf{r})^\star]_2$ and set $(\mathsf{z}-\mathsf{r}) = (\mathsf{z}-\mathsf{r})^\star + \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i}) + \psi$.
- P_1, P_2 locally truncate $(\mathsf{z}-\mathsf{r})$ to obtain $(\mathsf{z}-\mathsf{r})^d$ and execute $\Pi_{\mathsf{jsh}}(P_1, P_2, (\mathsf{z}-\mathsf{r})^d)$ to generate $[\![(\mathsf{z}-\mathsf{r})^d]\!]$.
- Servers locally compute $[\![\mathsf{z}]\!] = [\![(\mathsf{z}-\mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$.

Fig. 20: 3PC: Dot Product Protocol with Truncation

Lemma B.11 (Communication)**.** *Protocol Π_{dotpt} (Fig. 20) requires an amortized communication of $3\mathsf{n}\ell + 2\ell$ bits in the preprocessing phase and requires 1 round and an amortized communication of 3ℓ bits in the online phase.*

Proof: During the preprocessing phase, servers execute the preprocessing phase of Π_{dotp}, resulting in a communication of $3\mathsf{n}\ell$ bits (Lemma B.9). In parallel, servers execute one instance of Π_{trgen} protocol resulting in an additional communication of 2ℓ bits (Lemma B.10).

The online phase follows from that of Π_{dotp} protocol except that, now, P_1, P_2 compute additive shares of $\mathsf{z} - \mathsf{r}$, where $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$, which is achieved using two executions of Π_{jmp} in parallel. This requires one round and an amortized communication cost of 2ℓ bits. P_1, P_2 then jointly share the truncated value of $\mathsf{z}-\mathsf{r}$ with P_0, which requires one round and ℓ bits. However, this step can be deferred till the end for multiple dot product with truncation instances, which amortizes the cost. ■

J. Activation Functions

Lemma B.12 (Communication)**.** *Protocol* relu *requires an amortized communication of* $21\ell - 6$ *bits in the preprocessing phase and requires* $\log \ell + 4$ *rounds and an amortized communication of* $16\ell - 6$ *bits in the online phase.*

Proof: One instance of relu protocol comprises of execution of one instance of Π_{bitext}, followed by Π_{BitInj}. The cost, therefore, follows from Lemma B.6, and Lemma B-F3. ∎

The formal details of the MPC-friendly variant of the Sigmoid function [1], [4], [6] is given below:

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

Lemma B.13 (Communication)**.** *Protocol* sig *requires an amortized communication of* $39\ell - 9$ *bits in the preprocessing phase and requires* $\log \ell + 4$ *rounds and an amortized communication of* $29\ell - 9$ *bits in the online phase.*

Proof: An instance of sig protocol involves the execution of the following protocols in order– i) two parallel instances of Π_{bitext} protocol, ii) once instance of Π_{mult} protocol over boolean value, and iii) one instance of Π_{BitInj} and Π_{bit2A} in parallel. The cost follows from Lemma B.6, Lemma B.7 and Lemma B.8. ∎

Appendix C
4PC Protocols

In this section, we give the formal details for the 4PC protocols along with communication cost analysis.

A. 4PC Joint Message Passing Primitive

The ideal functionality for jmp4 primitive appears in Fig. 21.

Functionality $\mathcal{F}_{\mathsf{jmp4}}$

$\mathcal{F}_{\mathsf{jmp4}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$.

Step 1: $\mathcal{F}_{\mathsf{jmp}}$ 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 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, 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 $s \in \{0, 1, 2, 3\}$.

Fig. 21: 4PC: Ideal functionality for jmp4 primitive

Lemma C.1 (Communication)**.** *Protocol* Π_{jmp4} *(Fig. 9) requires* 1 *round and an amortized communication of* ℓ *bits in the online phase.*

Proof: Server P_i sends the value v to P_k while P_j sends hash of the same to P_k. This accounts for one round of communication. Values sent by P_j for several instances can be concatenated and hashed to obtain a single value. Hence the cost of sending the hash gets amortized over multiple instances. Similarly, the two round exchange of inconsistency bits, along with the two round signalling to the TTP can be combined for multiple instances, thereby amortizing this cost. Thus, the amortized cost of this protocol is ℓ bits. ∎

B. Sharing Protocol

Lemma C.2 (Communication)**.** *In the online phase,* Π_{sh4} *(Fig. 10) requires* 2 *rounds and an amortized communication of* 2ℓ *bits when* P_0, P_1, P_2 *share a value, whereas it requires an amortized communication of* 3ℓ *bits when* P_3 *shares a value.*

Proof: The proof for P_0, P_1, P_2 sharing a value follows from B.2. For the case when P_3 wants to share a value v, it first sends $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0 which requires one round and ℓ bits of communication. This is followed by 2 parallel calls to Π_{jmp4} which together require one round and an amortized communication of 2ℓ bits. ∎

C. Joint Sharing Protocol

Protocol Π_{jsh4} enables a pair of (unordered) servers (P_i, P_j) to jointly generate a $[\![\cdot]\!]$-sharing of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ known to both of them. In case of an inconsistency, the server outside the computation serves as a TTP. The protocol is described in Fig. 22.

Protocol $\Pi_{\mathsf{jsh4}}(P_i, P_j, \mathsf{v})$

Preprocessing:

- If $(P_i, P_j) = (P_1, P_2)$: P_1, P_2, P_3 sample $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$. Servers locally set $[\alpha_{\mathsf{v}}]_1 = [\alpha_{\mathsf{v}}]_2 = 0$.
- If $(P_i, P_j) = (P_s, P_0)$, for $s \in \{1, 2\}$: Servers execute the preprocessing of $\Pi_{\mathsf{sh4}}(P_s, \mathsf{v})$. Servers locally set $\gamma_{\mathsf{v}} = 0$.
- If $(P_i, P_j) = (P_s, P_3)$, for $s \in \{0, 1, 2\}$: Servers execute the preprocessing of $\Pi_{\mathsf{sh4}}(P_s, \mathsf{v})$.

Online:

- If $(P_i, P_j) = (P_1, P_2)$: P_1, P_2 set $\beta_{\mathsf{v}} = \mathsf{v}$ and jmp4-send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0.
- If $(P_i, P_j) = (P_s, P_0)$, for $s \in \{1, 2, 3\}$: P_s, P_0 compute $\beta_{\mathsf{v}} = \mathsf{v} + [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$ and jmp4-send β_{v} to P_k, where $(k \in \{1, 2\}) \wedge (k \neq s)$.
- If $(P_i, P_j) = (P_s, P_3)$, for $s \in \{1, 2\}$: P_3, P_s compute β_{v} and $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$. P_s, P_3 jmp4-send β_{v} to P_k, where $(k \in \{1, 2\}) \wedge (k \neq s)$. In parallel, P_s, P_3 jmp4-send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0.

Fig. 22: 4PC: $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ jointly by P_i, P_j

When P_3, P_0 want to jointly share a value v which is available in the preprocessing phase, protocol Π_{jsh4} can be performed with a single element of communication (as opposed to 2 elements in Fig. 22). P_0, P_3 can jointly share v as follows. P_0, P_3, P_1 sample a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and set $[\alpha_{\mathsf{v}}]_1 = \mathsf{r}$. P_0, P_3 set $[\alpha_{\mathsf{v}}]_2 = -(\mathsf{r} + \mathsf{v})$ and jmp4-send $[\alpha_{\mathsf{v}}]_2$ to P_2. This is followed by servers locally setting $\gamma_{\mathsf{v}} = \beta_{\mathsf{v}} = 0$.

We further observe that servers can generate a $[\![\cdot]\!]$-sharing of v non-interactively when v is available with P_0, P_1, P_2. To do this, servers set $[\alpha_{\mathsf{v}}]_1 = [\alpha_{\mathsf{v}}]_2 = \gamma_{\mathsf{v}} = 0$ and $\beta_{\mathsf{v}} = \mathsf{v}$. We abuse notation and use $\Pi_{\mathsf{jsh4}}(P_0, P_1, P_2, \mathsf{v})$ to denote this sharing.

Lemma C.3 (Communication)**.** *In the online phase,* Π_{jsh4} *(Fig. 22) requires* 1 *round and an amortized communication of* 2ℓ *bits when* (P_3, P_s) *for* $s \in \{0, 1, 2\}$ *share a value, and requires an amortized communication of* ℓ *bits, otherwise.*

Proof: When (P_3, P_s) for $s \in \{0,1,2\}$ want to share a value v, there are two parallel calls to Π_{jmp4} which requires an amortized communication of 2ℓ bits and one round. In the other cases, Π_{jmp4} is invoked only once, resulting in an amortized communication of ℓ bits. ■

D. $\langle\cdot\rangle$-sharing Protocol

In some protocols, scenarios arise where P_3 is required to generate $\langle\cdot\rangle$-sharing of a value v in the preprocessing phase, where $\langle\cdot\rangle$-sharing of v is same as that defined in 3PC (where $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$, and P_0 possesses $(\mathsf{v}_0, \mathsf{v}_1)$, P_1 possesses $(\mathsf{v}_1, \mathsf{v}_2)$, and P_2 possess $(\mathsf{v}_2, \mathsf{v}_0)$) with the addition that P_3 now possesses $(\mathsf{v}_0, \mathsf{v}_1, \mathsf{v}_2)$. We call the resultant protocol Π_{ash4} and it appears in Fig. 23.

Protocol $\Pi_{\mathsf{ash4}}(P_3, \mathsf{v})$

Preprocessing :

- Servers P_0, P_3, P_1 sample a random $\mathsf{v}_1 \in \mathbb{Z}_{2^\ell}$, while servers P_0, P_3, P_2 sample a random $\mathsf{v}_0 \in \mathbb{Z}_{2^\ell}$.
- P_3 computes $\mathsf{v}_2 = \mathsf{v} - \mathsf{v}_0 - \mathsf{v}_1$ and sends v_2 to P_2. P_3, P_2 jmp4-send v_2 to P_1.

Fig. 23: 4PC: $\langle\cdot\rangle$-sharing of value v by P_3

Note that servers can locally convert $\langle\mathsf{v}\rangle$ to $[\![\mathsf{v}]\!]$ by setting their shares as shown in Table VIII.

	P_0	P_1	P_2	P_3
$[\![\mathsf{v}]\!]$	$(-\mathsf{v}_1, -\mathsf{v}_0, 0)$	$(-\mathsf{v}_1, \mathsf{v}_2, -\mathsf{v}_2)$	$(-\mathsf{v}_0, \mathsf{v}_2, -\mathsf{v}_2)$	$(-\mathsf{v}_0, -\mathsf{v}_1, -\mathsf{v}_2)$

TABLE VIII: Local conversion of shares from $\langle\cdot\rangle$-sharing to $[\![\cdot]\!]$-sharing for a value v. Here, $[\alpha_{\mathsf{v}}]_1 = -\mathsf{v}_1, [\alpha_{\mathsf{v}}]_2 = -\mathsf{v}_0, \beta_{\mathsf{v}} = \mathsf{v}_2, \gamma_{\mathsf{v}} = -\mathsf{v}_2$.

