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

Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh,
Indian Institute of Science, Bangalore

https://www.usenix.org/conference/usenixsecurity21/presentation/koti

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

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

Nishat Koti
Indian Institute of Science

Mahak Pancholi
Indian Institute of Science

Arpita Patra
Indian Institute of Science

Ajith Suresh
Indian Institute of Science

Abstract

Performing machine learning (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 the 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 (and is strictly better in some cases than) the best-known 3PC framework BLAZE (Patra et al. NDSS'20), which only achieves fairness. 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 our framework's practical relevance by benchmarking popular ML algorithms such as Logistic Regression and deep Neural Networks such as VGG16 and LeNet, both over a 64-bit ring in a WAN setting. For deep NN, our results testify to our claims that we provide improved security guarantee while incurring no additional overhead for 3PC and obtaining 2× improvement for 4PC.

1 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. This is addressed by the Secure Outsourced Computation (SOC) paradigm, and thus is an apt fit for the need of the moment. Many recent works [11, 14, 15, 41, 43, 45, 48, 50, 55] 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[1] 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 [5, 11, 14, 15, 19, 24–26, 33–35, 48] cast their protocols into the preprocessing model wherein the input-independent (yet function-dependent) phase computationally heavy tasks are computed 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 [6, 20, 23, 25, 27] 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

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

customizations that they permit, for instance, a cheaper realisation of multiplication through custom-made secret sharing schemes [3, 4, 11, 14, 15, 48].

We now motivate the need for robustness aka guaranteed output delivery (GOD) over fairness[2], or even abort security[3], in the domain of PPML. Robustness provides the guarantee of output delivery to all protocol participants, no matter how the adversary misbehaves. Robustness is 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 lead to the protocol aborting, resulting in the user not obtaining the desired output. This leads to denial of service and heavy economic losses for the service provider. For data providers, as more training data leads to more accurate models, collaboratively building a model enables them to provide better ML services, and consequently, attract more clients. A robust framework encourages active involvement from multiple data providers. Hence, for the seamless adoption of PPML solutions in the real world, the robustness of the protocol is of utmost importance. Several works [14, 15, 43, 48, 55] realizing PPML via MPC settle for weaker guarantees such as abort and fairness. Achieving the strongest notion of GOD without degrading performance is an interesting goal which forms the core focus of this work. The hall-mark result of [17] suggests that an honest-majority amongst the servers is necessary to achieve robustness. Consequent to the discussion above, we focus on the honest-majority setting with a small set of parties, especially 3 and 4 parties, both of which have drawn enormous attention recently [3, 4, 8, 9, 11, 13–15, 30, 44, 46, 48]. The $3/4$-party setting enables simpler, more efficient, and customized secure protocols compared to the n-party setting. Real-world MPC applications and frameworks such as the Danish sugar beet auction [7] and Sharemind [6], have demonstrated the practicality of 3-party protocols. Additionally, in an outsourced setting, $3/4$PC is useful and relevant even when there are more parties. Specifically, here the entire computation is offloaded to $3/4$ hired servers, after initial sharing of inputs by the parties amongst the servers. This is precisely what we (and some existing papers [11, 42, 48]) contemplate as the setting for providing ML-as-a-service. Our protocols work over rings, are cast in the preprocessing paradigm, and achieve GOD.

Related Work We restrict the relevant work to 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 [12, 13, 47], and high throughput protocols [1, 3, 4, 6, 9, 14, 16, 29, 30, 46, 48]. [4, 14] provide efficient semi-honest protocols wherein ASTRA [14] improved upon [4] 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 [3]. Later, a maliciously-secure 3PC protocol based on distributed zero-knowledge techniques was proposed by Boneh et al. [8] providing abort security. Further, building on [8] and enhancing the security to GOD, Boyle et al. [9] 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 [48] 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 [9].

In the regime of 4PC, Gordon et al. [31] presented protocols achieving abort security and GOD. However, [31] relied on expensive public-key primitives and broadcast channels to achieve GOD. Trident [15] improved over the abort protocol of [31], providing a fast online phase achieving security with fairness, and presented a framework for mixed world computations [27]. A robust 4PC protocol was provided in FLASH [11], which requires communicating 6 ring elements, each, in the preprocessing and online phases.