Lemma C.4 (Communication)**.** *Protocol Π_{ash4} (Fig. 23) requires 2 rounds and an amortized communication of 2ℓ bits.*

Proof: Communicating v_2 to P_2 requires ℓ bits and 1 round. This is followed by one invocation of Π_{jmp4} which requires ℓ bits and 1 round. Thus, the amortized communication cost is 2ℓ bits and two rounds. ■

E. Multiplication Protocol

Lemma C.5 (Communication)**.** *Π_{mult4} (Fig. 11) requires an amortized communication of 3ℓ bits in the preprocessing phase, and 1 round with an amortized communication of 3ℓ bits in the online phase.*

Proof: In the preprocessing phase, the servers execute Π_{jmp4} to jmp4-send $[\Gamma_{\mathsf{xy}}]_2$ to P_2 resulting in amortized communication of ℓ bits. This is followed by 2 parallel invocations of Π_{jmp4} to jmp4-send $[\chi]_1, [\chi]_2$ to P_0 which require an amortized communication of 2ℓ bits. Thus, the amortized communication cost in preprocessing is 3ℓ bits. In the online phase, there are 2 parallel invocations of Π_{jmp4} to jmp4-send $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$ to P_2, P_1, respectively, which requires amortized communication of 2ℓ bits and one round. This is followed by another call to Π_{jmp4} to jmp4-send $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0 which requires one more round and amortized communication of ℓ bits. However, jmp4-send of $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ can be delayed till the end of the protocol, and will require only one round for multiple multiplication gates and hence, can be amortized. Thus, the total number of rounds required for multiplication in the online phase is one with an amortized communication of 3ℓ bits. ■

F. Reconstruction Protocol

Lemma C.6 (Communication)**.** *Π_{rec4} (Fig. 12) requires an amortized communication of 8ℓ bits and 1 round in the online phase.*

Proof: Each P_s for $s \in \{0,1,2,3\}$ receives the missing share in clear from two other servers, while the hash of it from the third. As before, the missing share sent by the third server can be concatenated over multiple instances and hashed to obtain a single value. Thus, the amortized communication cost is 2ℓ bits per server, resulting in a total cost of 8ℓ bits. ■

G. Special protocols

Here we provide details regarding the special protocols - i) Bit Extraction, ii) Bit2A, and iii) Bit Injection.

1) Bit Extraction Protocol: This protocol enables the servers to compute a boolean sharing of the most significant bit (MSB) of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ given the arithmetic sharing $[\![\mathsf{v}]\!]$. To compute the MSB, we use the optimized Parallel Prefix Adder (PPA) circuit from ABY3 [4], which takes as input two boolean values and outputs the MSB of the sum of the inputs. The circuit requires $2(\ell - 1)$ AND gates and has a multiplicative depth of $\log \ell$. The protocol for bit extraction (Π_{bitext4}) involves computing the boolean PPA circuit using the protocols described in §IV. The two inputs to this boolean circuit are generated as follows. The value v whose MSB needs to be extracted can be represented as the sum of two values as $\mathsf{v} = \beta_{\mathsf{v}} + (-\alpha_{\mathsf{v}})$ where the first input to the circuit will be β_{v} and the second input will be $-\alpha_{\mathsf{v}}$. Since β_{v} is held by P_1, P_2, servers execute Π_{jsh4} to generate $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$. Similarly, P_0, P_3 possess α_{v}, and servers execute Π_{jsh4} to generate $[\![-\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$. Servers in $\mathcal{P}$ use the $[\![\cdot]\!]^{\mathbf{B}}$-shares of these two inputs $(\beta_{\mathsf{v}}, -\alpha_{\mathsf{v}})$ to compute the optimized PPA circuit which outputs the $[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}}$.

Lemma C.7 (Communication)**.** *The protocol Π_{bitext4} requires an amortized communication of $7\ell - 6$ bits in the preprocessing phase, and $\log \ell$ rounds with amortized communication of $7\ell - 6$ bits in the online phase.*

Proof: Generation of boolean sharing of α_{v} requires ℓ bits in the preprocessing phase (since Π_{jsh4} with P_0, P_3 can be achieved with ℓ bits of communication in the preprocessing phase), and generation of boolean sharing of β_{v} requires ℓ bits and one round (which can be deferred towards the end of the protocol thereby requiring one round for several instances) in the online phase. Further, the boolean PPA circuit to be computed requires $2(\ell - 1)$ AND gates. Since each AND gate requires Π_{mult4} to be executed, it requires an amortized communication of $6\ell - 6$ bits in both the preprocessing phase and the online phase. Thus, the overall communication is $7\ell - 6$ bits, in both, the preprocessing and online phase. The circuit has a multiplicative depth of $\log \ell$ which results in $\log \ell$ rounds in the online phase. ■

2) Bit2A Protocol: This protocol enables servers to compute the arithmetic sharing of a bit b given its boolean sharing. Let b^{R} denote the value of b in the ring $\mathbb{Z}_{2^\ell}$. We observe that b^{R} can be written as follows. $\mathsf{b}^{\mathsf{R}} = (\alpha_{\mathsf{b}} \oplus \beta_{\mathsf{b}})^{\mathsf{R}} = \alpha_{\mathsf{b}}^{\mathsf{R}} + \beta_{\mathsf{b}}^{\mathsf{R}} - 2\alpha_{\mathsf{b}}^{\mathsf{R}}\beta_{\mathsf{b}}^{\mathsf{R}}$. Thus, to obtain an arithmetic sharing of b^{R}, the servers can compute an arithmetic sharing of $\beta_{\mathsf{b}}^{\mathsf{R}}$, $\alpha_{\mathsf{b}}^{\mathsf{R}}$ and $\beta_{\mathsf{b}}^{\mathsf{R}}\alpha_{\mathsf{b}}^{\mathsf{R}}$. This can be done as follows. P_0, P_3 execute Π_{jsh4} on $\alpha_{\mathsf{b}}^{\mathsf{R}}$ in the preprocessing phase to generate $[\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$. Similarly, P_1, P_2 execute Π_{jsh4} on $\beta_{\mathsf{b}}^{\mathsf{R}}$ in the online phase to generate $[\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!]$. This is followed by Π_{mult4} on $[\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!], [\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$, followed by local computation to obtain $[\![\mathsf{b}^{\mathsf{R}}]\!]$.

Protocol $\Pi_{\mathsf{bit2A4}}(\mathcal{P}, [\![\mathsf{b}]\!]^{\mathbf{B}})$

Preprocessing :

- Servers execute $\Pi_{\mathsf{ash4}}(P_3, \mathsf{e}^{\mathsf{R}})$ (Fig. 23) where $\mathsf{e} = \alpha_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$. Let the shares be $\langle \mathsf{e}^{\mathsf{R}} \rangle_0 = (\mathsf{e}_0, \mathsf{e}_1), \langle \mathsf{e}^{\mathsf{R}} \rangle_1 = (\mathsf{e}_1, \mathsf{e}_2), \langle \mathsf{e}^{\mathsf{R}} \rangle_2 = (\mathsf{e}_2, \mathsf{e}_0), \langle \mathsf{e}^{\mathsf{R}} \rangle_3 = (\mathsf{e}_0, \mathsf{e}_1, \mathsf{e}_2)$.
- Verification of $\langle \mathsf{e}^{\mathsf{R}} \rangle$-sharing is performed as follows:
  - P_1, P_2, P_3 sample a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and a bit $\mathsf{r}_{\mathsf{b}} \in \mathbb{Z}_{2^1}$.
  - P_1, P_2 compute $\mathsf{x}_1 = \gamma_{\mathsf{b}} \oplus \mathsf{r}_{\mathsf{b}}$, and jmp4-send x_1 to P_0.
  - P_1, P_3 compute $\mathsf{y}_1 = (\mathsf{e}_1 + \mathsf{e}_2)(1 - 2\mathsf{r}_{\mathsf{b}}^{\mathsf{R}}) + \mathsf{r}_{\mathsf{b}}^{\mathsf{R}} + \mathsf{r}$, and jmp4-send y_1 to P_0.
  - P_2, P_3 compute $\mathsf{y}_2 = \mathsf{e}_0(1 - 2\mathsf{r}_{\mathsf{b}}^{\mathsf{R}}) - \mathsf{r}$, and jmp4-send $\mathsf{H}(\mathsf{y}_2)$ to P_0.
  - P_0 computes $\mathsf{x} = \mathsf{e} \oplus \mathsf{r}_{\mathsf{b}} = [\alpha_{\mathsf{b}}]_1 \oplus [\alpha_{\mathsf{b}}]_2 \oplus \mathsf{x}_1$ and checks if $\mathsf{H}(\mathsf{x}^{\mathsf{R}} - \mathsf{y}_1) = \mathsf{H}(\mathsf{y}_2)$.
  - If verification fails, P_0 sets $\mathsf{flag} = 1$, else it sets $\mathsf{flag} = 0$. P_0 sends flag to P_1. Next, P_1, P_0 jmp4-send flag to P_2 and P_3. Servers set $\mathsf{TTP} = P_1$ if $\mathsf{flag} = 1$.
- If verification succeeds, servers locally convert $\langle \mathsf{e}^{\mathsf{R}} \rangle$ to $[\![\mathsf{e}^{\mathsf{R}}]\!]$ by setting their shares according to Table VIII.

Online :

- Servers execute $\Pi_{\mathsf{jsh4}}(P_0, P_1, P_2, \mathsf{c}^{\mathsf{R}})$ where $\mathsf{c} = \beta_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$.
- Servers execute $\Pi_{\mathsf{mult4}}(\mathcal{P}, [\![\mathsf{e}^{\mathsf{R}}]\!], [\![\mathsf{c}^{\mathsf{R}}]\!])$ to generate $[\![\mathsf{e}^{\mathsf{R}}\mathsf{c}^{\mathsf{R}}]\!]$.
- Servers compute $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\mathsf{e}^{\mathsf{R}}]\!] + [\![\mathsf{c}^{\mathsf{R}}]\!] - 2[\![\mathsf{e}^{\mathsf{R}}\mathsf{c}^{\mathsf{R}}]\!]$.

Fig. 24: 4PC: Bit2A Protocol

While the above approach serves the purpose, we now provide an improved version, which further helps in reducing the online cost. We observe that b^{R} can be written as follows. $\mathsf{b}^{\mathsf{R}} = (\alpha_{\mathsf{b}} \oplus \beta_{\mathsf{b}})^{\mathsf{R}} = ((\alpha_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}) \oplus (\beta_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}))^{\mathsf{R}} = (\mathsf{e} \oplus \mathsf{c})^{\mathsf{R}} = \mathsf{e}^{\mathsf{R}} + \mathsf{c}^{\mathsf{R}} - 2\mathsf{e}^{\mathsf{R}}\mathsf{c}^{\mathsf{R}}$ where $\mathsf{e} = \alpha_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$ and $\mathsf{c} = \beta_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$. Thus, to obtain an arithmetic sharing of b^{R}, P_3 generates $\langle \cdot \rangle$-sharing of e^{R}. To ensure the correctness of the shares, the servers P_0, P_1, P_2 check whether the following equation holds: $(\mathsf{e} \oplus r_b)^{\mathsf{R}} = \mathsf{e}^{\mathsf{R}} + \mathsf{r}_b^{\mathsf{R}} - 2\mathsf{e}^{\mathsf{R}}\mathsf{r}_b^{\mathsf{R}}$. If the verification fails, a TTP is identified. Else, this is followed by servers locally converting $\langle \mathsf{e}^{\mathsf{R}} \rangle$-shares to $[\![\mathsf{e}^{\mathsf{R}}]\!]$ according to Table VIII, followed by multiplying $[\![\mathsf{e}^{\mathsf{R}}]\!], [\![\mathsf{c}^{\mathsf{R}}]\!]$ and locally computing $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\mathsf{e}^{\mathsf{R}}]\!] + [\![\mathsf{c}^{\mathsf{R}}]\!] - 2[\![\mathsf{e}^{\mathsf{R}}\mathsf{c}^{\mathsf{R}}]\!]$. Note that during $\Pi_{\mathsf{jsh4}}(P_0, P_1, P_2, \mathsf{c}^{\mathsf{R}})$ since $\alpha_{\mathsf{c}^{\mathsf{R}}}$ and $\gamma_{\mathsf{c}^{\mathsf{R}}}$ are set to 0, the preprocessing of multiplication can be performed locally.

Lemma C.8 (Communication)**.** *Π_{bit2A4} (Fig. 24) requires an amortized communication of $3\ell + 4$ bits in the preprocessing phase, and 1 round with amortized communication of 3ℓ bits in the online phase.*

Proof: During preprocessing, one instance of Π_{ash4} requires 2ℓ bits of communication. Further, sending x_1 requires 1 bit, while sending y_1 requires ℓ bits. Sending of $\mathsf{H}(\mathsf{y}_2)$ can be amortized over several instances. Finally, communicating flag requires 3 bits. Thus, the overall amortized communication cost in preprocessing phase is $3\ell + 4$ bits. In the online phase, joint sharing of c^{R} can be performed non-interactively. The only cost is due to the online phase of multiplication which requires 3ℓ bits and one round. Thus, the amortized communication cost in the online phase is 3ℓ bits with one round of communication. ■

3) Bit Injection Protocol: Given the boolean sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, and the arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, protocol Π_{bitinj4} computes $[\![\cdot]\!]$-sharing of bv. This can be naively computed by servers first executing Π_{bit2A4} on $[\![\mathsf{b}]\!]^{\mathbf{B}}$ to generate $[\![\mathsf{b}]\!]$, followed by servers computing $[\![\mathsf{bv}]\!]$ by executing Π_{mult4} protocol on $[\![\mathsf{b}]\!]$ and $[\![\mathsf{v}]\!]$. Instead, we provide an optimized variant which helps in reducing the communication cost of the naive approach in, both, the preprocessing and online phase. We give the details next.

Let $\mathsf{z} = \mathsf{b}^{\mathsf{R}}\mathsf{v}$, where b^{R} denotes the value of b in $\mathbb{Z}_{2^\ell}$. Then, during the computation of $[\![\mathsf{z}]\!]$, we observe the following:

$$\begin{aligned}
\mathsf{z} &= \mathsf{b}^{\mathsf{R}}\mathsf{v} = (\alpha_{\mathsf{b}} \oplus \beta_{\mathsf{b}})^{\mathsf{R}}(\beta_{\mathsf{v}} - \alpha_{\mathsf{v}}) \\
&= ((\alpha_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}) \oplus (\beta_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}))^{\mathsf{R}}((\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}) - (\alpha_{\mathsf{v}} + \gamma_{\mathsf{v}})) \\
&= (\mathsf{c}_{\mathsf{b}} \oplus \mathsf{e}_{\mathsf{b}})^{\mathsf{R}}(\mathsf{c}_{\mathsf{v}} - \mathsf{e}_{\mathsf{v}}) = (\mathsf{c}_{\mathsf{b}}^{\mathsf{R}} + \mathsf{e}_{\mathsf{b}}^{\mathsf{R}} - 2\mathsf{c}_{\mathsf{b}}^{\mathsf{R}}\mathsf{e}_{\mathsf{b}}^{\mathsf{R}})(\mathsf{c}_{\mathsf{v}} - \mathsf{e}_{\mathsf{v}}) \\
&= \mathsf{c}_{\mathsf{b}}^{\mathsf{R}}\mathsf{c}_{\mathsf{v}} - \mathsf{c}_{\mathsf{b}}^{\mathsf{R}}\mathsf{e}_{\mathsf{v}} + (\mathsf{c}_{\mathsf{v}} - 2\mathsf{c}_{\mathsf{b}}^{\mathsf{R}}\mathsf{c}_{\mathsf{v}})\mathsf{e}_{\mathsf{b}}^{\mathsf{R}} + (2\mathsf{c}_{\mathsf{b}}^{\mathsf{R}} - 1)\mathsf{e}_{\mathsf{b}}^{\mathsf{R}}\mathsf{e}_{\mathsf{v}}
\end{aligned}$$

where $\mathsf{c}_{\mathsf{b}} = \beta_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$, $\mathsf{e}_{\mathsf{b}} = \alpha_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$, $\mathsf{c}_{\mathsf{v}} = \beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ and $\mathsf{e}_{\mathsf{v}} = \alpha_{\mathsf{v}} + \gamma_{\mathsf{v}}$. The protocol proceeds with P_3 generating $\langle \cdot \rangle$-shares of $\mathsf{e}_{\mathsf{b}}^{\mathsf{R}}$ and $\mathsf{e}_{\mathsf{z}} = \mathsf{e}_{\mathsf{b}}^{\mathsf{R}}\mathsf{e}_{\mathsf{v}}$, followed by verification of the same by P_0, P_1, P_2. If verification succeeds, then to enable P_2 to compute $\beta_{\mathsf{z}} = \mathsf{z} + \alpha_{\mathsf{z}}$, P_1, P_0 jmp4-send the missing share of β_{z} to P_2. Similarly for P_1. Next, P_1, P_2 reconstruct β_{z}, and jmp4-send $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0 completing the protocol.

Lemma C.9 (Communication)**.** *Protocol Π_{bitinj4} requires an amortized communication cost of $6\ell + 4$ bits in the preprocessing phase, and requires 1 round with an amortized communication of 3ℓ bits in the online phase.*

Proof: The preprocessing phase requires two instances of Π_{ash4} which require 4ℓ bits of communication. Verifying correctness of $\langle \mathsf{e}_{\mathsf{b}}^{\mathsf{R}} \rangle$ requires $\ell + 1$ bits, whereas for $\langle \mathsf{e}_{\mathsf{z}} \rangle$ we require ℓ bits. Finally, communicating the flag requires 3 bits. This results in the amortized communication of $6\ell + 4$ bits in the preprocessing phase. The online phase consists of three calls to Π_{jmp4} which requires 3ℓ bits of amortized communication. Note that the last call can be deferred towards the end of the computation, thereby requiring a single round for multiple instances. Thus, the number of rounds required in the online phase is one. ■

Protocol $\Pi_{\mathsf{bitinj4}}(\mathcal{P}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{v}]\!])$

Let $\mathsf{c}_{\mathsf{b}} = \beta_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$, $\mathsf{e}_{\mathsf{b}} = \alpha_{\mathsf{b}} \oplus \gamma_{\mathsf{b}}$, $\mathsf{c}_{\mathsf{v}} = \beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$, $\mathsf{e}_{\mathsf{v}} = \alpha_{\mathsf{v}} + \gamma_{\mathsf{v}}$ and $\mathsf{e}_{\mathsf{z}} = \mathsf{e}_{\mathsf{b}}^{\mathsf{R}}\mathsf{e}_{\mathsf{v}}$.

Preprocessing :

- P_0, P_3, P_j for $j \in \{1, 2\}$ sample $[\alpha_{\mathsf{z}}]_1 \in \mathbb{Z}_{2^\ell}$ while P_1, P_2, P_3 sample $\gamma_{\mathsf{z}} \in \mathbb{Z}_{2^\ell}$.
- Servers execute $\Pi_{\mathsf{ash4}}(P_3, \mathsf{e}_{\mathsf{b}}^{\mathsf{R}})$ and $\Pi_{\mathsf{ash4}}(P_3, \mathsf{e}_{\mathsf{z}})$. Shares of $\langle \mathsf{e}_{\mathsf{v}} \rangle$ are set locally as $\mathsf{e}_{\mathsf{v}_0} = [\alpha_{\mathsf{v}}]_2$, $\mathsf{e}_{\mathsf{v}_1} = [\alpha_{\mathsf{v}}]_1$, $\mathsf{e}_{\mathsf{v}_3} = \gamma_{\mathsf{v}}$.

– Servers verify correctness of $\langle e_b^R \rangle$ using steps similar to Π_{bit2A4} (Fig. 24). Correctness of $\langle e_z \rangle$ is verified as follows.
  – P_0, P_3, P_j for $j \in \{1, 2\}$ sample a random $r_j \in \mathbb{Z}_{2^\ell}$ while P_1, P_2, P_3 sample a random $r_0 \in \mathbb{Z}_{2^\ell}$. P_0, P_3 set $a_0 = r_1 - r_2$, P_1, P_3 set $a_1 = r_0 - r_1$ and P_2, P_3 set $a_2 = r_2 - r_0$.
  – P_1, P_3 compute $x_1 = e_{v_2} e_{b_2} + e_{v_1} e_{b_2} + e_{v_2} e_{b_1} + a_1$.
  – P_2, P_3 compute $x_2 = e_{v_0} e_{b_0} + e_{v_0} e_{b_2} + e_{v_2} e_{b_0} + a_2$.
  – P_0 computes $x_0 = e_{v_1} e_{b_1} + e_{v_1} e_{b_0} + e_{v_0} e_{b_1} + a_0$.
  – P_1, P_3 jmp4-send $y_1 = x_1 - e_{z_1}$ to P_0, while P_2, P_3 jmp4-send $H(-y_2)$ to P_0, where $y_2 = x_2 - e_{z_2}$.
  – P_0 computes $y_0 = x_0 - e_{z_0}$, and checks if $H(y_0 + y_1) = H(-y_2)$.
  – If verification fails, P_0 sets flag $= 1$, else it sets flag $= 0$. P_0 sends flag to P_1. Next, P_1, P_0 jmp4-send flag to P_2 and P_3. Servers set TTP $= P_1$ if flag $= 1$.

Online :

– P_0, P_1 compute $u_1 = -c_b^R e_{v_1} + (c_v - 2c_b^R c_v) e_{b_1}^R + (2c_b^R - 1) e_{z_1} + [\alpha_z]_1$, and jmp4-send u_1 to P_2.
– P_0, P_2 compute $u_2 = -c_b^R e_{v_0} + (c_v - 2c_b^R c_v) e_{b_0}^R + (2c_b^R - 1) e_{z_0} + [\alpha_z]_2$, and jmp4-send u_2 to P_1.
– P_1, P_2 compute $\beta_z = u_1 + u_2 - c_b^R e_{v_2} + (c_v - 2c_b^R c_v) e_{b_2}^R + (2c_b^R - 1) e_{z_2} + c_b^R c_v$.
– P_1, P_2 jmp4-send $\beta_z + \gamma_z$ to P_0.