In many recent works [9, 11, 13], including this work, GOD is achieved by having an *honest* party, identified as a trusted third party (TTP), compute the function on the 'clear' inputs of all the parties (in the case of a misbehaviour). The classical security definition allows this leakage of inputs since the selected TTP is honest. There has been a recent study on the additional requirement of hiding the inputs from a quorum of honest parties (treating them as semi-honest), termed as Friends-and-Foes (FaF) security notion [2]. This is a stronger security goal than the classical one. Recently, the work of [22] attempts to offer a variant of GOD, referred to as *private robustness* in 4PC setting. As per the authors, in a private robust protocol, no single honest party learns the other honest parties' input. We want to point out that [22] does not achieve FaF security notion [2], since an adversary can reveal its view to an honest party, making it obtain the inputs of the other honest parties. [4]

In the PPML domain, MPC has been used for various ML algorithms such as Decision Trees [40], Linear Regression [28, 51], k-means clustering [10, 32], SVM Classification [54, 57], Logistic Regression [53]. In the 3PC SOC setting, the works of ABY3 [43] and SecureNN [55], provide security with abort. This was followed by ASTRA [14], which

[2] This ensures either all parties or none learn the output.

[3] This may allow the corrupt parties *alone* to learn the output.

[4] Lastly, but importantly, the precise security notion achieved in [22] is unclear due to the lack of formal details.

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 [11] which proposed efficient building blocks for ML such as dot product, truncation, MSB extraction, and bit conversion. The works of [11,14,15,43,45,48,55] 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 3PC and 4PC settings. We now present our contributions and compare them with these works.

1.1 Our Contributions

We propose, **SWIFT** [36], a robust maliciously-secure framework for PPML in the SOC setting, 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 obtain a fast online phase. As mentioned earlier, the input-independent (yet function-dependent) computations will be performed in the preprocessing phase.

To the best of our knowledge, SWIFT is the first robust and efficient PPML framework in the 3PC setting and is as fast as (and is strictly better in some cases than) the best known *fair* 3PC framework BLAZE [48]. We extend our 3PC framework for 4 servers. In this regime, SWIFT is as fast as the best known *fair* 4PC framework Trident [15] and twice faster than best known *robust* 4PC framework FLASH [11]. We detail our contributions next.

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. We use secret-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 in the SOC setting. A highlight of our 3PC framework, which, to the best of our knowledge is achieved for the first time, is a robust dot-product protocol whose (amortized) communication cost is independent of the vector size, which we obtain by extending the techniques of [8,9]. The performance comparison in terms of concrete cost for communication and rounds, for PPML primitives in both 3PC and 4PC setting, appear in Table 1. As claimed, SWIFT is on par with BLAZE for most of the primitives (while improving security from fair to GOD) and is strictly better than BLAZE in case of dot product and dot product with truncation. For 4PC, SWIFT is on par with Trident in most cases (and is slightly better for dot product with truncation and bit injection), while it is doubly faster than FLASH. Since BLAZE outperforms the 3PC abort framework of ABY3 [43] while Trident outperforms the known 4PC with abort [31], SWIFT attains robustness with better cost than the know protocols with weaker guarantees. No performance loss coupled with the strongest security guarantee makes our robust framework an opt choice for practical applications including PPML.

Applications and Benchmarking We demonstrate the practicality of our protocols by benchmarking PPML, particularly, Logistic Regression (training and inference) and popular Neural Networks (inference) such as [45], LeNet [38] and VGG16 [52] having millions of parameters. The NN training requires mixed-world conversions [15,27,43], 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 [11, 14, 15, 48]).

Comparisons and Differences with Prior Works To begin with, 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 is identified. jmp is extremely efficient as for a message of ℓ elements it only incurs the minimal communication cost 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. All our primitives, either for a general 3PC or a PPML task, achieve GOD relying on jmp.