Fig. 25: 4PC: Bit Injection Protocol

H. Dot Product Protocol

The formal protocol for dot product is given in Fig. 26.

Protocol $\Pi_{\text{dotp4}}(\mathcal{P}, \{[\![x_i]\!], [\![y_i]\!]\}_{i\in[n]})$

Preprocessing :

– P_0, P_3, P_j, for $j \in \{1, 2\}$, sample random $[\alpha_z]_j \in \mathbb{Z}_{2^\ell}$, while P_0, P_1, P_3 sample random $[\Gamma_{\vec{x}\odot\vec{y}}]_1 \in \mathbb{Z}_{2^\ell}$.
– P_1, P_2, P_3 together sample random $\gamma_z, \psi, r \in \mathbb{Z}_{2^\ell}$ and set $[\psi]_1 = r$, $[\psi]_2 = \psi - r$.
– P_0, P_3 compute $[\Gamma_{\vec{x}\odot\vec{y}}]_2 = \Gamma_{\vec{x}\odot\vec{y}} - [\Gamma_{\vec{x}\odot\vec{y}}]_1$, where $\Gamma_{\vec{x}\odot\vec{y}} = \sum_{i=1}^{n} \alpha_{x_i} \alpha_{y_i}$. P_0, P_3 jmp4-send $[\Gamma_{\vec{x}\odot\vec{y}}]_2$ to P_2.
– P_3, P_j, for $j \in \{1, 2\}$, set $[\chi]_j = \sum_{i=1}^{n} (\gamma_{x_i} [\alpha_{y_i}]_j + \gamma_{y_i} [\alpha_{x_i}]_j) + [\Gamma_{\vec{x}\odot\vec{y}}]_j - [\psi]_j$.
– P_1, P_3 jmp4-send $[\chi]_1$ to P_0, and P_2, P_3 jmp4-send $[\chi]_2$ to P_0.

Online :

– P_0, P_j, for $j \in \{1, 2\}$, compute $[\beta_z^\star]_j = -\sum_{i=1}^{n} ((\beta_{x_i} + \gamma_{x_i}) [\alpha_{y_i}]_j + (\beta_{y_i} + \gamma_{y_i}) [\alpha_{x_i}]_j) + [\alpha_z]_j + [\chi]_j$.
– P_1, P_0 jmp4-send $[\beta_z^\star]_1$ to P_2, while P_2, P_0 jmp4-send $[\beta_z^\star]_2$ to P_1.
– P_j for $j \in \{1, 2\}$ computes $\beta_z^\star = [\beta_z^\star]_1 + [\beta_z^\star]_2$ and sets $\beta_z = \beta_z^\star + \sum_{i=1}^{n} (\beta_{x_i} \beta_{y_i}) + \psi$.
– P_1, P_2 jmp4-send $\beta_z + \gamma_z$ to P_0.

Fig. 26: 4PC: Dot Product Protocol ($z = \vec{x} \odot \vec{y}$)

Lemma C.10 (Communication). *Π_{dotp4} (Fig. 26) requires an amortized communication of 3ℓ bits in the preprocessing phase, and 1 round and an amortized communication of 3ℓ bits in the online phase.*

Proof: The preprocessing phase requires three calls to Π_{jmp4}, one to jmp4-send $[\Gamma_{\vec{x}\odot\vec{y}}]_2$ to P_2, and two to jmp4-send $[\chi]_1, [\chi]_2$ to P_0. Each invocation of Π_{jmp4} requires ℓ bits resulting in the amortized communication cost of preprocessing phase to be 3ℓ bits. In the online phase, there are 2 parallel invocations of Π_{jmp4} to jmp4-send $[\beta_z^\star]_1, [\beta_z^\star]_2$ to P_2, P_1, respectively, which require amortized communication of 2ℓ bits and one round. This is followed by another call to Π_{jmp4} to jmp4-send $\beta_z + \gamma_z$ to P_0 which requires one more round and amortized communication of ℓ bits. As in the multiplication protocol, this step can be delayed till the end of the protocol and clubbed for multiple instances. Thus, the online phase requires one round and an amortized communication of 3ℓ bits. ■

I. Truncation

Given the $[\![\cdot]\!]$-sharing of a value v, this protocol enables the servers to compute the $[\![\cdot]\!]$-sharing of the truncated value v^d (right shifted value by, say, d positions). Given $[\![v]\!]$ and a random truncation pair $([r], [\![r^d]\!])$, the value $(v - r)$ is opened, truncated and added to $[\![r^d]\!]$ to obtain $[\![v^d]\!]$. The protocol for generating the truncation pair $([r], [\![r^d]\!])$ is described in Fig. 27.

Protocol $\Pi_{\text{trgen4}}(\mathcal{P})$

– P_0, P_3, P_j, for $j \in \{1, 2\}$ sample random $R_j \in \mathbb{Z}_{2^\ell}$. P_0, P_3 sets $r = R_1 + R_2$ while P_j sets $[r]_j = R_j$.
– P_0, P_3 locally truncate r to obtain r^d and execute $\Pi_{\text{jsh4}}(P_0, P_3, r^d)$ to generate $[\![r^d]\!]$.

Fig. 27: 4PC: Generating Random Truncated Pair (r, r^d)

Lemma C.11 (Communication). *Π_{trgen4} (Fig. 27) requires 1 round and an amortized communication of ℓ bits in the online phase.*

Proof: The cost follows directly from that of Π_{jsh4} (Lemma C.1). ■

J. Dot Product with Truncation

Protocol Π_{dotpt4} (Fig. 28) enables servers to generate $[\![\cdot]\!]$-sharing of the truncated value of $z = \vec{x} \odot \vec{y}$, denoted as z^d, given the $[\![\cdot]\!]$-sharing of n-sized vectors $\vec{x}$ and $\vec{y}$.

Protocol $\Pi_{\text{dotpt4}}(\mathcal{P}, \{[\![x_i]\!], [\![y_i]\!]\}_{i\in[n]})$

Preprocessing :

– Servers execute the preprocessing phase of $\Pi_{\text{dotp4}}(\mathcal{P}, \{[\![x_i]\!], [\![y_i]\!]\}_{i\in[n]})$.
– Servers execute $\Pi_{\text{trgen4}}(\mathcal{P})$ to generate the truncation pair $([r], [\![r^d]\!])$. P_0 obtains the value r in clear.

Online :

– P_0, P_j, for $j \in \{1, 2\}$, compute $[\Psi]_j = -\sum_{i=1}^{n} ((\beta_{x_i} + \gamma_{x_i}) [\alpha_{y_i}]_j + (\beta_{y_i} + \gamma_{y_i}) [\alpha_{x_i}]_j) - [r]_j$ and sets $[(z - r)^\star]_j = [\Psi]_j + [\chi]_j$.
– P_1, P_0 jmp4-send $[(z - r)^\star]_1$ to P_2 and P_2, P_0 jmp4-send $[(z - r)^\star]_2$ to P_1.
– P_1, P_2 locally compute $(z - r)^\star = [(z - r)^\star]_1 + [(z - r)^\star]_2$ and set $(z - r) = (z - r)^\star + \sum_{i=1}^{n} (\beta_{x_i} \beta_{y_i}) + \psi$.
– P_1, P_2 locally truncate $(z - r)$ to obtain $(z - r)^d$ and execute $\Pi_{\text{jsh4}}(P_1, P_2, (z - r)^d)$ to generate $[\![(z - r)^d]\!]$.
– Servers locally compute $[\![z^d]\!] = [\![(z - r)^d]\!] + [\![r^d]\!]$.

Fig. 28: 4PC: Dot Product Protocol with Truncation

Lemma C.12 (Communication)**.** *Π_{dotpt4} (Fig. 28) requires an amortized communication of 4ℓ bits in the preprocessing phase, and 1 round with amortized communication of 3ℓ bits in the online phase.*

Proof: The preprocessing phase comprises of the preprocessing phase of Π_{dotp4} and Π_{trgen4} which results in an amortized communication of $3\ell + \ell = 4\ell$ bits. The online phase follows from that of Π_{dotp4} protocol except that, now, P_1, P_2 compute $[\cdot]$-shares of $\mathsf{z} - \mathsf{r}$. This requires one round and an amortized communication cost of 2ℓ bits. P_1, P_2 then jointly share the truncated value of $\mathsf{z} - \mathsf{r}$ with P_0, which requires 1 round and ℓ bits. However, this step can be deferred till the end for multiple instances, which results in amortizing this round. The total amortized communication is thus 3ℓ bits in online phase. ∎

K. Activation Functions

Lemma C.13 (Communication)**.** *Protocol for* relu *requires an amortized communication of $13\ell - 2$ bits in the preprocessing phase and requires $\log \ell + 1$ rounds and an amortized communication of $10\ell - 6$ bits in the online phase.*

Proof: One instance of relu protocol comprises of execution of one instance of Π_{bitext4}, followed by Π_{bitinj4}. The cost, therefore, follows from Lemma C.7, and Lemma C.9. ∎

Lemma C.14 (Communication)**.** *Protocol for* sig *requires an amortized communication of $23\ell - 1$ bits in the preprocessing phase and requires $\log \ell + 2$ rounds and an amortized communication of $20\ell - 9$ bits in the online phase.*

Proof: An instance of sig protocol involves the execution of the following protocols in order– i) two parallel instances of Π_{bitext4} protocol, ii) one instance of Π_{mult4} protocol over boolean value, and iii) one instance of Π_{bitinj4} and Π_{bit2A4} in parallel. The cost follows from Lemma C.7, Lemma C.8 and Lemma C.9. ∎

Appendix D
Security Analysis of Our Protocols

In this section, we provide detailed security proofs for our constructions in both the 3PC and 4PC domains. We prove security using the real-world/ ideal-word simulation based technique. We provide proofs in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model for the case of 3PC, where $\mathcal{F}_{\mathsf{setup}}$ (Fig. 14) denotes the ideal functionality for the three server shared-key setup. Similarly, the proofs for 4PC are provided in the $\mathcal{F}_{\mathsf{setup4}}$-hybrid model (Fig. 15).

Let $\mathcal{A}$ denote the real-world adversary corrupting at most one server in $\mathcal{P}$, and $\mathcal{S}$ denote the corresponding ideal world adversary. The strategy for simulating the computation of function f (represented by a circuit ckt) is as follows: The simulation begins with the simulator emulating the shared-key setup ($\mathcal{F}_{\mathsf{setup}}/\mathcal{F}_{\mathsf{setup4}}$) functionality and giving the respective keys to the adversary. This is followed by the input sharing phase in which $\mathcal{S}$ extracts the input of $\mathcal{A}$, using the known keys, and sets the inputs of the honest parties to be 0. $\mathcal{S}$ now knows all the inputs and can compute all the intermediate values for each of the building blocks in clear. Also, $\mathcal{S}$ can obtain the output of the ckt in clear. $\mathcal{S}$ now proceeds simulating each of the building block in topological order using the aforementioned values (inputs of $\mathcal{A}$, intermediate values and circuit output).

In some of our sub protocols, adversary is able to decide on which among the honest parties should be chosen as the Trusted Third Party (TTP) in that execution of the protocol. To capture this, we consider *corruption-aware* functionalities [54] for the sub-protocols, where the functionality is provided the identity of the corrupt server as an auxiliary information.

For modularity, we provide the simulation steps for each of the sub-protocols separately. These steps, when carried out in the respective order, result in the simulation steps for the entire 3/4PC protocol. If a TTP is identified during the simulation of any of the sub-protocols, simulator will stop the simulation at that step. In the next round, the simulator receives the input of the corrupt party in clear on behalf of the TTP for the 3PC case, whereas it receives the input shares from adversary for 4PC.

A. Security Proofs for 3PC protocols

The ideal functionality $\mathcal{F}_{\mathsf{3PC}}$ for evaluating ckt in the 3PC setting appears in Fig. 29.

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

$\mathcal{F}_{\mathsf{3PC}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. Let f denote the functionality to be computed. Let x_s be the input corresponding to the server P_s, and y_s be the corresponding output, i.e $(\mathsf{y}_0, \mathsf{y}_1, \mathsf{y}_2) = f(\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2)$.

Step 1: $\mathcal{F}_{\mathsf{3PC}}$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$, and computes $(\mathsf{y}_0, \mathsf{y}_1, \mathsf{y}_2) = f(\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2)$.

Step 2: $\mathcal{F}_{\mathsf{3PC}}$ sends $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$.

Fig. 29: 3PC: Ideal functionality for evaluating a function f

Now we provide the simulation for each of the sub-protocols.

1) Joint Message Passing (jmp) *Protocol:* This section provides the security proof for the jmp primitive, which forms the crux for achieving GOD guarantee in our constructions. Let $\mathcal{F}_{\mathsf{jmp}}$ (Fig. 1) denote the ideal functionality and let $\mathcal{S}_{\mathsf{jmp}}^{P_s}$ denote the corresponding simulator for the case of corrupt $P_s \in \mathcal{P}$.

We begin with the case for a corrupt sender, P_i. The case for a corrupt P_j is similar and hence we omit details for the same.

Simulator $\mathcal{S}_{\mathsf{jmp}}^{P_i}$

- $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ initializes $\mathsf{ttp} = \perp$ and receives v_i from $\mathcal{A}$ on behalf of P_k.
- In case, $\mathcal{A}$ fails to send a value $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ broadcasts "`(accuse,` P_i`)`", sets $\mathsf{ttp} = P_j$, $\mathsf{v}_i = \perp$, and skip to the last step.
- Else, it checks if $\mathsf{v}_i = \mathsf{v}$, where v is the value computed by $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ based on the interaction with $\mathcal{A}$, and using the knowledge of the shared keys. If the values are equal, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $b_k = 0$, else, sets $b_k = 1$, and sends the same to $\mathcal{A}$ on the behalf of P_k.
- If $\mathcal{A}$ broadcasts "`(accuse,` P_k`)`", $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{v}_i = \perp$, $\mathsf{ttp} = P_j$, and skips to the last step.
- $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ computes and sends b_j to $\mathcal{A}$ on behalf of P_j and receives

$b_{\mathcal{A}}$ from $\mathcal{A}$ on behalf of honest P_j.
– If $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ does not receive a $b_{\mathcal{A}}$ on behalf of P_j, it broadcasts "(accuse, P_i)", sets $\mathsf{v}_i = \perp$, $\mathsf{ttp} = P_k$. If $\mathcal{A}$ broadcasts "(accuse, P_j)", $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{v}_i = \perp$, $\mathsf{ttp} = P_k$. If ttp is set, skip to the last step.
– If $(\mathsf{v}_i = \mathsf{v})$ and $b_{\mathcal{A}} = 1$, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ broadcasts $\mathsf{H}_j = \mathsf{H}(\mathsf{v})$ on behalf of P_j.
– Else if $\mathsf{v}_i \neq \mathsf{v}_j$: $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ broadcasts $\mathsf{H}_j = \mathsf{H}(\mathsf{v})$ and $\mathsf{H}_k = \mathsf{H}(\mathsf{v}_i)$ on behalf of P_j and P_k, respectively. If $\mathcal{A}$ does not broadcast, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{ttp} = P_k$. Else if, $\mathcal{A}$ broadcasts a value $\mathsf{H}_{\mathcal{A}}$:
  - If $\mathsf{H}_{\mathcal{A}} \neq \mathsf{H}_j$: $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{ttp} = P_k$.
  - Else if $\mathsf{H}_{\mathcal{A}} \neq \mathsf{H}_k$: $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{ttp} = P_j$.
– $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ invokes $\mathcal{F}_{\mathsf{jmp}}$ on $(\mathsf{Input}, \mathsf{v}_i)$ and $(\mathsf{Select}, \mathsf{ttp})$ on behalf of $\mathcal{A}$.

Fig. 30: Simulator $\mathcal{S}_{\mathsf{jmp4}}^{P_i}$ for corrupt sender P_i

The case for a corrupt receiver, P_k is provided in Fig. 31.

Simulator $\mathcal{S}_{\mathsf{jmp}}^{P_k}$

– $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ initializes $\mathsf{ttp} = \perp$, computes v honestly and sends v and $\mathsf{H}(\mathsf{v})$ to $\mathcal{A}$ on behalf of P_i and P_j, respectively.
– If $\mathcal{A}$ broadcasts "(accuse, P_i)", set $\mathsf{ttp} = P_j$, else if $\mathcal{A}$ broadcasts "(accuse, P_j)", set $\mathsf{ttp} = P_i$. If both messages are broadcast, set $\mathsf{ttp} = P_i$. If ttp is set skip to the last step.
– On behalf of P_i, P_j, $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ receives $b_{\mathcal{A}}$ from $\mathcal{A}$. Let b_i (resp. b_j) denote the bit received by P_i (resp. P_j) from $\mathcal{A}$.
– If $\mathcal{A}$ failed to send bit $b_{\mathcal{A}}$ to P_i, $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ broadcasts "(accuse, P_k)", set $\mathsf{ttp} = P_j$. Similarly, for P_j. If both P_i, P_j broadcast "(accuse, P_k)", set $\mathsf{ttp} = P_i$. If ttp is set, skip to the last step.
– If $b_i \vee b_j = 1$: $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ broadcasts $\mathsf{H}_i, \mathsf{H}_j$ where $\mathsf{H}_i = \mathsf{H}_j = \mathsf{H}(\mathsf{v})$ on behalf of P_i, P_j, respectively.
– If $\mathcal{A}$ does not broadcast $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ sets $\mathsf{ttp} = \perp$. If $\mathcal{A}$ broadcasts a value $\mathsf{H}_{\mathcal{A}}$:
  - If $\mathsf{H}_{\mathcal{A}} \neq \mathsf{H}_i$: $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ sets $\mathsf{ttp} = P_j$.
  - Else if $\mathsf{H}_{\mathcal{A}} = \mathsf{H}_i = \mathsf{H}_j$: $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ sets $\mathsf{ttp} = P_i$.
– $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ invokes $\mathcal{F}_{\mathsf{jmp}}$ on $(\mathsf{Input}, \perp)$ and $(\mathsf{Select}, \mathsf{ttp})$ on behalf of $\mathcal{A}$.

Fig. 31: Simulator $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ for corrupt receiver P_k

2) Sharing Protocol: Here we give he simulation steps for Π_{sh}. The case for a corrupt P_0 is provided in Fig. 37.

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

Preprocessing: $\mathcal{S}_{\mathsf{sh}}^{P_0}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the keys $(k_{01}, k_{02}, k_{\mathcal{P}})$ to $\mathcal{A}$. The values that are commonly held along with $\mathcal{A}$ are sampled using appropriate shared key. Otherwise, values are sampled randomly.

Online:

– If the dealer $P_s = P_0$:
  - $\mathcal{S}_{\mathsf{sh}}^{P_0}$ receives β_{v} on behalf of P_1 and sets $\mathsf{msg} = \mathsf{v}$ accordingly.
  - Steps for Π_{jmp} protocol are simulated according to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ (Fig. 30), where P_0 plays the role of one of the senders.
– If the dealer $P_s = P_1$:
  - $\mathcal{S}_{\mathsf{sh}}^{P_0}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$.
  - Steps for Π_{jmp} protocol are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ (Fig. 31), with P_0 acting as the receiver.
– If the dealer if P_2 : Similar to the case when $P_s = P_1$.

Fig. 32: Simulator $\mathcal{S}_{\mathsf{sh}}^{P_0}$ for corrupt P_0

The case for a corrupt P_1 is provided in Fig. 33. The case for a corrupt P_2 is similar.

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

Preprocessing: $\mathcal{S}_{\mathsf{jsh}}^{P_1}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the keys $(k_{01}, k_{12}, k_{\mathcal{P}})$ to $\mathcal{A}$. The values that are commonly held along with $\mathcal{A}$ are sampled using appropriate shared key. Otherwise, values are sampled randomly.

Online:

– If dealer $P_s = P_1$: $\mathcal{S}_{\mathsf{sh}}^{P_1}$ receives β_{v} from $\mathcal{A}$ on behalf of P_2.
– If $P_s = P_0$: $\mathcal{S}_{\mathsf{sh}}^{P_1}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$ and sends β_{v} to $\mathcal{A}$ on behalf of P_s.
– If $P_s = P_2$: Similar to the case where $P_s = P_0$.
– Steps of Π_{jmp}, in all the steps above, are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ (Fig. 30), ie. the case of corrupt sender.

Fig. 33: Simulator $\mathcal{S}_{\mathsf{sh}}^{P_1}$ for corrupt P_1

3) Multiplication Protocol: Here we give he simulation steps for Π_{mult}. The case for a corrupt P_0 is provided in Fig. 34.

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

Preprocessing:

– $\mathcal{S}_{\mathsf{mult}}^{P_0}$ samples $[\alpha_{\mathsf{z}}]_1, [\alpha_{\mathsf{z}}]_2$ and γ_{z} on behalf of P_1, P_2 and generates the $\langle\cdot\rangle$-shares of d, e honestly.
– $\mathcal{S}_{\mathsf{mult}}^{P_0}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$, and extracts ψ, $[\chi]_1, [\chi]_2$ on behalf of P_1, P_2.

Online:

– $\mathcal{S}_{\mathsf{mult}}^{P_0}$ computes $[\beta_{\mathsf{z}}^{\star}]_1, [\beta_{\mathsf{z}}^{\star}]_2$ and steps of Π_{jmp} are simulated according to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ with $\mathcal{A}$ as one of the sender for both $[\beta_{\mathsf{z}}^{\star}]_1$, and $[\beta_{\mathsf{z}}^{\star}]_2$.
– $\mathcal{S}_{\mathsf{mult}}^{P_0}$ computes $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ on behalf of P_1, P_2 and steps of Π_{jmp} are simulated according to $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ with $\mathcal{A}$ as the receiver for $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 34: Simulator $\mathcal{S}_{\mathsf{mult}}^{P_0}$ for corrupt P_0

The case for a corrupt P_1 is provided in Fig. 35. The case for a corrupt P_2 is similar.