Second, instead of using the multiplication of [9] (which has the same overall communication cost as that of our online phase), we build a new protocol. This is because the former involves distributed zero-knowledge protocols. The cost of this heavy machinery gets amortized only for large circuits having millions of gates, which is very unlikely for inference and moderately heavy training tasks in PPML. As in BLAZE [48], we follow a similar structure for our multiplication protocol but differ considerably in techniques as our goal is to obtain GOD. Our approach is to manipulate and transform some of the protocol steps so that two other servers can locally compute the information required by a server in a round. However, this transformation is not straight forward since BLAZE was constructed with a focus towards providing only fairness (details appear in §3). The multiplication protocol forms a technical basis for our dot product protocol and

Building Blocks	3PC					4PC				
	Ref.	Pre.	Online		Security	Ref.	Pre.	Online		Security
		Comm. (ℓ)	Rounds	Comm. (ℓ)			Comm. (ℓ)	Rounds	Comm. (ℓ)	
Multiplication	[8]	1	1	2	Abort					
	[9]	-	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	**3**	**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	**15**	**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.

Table 1: 3PC and 4PC: Comparison of SWIFT with its closest competitors in terms of Communication and Round Complexity

other PPML building blocks. We emphasise again that the (amortized) cost of our dot product protocol is independent of the vector size.

Third, extending to 4PC brings several performance improvements over 3PC. Most prominent of all is a conceptually simple jmp instantiation, which forgoes the broadcast channel while retaining the same communication cost; and a dot product with cost independent of vector size sans the 3PC amortization technique.

Fourth, we provide robust protocols for 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 communication efficiency together makes these tasks slightly non-trivial. As a highlight, we introduce a super-fast online phase for reconstruction protocol, which gives 4× improvement in terms of rounds (apart from improvement in communication) compared to BLAZE. Although we aim for GOD, we ensure that an end-user is never part of a broadcast which is relatively expensive than *atomic* point-to-point communication.

Organisation of the paper. The rest of the paper is organized as follows. §2 describes the system model, preliminaries and notations used. §3 and §4 detail our constructs in the 3PC and respectively 4PC setting. These are followed by the Applications and benchmarking are detailed in §5. Additional preliminaries and ideal functionalities are elaborated in §A, §B and §C. Further details on the cost analysis and security are deferred to the full version of the paper [36].

2 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 [18]. For ML training, several data-owners who wish to jointly train a model, secret share (using the sharing semantics that will appear later) 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 a 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 [11, 14, 15, 43, 45, 48].

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 obtaining 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) [11, 14, 15, 43, 45, 48] 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 $\ell = 64, x = 13$, leaving $\ell - x - 1$ bits for the integer part.

The servers use a one-time key setup, modelled as a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 6), to establish pre-shared random keys for pseudo-random functions (PRF) between them. A similar setup is used in [3, 9, 14, 30, 43, 48, 50] for three server case and in [11, 15] for four 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 are deferred to §A.

Notation 2.1. 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 bit in the i^{th} position of an ℓ-bit value v is denoted by $\mathsf{v}[i]$.

Notation 2.2. 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.

3 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.

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 $([\alpha_{\mathsf{v}}]_s, \beta_{\mathsf{v}}, \gamma_{\mathsf{v}})$.

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 $\mathbb{Z}_{2^1}$.

Linearity of the Secret Sharing Scheme Given $[\cdot]$-shares of $\mathsf{v}_1, \mathsf{v}_2$, and public constants c_1, c_2, servers can locally compute $[\cdot]$-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 linearity property is satisfied by $\langle\cdot\rangle$ and $[\![\cdot]\!]$ sharings.

3.1 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. 8) and the protocol for the same appears in Fig. 1. Next, we give an overview.

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 hash is done once and for all from P_j to P_k. In the simplest case, P_k receives a consistent (value, hash) pair, and the protocol terminates. In all other cases, a trusted third party (TTP) is identified as follows without having to communicate v again. Importantly, the following part can be run once and for all instances of Π_{jmp} with P_i, P_j, P_k in same roles, invoked in the final 3PC protocol. Consequently, the cost due to this part vanishes in an amortized sense, yielding a rate-1 construction.

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

Each server P_s for $s \in \{i,j,k\}$ initializes bit $\mathsf{b}_s = 0$.

Send Phase: P_i sends v to P_k.

Verify Phase: P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.