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

Preprocessing:

– $\mathcal{S}_{\mathsf{mult}}^{P_1}$ samples $[\alpha_{\mathsf{z}}]_1, \gamma_{\mathsf{z}}$ and $[\alpha_{\mathsf{z}}]_2$ on behalf of P_0, P_2. $\mathcal{S}_{\mathsf{mult}}^{P_1}$ generates the $\langle\cdot\rangle$-shares of d, e honestly.
– $\mathcal{S}_{\mathsf{mult}}^{P_1}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$, and extracts $\psi, [\chi]_1, [\chi]_2$ on behalf of P_0, P_2.

Online:

– $\mathcal{S}_{\mathsf{mult}}^{P_1}$ computes $[\beta_{\mathsf{z}}^{\star}]_1, [\beta_{\mathsf{z}}^{\star}]_2$ on behalf of P_0, P_2, and steps of Π_{jmp} are simulated according to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ with $\mathcal{A}$ as one of the sender for $[\beta_{\mathsf{z}}^{\star}]_1$, and as the receiver for $[\beta_{\mathsf{z}}^{\star}]_2$.
– $\mathcal{S}_{\mathsf{mult}}^{P_1}$ computes $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ on behalf of P_2 and steps of Π_{jmp} are simulated according to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ with $\mathcal{A}$ one of the senders for $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 35: Simulator $\mathcal{S}_{\mathsf{mult}}^{P_1}$ for corrupt P_1

4) Reconstruction Protocol: Here we give he simulation steps for Π_{rec}. The case for a corrupt P_0 is provided in Fig. 52. The case for a corrupt P_1, P_2 is similar.

Simulator $\mathcal{S}_{\mathsf{rec}}$

Preprocessing:

- $\mathcal{S}_{\mathsf{rec}}$ computes commitments on $[\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2$ and γ_{v} on behalf of P_1, P_2, using the respective shared keys.
- The steps of Π_{jmp} are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ with $\mathcal{A}$ acting as the receiver for $\mathsf{Com}(\gamma_{\mathsf{v}})$, and $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ with $\mathcal{A}$ acting as one of the senders for $\mathsf{Com}([\alpha_{\mathsf{v}}]_1)$ and $\mathsf{Com}([\alpha_{\mathsf{v}}]_2)$.

Online:

- $\mathcal{S}_{\mathsf{rec}}$ receives openings for $\mathsf{Com}([\alpha_{\mathsf{v}}]_1), \mathsf{Com}([\alpha_{\mathsf{v}}]_2)$ on behalf of P_2 and P_1, respectively.
- $\mathcal{S}_{\mathsf{rec}}$ opens $\mathsf{Com}(\gamma_{\mathsf{v}})$ to $\mathcal{A}$ on behalf of P_1, P_2.

Fig. 36: Simulator $\mathcal{S}_{\mathsf{rec}}$ for corrupt P_0

5) Joint Sharing Protocol: Here we give he simulation steps for Π_{jsh}. The case for a corrupt P_0 is provided in Fig. 37. The case for a corrupt P_1, P_2 is similar.

Simulator $\mathcal{S}_{\mathsf{jsh}}$

Preprocessing: $\mathcal{S}_{\mathsf{sh}}^{P_0}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the keys $(k_{01}, k_{02}, k_{\mathcal{P}})$ to $\mathcal{A}$. The values that are commonly held along with $\mathcal{A}$ are sampled using appropriate shared key. Otherwise, values are sampled randomly.

Online:

- If $(P_i, P_j) = (P_1, P_0)$: $\mathcal{S}_{\mathsf{jsh}}$ computes $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ on behalf of P_1. The steps of Π_{jmp} are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$, where the $\mathcal{A}$ acts as one of the senders.
- If $(P_i, P_j) = (P_2, P_0)$: Similar to the case when $(P_i, P_j) = (P_1, P_0)$.
- If $(P_i, P_j) = (P_1, P_2)$: $\mathcal{S}_{\mathsf{jsh}}$ sets $\mathsf{v} = 0$ by setting $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$. The steps of Π_{jmp} are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_k}$, where the $\mathcal{A}$ acts as the receiver.

Fig. 37: Simulator $\mathcal{S}_{\mathsf{jsh}}$ for corrupt P_0

6) Dot Product Protocol: Here we give he simulation steps for Π_{dotp}. The case for a corrupt P_0 is provided in Fig. 38.

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

Preprocessing: $\mathcal{S}_{\mathsf{dotp}}$ emulates $\mathcal{F}_{\mathsf{DotPPre}}$ and derives ψ and respective $[\cdot]$-shares of χ honestly on behalf of P_1, P_2.

Online:

- $\mathcal{S}_{\mathsf{dotp}}^{P_0}$ computes $[\cdot]$-shares of $\beta_{\mathsf{z}}^{\star}$ on behalf of P_1, P_2. The steps of Π_{jmp}, required to provide P_1 with $[\beta_{\mathsf{z}}^{\star}]_2$, and P_2 with $[\beta_{\mathsf{z}}^{\star}]_1$, are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$, where P_0 acts as one of the sender in both cases.
- $\mathcal{S}_{\mathsf{dotp}}^{P_0}$ computes $\beta_{\mathsf{z}}^{\star}$ and β_{z} on behalf of P_1, P_2. The steps of the Π_{jmp}, required to provide P_0 with $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$, are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_k}$, where P_0 acts as the receiver.

Fig. 38: Simulator $\mathcal{S}_{\mathsf{dotp}}$ for corrupt P_0

The case for a corrupt P_1 is provided in Fig. 39. The case for a corrupt P_2 is similar.

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

Preprocessing: $\mathcal{S}_{\mathsf{dotp}}^{P_1}$ emulates $\mathcal{F}_{\mathsf{DotPPre}}$ and derives $[\cdot]$-shares of ψ, χ honestly on behalf of P_0, P_2.

Online:

- $\mathcal{S}_{\mathsf{dotp}}^{P_1}$ computes $[\beta_{\mathsf{z}}^{\star}]_1$ on behalf of P_0, and $[\beta_{\mathsf{z}}^{\star}]_2$ on behalf of P_0 and P_2. The steps of Π_{jmp}, required to provide P_1 with $[\beta_{\mathsf{z}}^{\star}]_2$, and P_2 with $[\beta_{\mathsf{z}}^{\star}]_1$, are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$, where $\mathcal{A}$ acts as one of the sender in the former case, and as a receiver in the latter case.
- $\mathcal{S}_{\mathsf{dotp}}^{P_1}$ computes $\beta_{\mathsf{z}}^{\star}$ and β_{z} on behalf of P_2. The steps of Π_{jmp}, required to provide P_0 with $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$, are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$, where $\mathcal{A}$ acts as one of the sender.

Fig. 39: Simulator $\mathcal{S}_{\mathsf{dotp}}$ for corrupt P_1

7) Truncation Protocol: Here we give he simulation steps for Π_{trgen}. The case for a corrupt P_0 is provided in Fig. 40.

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

- $\mathcal{S}_{\mathsf{trgen}}^{P_0}$ samples R_1, R_2 using the respective keys with $\mathcal{A}$.
- Steps corresponding to Π_{sh} are simulated similar to the steps $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{P_0}$ for corrupt P_0.
- $\mathcal{S}_{\mathsf{trgen}}^{P_0}$ computes u, and steps corresponding to Π_{jmp} are simulated similar to $\mathcal{S}_{\Pi_{\mathsf{jmp}}}^{P_i}$.
- $\mathcal{S}_{\mathsf{trgen}}^{P_0}$ computes v. If $\mathsf{H}(\mathsf{u}) \neq \mathsf{H}(\mathsf{v})$, $\mathcal{S}_{\mathsf{trgen}}^{P_0}$ broadcasts "$(\mathtt{accuse}, P_0)$", and sets $\mathsf{ttp} = P_1$.

Fig. 40: Simulator $\mathcal{S}_{\mathsf{trgen}}^{P_0}$ for corrupt P_0

The case for a corrupt P_1 is provided in Fig. 41. The case for a corrupt P_2 is similar.

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

- $\mathcal{S}_{\mathsf{trgen}}^{P_1}$ samples R_1 using the key k_{01} with $\mathcal{A}$, and samples random R_2. $\mathcal{S}_{\mathsf{trgen}}^{P_1}$ sets $\mathsf{r} = R_1 + R_2$, and truncates it to obtain r_d.
- Steps corresponding to Π_{jsh} are simulated similar to the steps in $\mathcal{S}_{\Pi_{\mathsf{jsh}}}$. $\mathcal{S}_{\mathsf{trgen}}^{P_1}$ computes u, and steps corresponding to Π_{jmp} are simulated similar to the steps in $\mathcal{S}_{\Pi_{\mathsf{jmp}}}^{P_i}$.

Fig. 41: Simulator $\mathcal{S}_{\mathsf{trgen}}^{P_1}$ for corrupt P_1

Observe from the simulation steps, that the view of $\mathcal{A}$ in the real world and the ideal world is indistinguishable.

B. Security Proofs for 4PC protocols

The ideal functionality $\mathcal{F}_{\mathsf{4PC}}$ for evaluating a function f to be computed by ckt in the 4PC setting appears in Fig. 42.

Functionality $\mathcal{F}_{\mathsf{4PC}}$

$\mathcal{F}_{\mathsf{4PC}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. Let f denote the function to be computed. Let x_s be the input corresponding to the server P_s, and y_s be the corresponding output, i.e $(\mathsf{y}_0, \mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3) = f(\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3)$.

Step 1: $\mathcal{F}_{\mathsf{4PC}}$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$, and computes $(\mathsf{y}_0, \mathsf{y}_1, \mathsf{y}_3, \mathsf{y}_3) = f(\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3)$.

Step 2: $\mathcal{F}_{\mathsf{4PC}}$ sends $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$.

Fig. 42: 4PC: Ideal functionality for computing f in 4PC setting

Now we provide the simulation for each of the sub-protocols.

1) Joint Message Passing: This section provides the security proof for the jmp4 primitive, which forms the crux for achieving GOD in our constructions. Let $\mathcal{F}_{\text{jmp4}}$ Fig. 21 denote the ideal functionality and let $\mathcal{S}_{\text{jmp4}}^{P_s}$ denote the corresponding simulator for the case of corrupt $P_s \in \mathcal{P}$.

We begin with the case for a corrupt sender, P_i, which is provided in Fig. 43. The case for a corrupt P_j is similar and hence we omit details for the same.

Simulator $\mathcal{S}_{\text{jmp4}}^{P_i}$

– $\mathcal{S}_{\text{jmp4}}^{P_i}$ receives v_i from $\mathcal{A}$ on behalf of honest P_k. If $\mathsf{v}_i = \mathsf{v}_j$ (where v_j is the value computed by $\mathcal{S}_{\text{jmp4}}^{P_i}$ based on the interaction with $\mathcal{A}$, and using the knowledge of the shared keys), then $\mathcal{S}_{\text{jmp4}}^{P_i}$ sets $b_k = 0$, else it sets $b_k = 1$. If $\mathcal{A}$ fails to send a value, b_k is set to be 1. $\mathcal{S}_{\text{jmp4}}^{P_i}$ sends b_k to $\mathcal{A}$ on behalf of P_k.
– $\mathcal{S}_{\text{jmp4}}^{P_i}$ sends $b_j = b_k$ to $\mathcal{A}$, and receives b_i from $\mathcal{A}$ on behalf of honest P_j. If $\mathcal{A}$ fails to send a value, it is assumed to be 1.
– $\mathcal{S}_{\text{jmp4}}^{P_i}$ receives b_i from $\mathcal{A}$ on behalf of honest P_l. If $b_k = 1$, $\mathcal{S}_{\text{jmp4}}^{P_i}$ sets $b_l = 1$ and $\mathsf{ttp} = P_l$, else it sets $b_l = 0$ and $\mathsf{ttp} = \perp$. $\mathcal{S}_{\text{jmp4}}^{P_i}$ sends b_l to $\mathcal{A}$ on behalf of P_l. $\mathcal{S}_{\text{jmp4}}^{P_i}$ invokes $\mathcal{F}_{\text{jmp4}}$ with $(\mathsf{Input}, \mathsf{v}_i)$ and (Select, b_l) on behalf of $\mathcal{A}$.

Fig. 43: Simulator $\mathcal{S}_{\text{jmp4}}^{P_i}$ for corrupt sender P_i

The case for a corrupt receiver, P_k is provided in Fig. 44.

Simulator $\mathcal{S}_{\text{jmp4}}^{P_k}$

– $\mathcal{S}_{\text{jmp4}}^{P_k}$ sends v, $\mathsf{H}(\mathsf{v})$ (where v is the value computed by $\mathcal{S}_{\text{jmp4}}^{P_k}$ based on the interaction with $\mathcal{A}$, and using the knowledge of the shared keys) to $\mathcal{A}$ on behalf of honest P_i, P_j, respectively.
– $\mathcal{S}_{\text{jmp4}}^{P_k}$ receives b_{ki}, b_{kj} from $\mathcal{A}$ on behalf of P_i, P_j, respectively. If $\mathcal{A}$ fails to send a value, it is assumed to be 1.
– $\mathcal{S}_{\text{jmp4}}^{P_k}$ receives b_k from $\mathcal{A}$ on behalf of honest P_l. If $\mathcal{A}$ fails to send a value, b_k is assumed to be 0. If $b_k = 1$ and $b_{ki} \vee b_{kj} = 1$, $\mathcal{S}_{\text{jmp4}}^{P_k}$ sets $b_l = 1$ and $\mathsf{ttp} = P_l$, else it sets $b_l = 0$ and $\mathsf{ttp} = \perp$. $\mathcal{S}_{\text{jmp4}}^{P_k}$ sends b_l to $\mathcal{A}$ on behalf of P_l. $\mathcal{S}_{\text{jmp4}}^{P_k}$ invokes $\mathcal{F}_{\text{jmp4}}$ with $(\mathsf{Input}, \perp)$ and (Select, b_l) on behalf of $\mathcal{A}$.

Fig. 44: Simulator $\mathcal{S}_{\text{jmp4}}^{P_k}$ for corrupt receiver P_k

The case for a corrupt receiver, P_l, which is the server outside the computation involved in Π_{jmp4}, is provided in Fig. 45.

Simulator $\mathcal{S}_{\text{jmp4}}^{P_l}$

– $\mathcal{S}_{\text{jmp4}}^{P_l}$ sends $b_i = b_j = b_k = 0$ to $\mathcal{A}$ on behalf of P_i, P_j, P_k, respectively.
– $\mathcal{S}_{\text{jmp4}}^{P_l}$ receives b_l from $\mathcal{A}$ on behalf of P_i, P_j, P_k, and sets $\mathsf{ttp} = \perp$.
– $\mathcal{S}_{\text{jmp4}}^{P_l}$ invokes $\mathcal{F}_{\text{jmp4}}$ with $(\mathsf{Input}, \perp)$ and (Select, b_l) on behalf of $\mathcal{A}$.

Fig. 45: Simulator $\mathcal{S}_{\text{jmp4}}^{P_l}$ for corrupt fourth server P_l

2) Sharing Protocol: Here we give the simulation steps for Π_{sh4}. The case for corrupt P_0 is given in Fig. 46.

Simulator $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_0}$

Preprocessing: $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_0}$ emulates $\mathcal{F}_{\text{setup4}}$ and gives the keys $(k_{01}, k_{02}, k_{03}, k_{012}, k_{013}, k_{023}$ and $k_{\mathcal{P}})$ to $\mathcal{A}$. The values that are commonly held with $\mathcal{A}$ are sampled using the respective keys, while others are sampled randomly.

Online:

– If dealer is P_0, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_0}$ receives β_{v} from $\mathcal{A}$ on behalf of P_1. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_i}$ where P_0 acts as one of the sender for sending β_{v}.
– If dealer is P_1 or P_2, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_0}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$. Steps corresponding to Π_{jmp4} for sending $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to $\mathcal{A}$ are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_k}$ where P_0 acts as the receiver.
– If dealer is P_3, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_0}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$. $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_0}$ sends $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to $\mathcal{A}$ on behalf of P_3. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_j}$ where P_0 acts as one of the sender with P_1, P_2 as the receivers, separately.

Fig. 46: Simulator $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_0}$ for corrupt P_0

The case for corrupt P_1 is given in Fig. 47. The case for a corrupt P_2 is similar.

Simulator $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$

Preprocessing: $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$ emulates $\mathcal{F}_{\text{setup4}}$ and gives the keys $(k_{01}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}$ and $k_{\mathcal{P}})$ to $\mathcal{A}$. The values that are commonly held with $\mathcal{A}$ are sampled using the respective keys, while others are sampled randomly.

Online:

– If dealer is P_1, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$ receives β_{v} from $\mathcal{A}$ on behalf of P_2. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_i}$ where P_1 acts as one of the sender for sending $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0.
– If dealer is P_0 or P_2, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$.
  - If dealer is P_0, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$ sends β_{v} to $\mathcal{A}$ on behalf of P_0. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_j}$ where P_1 acts as one of the sender to send β_{v}.
  - If dealer is P_2, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$ sends β_{v} to $\mathcal{A}$ on behalf of P_2. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_i}$ where P_1 acts as one of the sender to send $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$.
– If dealer is P_3, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_k}$ where P_1 acts as the receiver for receiving $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$.

Fig. 47: Simulator $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_1}$ for corrupt P_1

The case for corrupt P_3 is given in Fig. 48.

Simulator $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_3}$

Preprocessing: $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_3}$ emulates $\mathcal{F}_{\text{setup4}}$ and gives the keys $(k_{03}, k_{13}, k_{23}, k_{013}, k_{023}, k_{123}$ and $k_{\mathcal{P}})$ to $\mathcal{A}$. The values that are commonly held with $\mathcal{A}$ are sampled using the respective keys, while others are sampled randomly.

Online:

– If dealer is P_3, $\mathcal{S}_{\Pi_{\text{sh4}}}^{P_3}$ receives $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ from $\mathcal{A}$ on behalf of P_0. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}_{\Pi_{\text{jmp4}}}^{P_i}$ where P_3 acts as one of the sender with P_1, P_2 as the receivers, separately.
– If dealer is P_0 or P_1 or P_2, steps corresponding to Π_{jmp4} are

simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as the server outside the computation.

Fig. 48: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{sh4}}}$ for corrupt P_3

3) Multiplication Protocol: Here we give the simulation steps for Π_{mult4}. The case for corrupt P_0 is given in Fig. 49.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{mult4}}}$

Preprocessing:

- $\mathcal{S}^{P_0}_{\Pi_{\mathsf{mult4}}}$ samples $[\alpha_{\mathsf{z}}]_1, [\alpha_{\mathsf{z}}]_2, [\Gamma_{\mathsf{xy}}]_1$ using the respective keys with $\mathcal{A}$. $\mathcal{S}^{P_0}_{\Pi_{\mathsf{mult4}}}$ samples $\gamma_{\mathsf{z}}, \psi, \mathsf{r}$ randomly on behalf of the respective honest parties, and computes $[\Gamma_{\mathsf{xy}}]_2$ honestly.
- Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as one of the sender for sending $[\Gamma_{\mathsf{xy}}]_2$.
- $\mathcal{S}^{P_0}_{\Pi_{\mathsf{mult4}}}$ computes $[\chi]_1, [\chi]_2$ honestly. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as the receiver for $[\chi]_1, [\chi]_2$.

Online:

- $\mathcal{S}^{P_0}_{\Pi_{\mathsf{mult4}}}$ computes $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$ honestly. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as one of the sender for sending $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$.
- $\mathcal{S}^{P_0}_{\Pi_{\mathsf{mult4}}}$ computes $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as the receiver for receiving $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 49: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{mult4}}}$ for corrupt P_0

The case for corrupt P_1 is given in Fig. 50. The case for a corrupt P_2 is similar.

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{mult4}}}$

Preprocessing:

- $\mathcal{S}^{P_1}_{\Pi_{\mathsf{mult4}}}$ samples $[\alpha_{\mathsf{z}}]_1, \gamma_{\mathsf{z}}, \psi, \mathsf{r}, [\Gamma_{\mathsf{xy}}]_1$ using the respective keys with $\mathcal{A}$. $\mathcal{S}^{P_1}_{\Pi_{\mathsf{mult4}}}$ samples $[\alpha_{\mathsf{z}}]_2$ randomly on behalf of the respective honest parties.
- Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as the server outside the computation while communicating $[\Gamma_{\mathsf{xy}}]_2$.
- $\mathcal{S}^{P_1}_{\Pi_{\mathsf{mult4}}}$ computes $[\chi]_1$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as one of the sender for $[\chi]_1$.
- Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as the server outside the computation while communicating $[\chi]_2$.

Online:

- $\mathcal{S}^{P_1}_{\Pi_{\mathsf{mult4}}}$ computes $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ and $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$, where P_1 acts as one of the sender for sending $[\beta^\star_{\mathsf{z}}]_1$, and P_1 acts as the receiver for receiving $[\beta^\star_{\mathsf{z}}]_2$, respectively.
- $\mathcal{S}^{P_1}_{\Pi_{\mathsf{mult4}}}$ computes $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as one of the sender for sending $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 50: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{mult4}}}$ for corrupt P_1

The case for corrupt P_3 is given in Fig. 51.

Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{mult4}}}$

Preprocessing:

- $\mathcal{S}^{P_3}_{\Pi_{\mathsf{mult4}}}$ samples $[\alpha_{\mathsf{z}}]_1, [\alpha_{\mathsf{z}}]_2, \gamma_{\mathsf{z}}, \psi, \mathsf{r}, [\Gamma_{\mathsf{xy}}]_1$ using the respective keys with $\mathcal{A}$. $\mathcal{S}^{P_3}_{\Pi_{\mathsf{mult4}}}$ computes $[\Gamma_{\mathsf{xy}}]_2$ honestly.
- Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as one of the sender for sending $[\Gamma_{\mathsf{xy}}]_2$.
- $\mathcal{S}^{P_3}_{\Pi_{\mathsf{mult4}}}$ computes $[\chi]_1, [\chi]_2$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as one of the sender for sending $[\chi]_1$ and $[\chi]_2$.

Online:

- Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as the server outside the computation involving $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$ and $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 51: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{mult4}}}$ for corrupt P_3

4) Reconstruction Protocol: Here we give the simulation steps for Π_{rec4}. The case for corrupt P_0 is given in Fig. 52. The cases for corrupt P_1, P_2, P_3 are similar.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{rec4}}}$

- $\mathcal{S}^{P_0}_{\Pi_{\mathsf{rec4}}}$ sends γ_{v} to $\mathcal{A}$ on behalf of P_1, P_2, and $\mathsf{H}(\gamma_{\mathsf{v}})$ on behalf of P_3, respectively.
- $\mathcal{S}^{P_0}_{\Pi_{\mathsf{rec4}}}$ receives $\mathsf{H}([\alpha_{\mathsf{v}}]_1), \mathsf{H}([\alpha_{\mathsf{v}}]_2), \beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ from $\mathcal{A}$ on behalf of P_2, P_1, P_3, respectively.

Fig. 52: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{rec4}}}$ for corrupt P_0

5) Joint Sharing Protocol: Here we give the simulation steps for Π_{jsh4}. The case for corrupt P_0 is given in Fig. 53.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{jsh4}}}$

Preprocessing:

- $\mathcal{S}^{P_0}_{\Pi_{\mathsf{jsh4}}}$ has knowledge of α_{v} and γ_{v}, which it obtains while emulating $\mathcal{F}_{\mathsf{setup4}}$. The common values shared with the $\mathcal{A}$ are sampled using the appropriate shared keys, while other values are sampled at random.

Online:

- If dealers are (P_0, P_1): $\mathcal{S}^{P_0}_{\Pi_{\mathsf{jsh4}}}$ computes β_{v} using v. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as one of the sender for β_{v}.
- If dealers are (P_0, P_2) or (P_0, P_3): Analogous to the above case.
- If dealers are (P_1, P_2): $\mathcal{S}^{P_0}_{\Pi_{\mathsf{jsh4}}}$ sets $\mathsf{v} = 0$ and $\beta_{\mathsf{v}} = [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as the receiver for $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$.
- If dealers are (P_3, P_1): $\mathcal{S}^{P_0}_{\Pi_{\mathsf{jsh4}}}$ sets $\mathsf{v} = 0$ and $\beta_{\mathsf{v}} = [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as the server outside the computation for β_{v}, and according to $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as the receiver for $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$.
- If dealers are (P_3, P_2): Analogous to the above case.

Fig. 53: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{jsh4}}}$ for corrupt P_0

The case for corrupt P_1 is given in Fig. 54. The case for corrupt P_2 is similar.

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{jsh4}}}$

Preprocessing:

– $\mathcal{S}^{P_1}_{\Pi_{\mathsf{jsh4}}}$ has knowledge of α-values and γ corresponding to v which it obtains while emulating $\mathcal{F}_{\mathsf{setup4}}$. The common values shared with the $\mathcal{A}$ are sampled using the appropriate shared keys, while other values are sampled at random.

Online:

– If dealers are (P_0, P_1): $\mathcal{S}^{P_1}_{\Pi_{\mathsf{jsh4}}}$ computes β_{v} using v. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as one of the sender for β_{v}.
– If dealers are (P_1, P_2): Analogous to the previous case, except that now $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ is sent instead of β_{v}.
– If dealers are (P_3, P_1): $\mathcal{S}^{P_1}_{\Pi_{\mathsf{jsh4}}}$ computes β_{v} and $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as one of the sender for β_{v}, $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$.
– If dealers are (P_0, P_2) or (P_0, P_3) or (P_3, P_2): $\mathcal{S}^{P_1}_{\Pi_{\mathsf{jsh4}}}$ sets $\mathsf{v} = 0$ and $\beta_{\mathsf{v}} = [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as the receiver for β_{v}.

Fig. 54: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{jsh4}}}$ for corrupt P_1

The case for corrupt P_3 is given in Fig. 55.

Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{jsh4}}}$

Preprocessing:

– $\mathcal{S}^{P_3}_{\Pi_{\mathsf{jsh4}}}$ has knowledge of α-values and γ corresponding to v which it obtains while emulating $\mathcal{F}_{\mathsf{setup4}}$. The common values shared with the $\mathcal{A}$ are sampled using the appropriate shared keys, while other values are sampled at random.

Online:

– If dealers are (P_1, P_2): $\mathcal{S}^{P_3}_{\Pi_{\mathsf{jsh4}}}$ sets $\mathsf{v} = 0$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as the server outside the computation for $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$.
– If dealers are (P_0, P_1) or (P_0, P_2): Analogous to the above case.
– If dealers are (P_0, P_3): $\mathcal{S}^{P_3}_{\Pi_{\mathsf{jsh4}}}$ computes β_{v} using v. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as one of the sender for sending β_{v}.
– If dealers are (P_3, P_1): $\mathcal{S}^{P_3}_{\Pi_{\mathsf{jsh4}}}$ computes β_{v} and $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as one of the sender for sending $\beta_{\mathsf{v}}, \beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$.
– If dealers are (P_3, P_2): Analogous to the above case.

Fig. 55: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{jsh4}}}$ for corrupt P_3

6) Dot Product Protocol: Here we give the simulation steps for Π_{dotp4}. The case for corrupt P_0 is given in Fig. 56.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{dotp4}}}$

Preprocessing:

– $\mathcal{S}^{P_0}_{\Pi_{\mathsf{dotp4}}}$ samples $[\alpha_{\mathsf{z}}]_1, [\alpha_{\mathsf{z}}]_2, [\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_1$ using the respective keys with $\mathcal{A}$. $\mathcal{S}^{P_0}_{\Pi_{\mathsf{dotp4}}}$ samples $\gamma_{\mathsf{z}}, \psi, \mathsf{r}$ randomly on behalf of the respective honest parties, and computes $[\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_2$ honestly.
– Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as one of the sender for $[\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_2$.
– $\mathcal{S}^{P_0}_{\Pi_{\mathsf{dotp4}}}$ computes χ_1, χ_2 honestly. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as the receiver for χ_1 and χ_2.

Online:

– $\mathcal{S}^{P_0}_{\Pi_{\mathsf{dotp4}}}$ computes $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$ honestly. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as one of the sender for $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$.
– $\mathcal{S}^{P_0}_{\Pi_{\mathsf{dotp4}}}$ computes $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$ where P_0 acts as the receiver for $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 56: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{dotp4}}}$ for corrupt P_0

The case for corrupt P_1 is given in Fig. 57. The case for corrupt P_2 is similar.

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{dotp4}}}$

Preprocessing:

– $\mathcal{S}^{P_1}_{\Pi_{\mathsf{dotp4}}}$ samples $[\alpha_{\mathsf{z}}]_1, \gamma_{\mathsf{z}}, \psi, \mathsf{r}, [\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_1$ using the respective keys with $\mathcal{A}$. $\mathcal{S}^{P_1}_{\Pi_{\mathsf{dotp4}}}$ samples $[\alpha_{\mathsf{z}}]_2$ randomly on behalf of the respective honest parties.
– Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as the server outside the computation for $[\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_2$.
– $\mathcal{S}^{P_1}_{\Pi_{\mathsf{dotp4}}}$ computes χ_1. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as one of the sender for χ_1.
– Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as the server outside the computation for χ_2.

Online:

– $\mathcal{S}^{P_1}_{\Pi_{\mathsf{dotp4}}}$ computes $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ and $\mathcal{S}^{P_k}_{\Pi_{\mathsf{jmp4}}}$, where P_1 acts as one of the sender for $[\beta^\star_{\mathsf{z}}]_1$, and P_1 acts as the receiver for $[\beta^\star_{\mathsf{z}}]_2$.
– $\mathcal{S}^{P_1}_{\Pi_{\mathsf{dotp4}}}$ computes $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_i}_{\Pi_{\mathsf{jmp4}}}$ where P_1 acts as one of the sender for $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 57: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{dotp4}}}$ for corrupt P_1

The case for corrupt P_3 is given in Fig. 58.

Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{dotp4}}}$

Preprocessing:

– $\mathcal{S}^{P_3}_{\Pi_{\mathsf{dotp4}}}$ samples $[\alpha_{\mathsf{z}}]_1, [\alpha_{\mathsf{z}}]_2, \gamma_{\mathsf{z}}, \psi, \mathsf{r}, [\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_1$ using the respective keys with $\mathcal{A}$. $\mathcal{S}^{P_3}_{\Pi_{\mathsf{dotp4}}}$ computes $[\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]$ honestly.
– Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as one of the sender for $[\Gamma_{\vec{\mathbf{x}}\odot\vec{\mathbf{y}}}]_2$.
– $\mathcal{S}^{P_3}_{\Pi_{\mathsf{dotp4}}}$ computes χ_1, χ_2. Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_j}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as one of the sender for χ_1, χ_2.

Online:

– Steps corresponding to Π_{jmp4} are simulated according to $\mathcal{S}^{P_l}_{\Pi_{\mathsf{jmp4}}}$ where P_3 acts as the server outside the computation for $[\beta^\star_{\mathsf{z}}]_1, [\beta^\star_{\mathsf{z}}]_2, \beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$.

Fig. 58: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{dotp4}}}$ for corrupt P_3

7) Truncation Pair Generation: Here we give the simulation steps for Π_{trgen4}. The case for corrupt P_0 is given in Fig. 59. The case for corrupt P_3 is similar.

Simulator $\mathcal{S}_{\Pi_{\mathsf{trgen4}}}^{P_0}$

– $\mathcal{S}_{\Pi_{\mathsf{trgen4}}}^{P_0}$ samples R_1, R_2 using the respective keys with $\mathcal{A}$.

– Steps corresponding to Π_{jsh4} are simulated according to $\mathcal{S}_{\Pi_{\mathsf{jsh4}}}^{P_0}$ (Fig. 53).

Fig. 59: Simulator $\mathcal{S}_{\Pi_{\mathsf{trgen4}}}^{P_0}$ for corrupt P_0

The case for corrupt P_1 is given in Fig. 60. The case for corrupt P_2 is similar.

Simulator $\mathcal{S}_{\Pi_{\mathsf{trgen4}}}^{P_1}$

– $\mathcal{S}_{\Pi_{\mathsf{trgen4}}}^{P_1}$ samples R_1 using the respective keys with $\mathcal{A}$, and samples R_2 randomly.

– Steps corresponding to Π_{jsh4} are simulated according to $\mathcal{S}_{\Pi_{\mathsf{jsh4}}}^{P_1}$ (Fig. 54).

Fig. 60: Simulator $\mathcal{S}_{\Pi_{\mathsf{trgen4}}}^{P_1}$ for corrupt P_1

Observe from the simulation steps, that the view of $\mathcal{A}$ in the real world and the ideal world is indistinguishable.