– P_k broadcasts "`(accuse,`P_i`)`", if P_i is silent and $\mathsf{TTP} = P_j$. Analogously for P_j. If P_k accuses both P_i, P_j, then $\mathsf{TTP} = P_i$. 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 $\mathsf{TTP} = P_j$. Analogously for P_j. If both P_i, P_j accuse P_k, then $\mathsf{TTP} = P_i$. 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 $\mathsf{TTP} = P_k$. 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 $\mathsf{TTP} = P_k$. Otherwise,

- If $\mathsf{H}_i \neq \mathsf{H}_j$: $\mathsf{TTP} = P_k$.
- Else if $\mathsf{H}_i \neq \mathsf{H}_k$: $\mathsf{TTP} = P_j$.
- Else if $\mathsf{H}_i = \mathsf{H}_j = \mathsf{H}_k$: $\mathsf{TTP} = P_i$.

Figure 1: 3PC: Joint Message Passing Protocol

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 an 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 server broadcasts a hash of its value when its inconsistency bit is on;[5] 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.

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

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

Notation 3.1. 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 jmp protocol needs to be viewed as 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.

3.2 3PC Protocols

We now describe the protocols for 3 parties/servers and refer readers to the full version [36] for the communication analysis and security proofs of our protocols.

Sharing Protocol Protocol Π_{sh} 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}. We defer formal details of Π_{sh} to the full version [36].

Joint Sharing Protocol Protocol Π_{jsh} (Fig. 9) 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} 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.

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

Multiplication Protocol Protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$ (Fig. 2) 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 [48] 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,

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

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, $[\cdot]$-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 $\mathsf{jmp\text{-}send}$ $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ 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 $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0 (in addition to P_1's communication of $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ 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 $\beta_x + \gamma_x$, $\beta_y + \gamma_y$, α_x, α_y, α_z and Γ_{xy}, P_0 computes:

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

Now if χ can be made available to P_0, it can send $\beta_z^\star + \chi$ to P_1 and P_2 who using the knowledge of β_x, β_y, can verify the correctness of β_z by computing $\beta_z - \beta_x\beta_y$ and checking against the value $\beta_z^\star + \chi$ received from P_0. However, disclosing χ on clear to P_0 will cause a privacy issue when P_0 is corrupt, because one degree of freedom on the pair (γ_x, γ_y) is lost and the same impact percolates down to (β_x, β_y) and further to the actual values (v_x, v_y) on the wires x, y. This is resolved through a random value $\psi \in \mathbb{Z}_{2^\ell}$, sampled together by P_1 and P_2. Now, χ and $\beta_z^\star$ are set to $\gamma_x\alpha_y + \gamma_y\alpha_x - \Gamma_{xy} + \psi$, $(\beta_z - \beta_x\beta_y + \psi) - \chi$, respectively and the check by P_1, P_2 involves computing $\beta_z - \beta_x\beta_y + \psi$. 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). Tying the ends together, they identify the precise shared multiplication triple and map its components to χ and Γ_{xy} so that these values are correct by virtue of the correctness of the product relation. This reduces ensuring the correctness of these values to doing a single multiplication of two values in the preprocessing phase.

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 2):

$$(d_0{=}[\alpha_x]_2, d_1{=}[\alpha_x]_1, d_2{=}\gamma_x), \quad (e_0{=}[\alpha_y]_2, e_1{=}[\alpha_y]_1, e_2{=}\gamma_y)$$

– 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.

Figure 2: 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 $\{P_0, P_1\}, \{P_0, P_2\}$, 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 server which computes $\beta_z^\star$. Due to the presence of one honest server 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. The outcome of our approach is a win-win situation i.e. either success or TTP selection. Our approach of computing β_z from $\beta_z^\star$ is a departure from BLAZE, where the latter suggests computing β_z from the exchange P_1, P_2's respective share of β_z (as in the semi-honest construction) and use $\beta_z^\star$ for verification. 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 2, (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. 10) and finally (c) extract out the required preprocessing data *locally* as in Eq. 2.

$\langle v\rangle$	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 2: The $\langle\cdot\rangle$-sharing of values d and e

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

We switch to $\langle\cdot\rangle$-sharing in this part to be able to use the best robust multiplication protocol of [9] that supports this form of secret sharing and requires communication of just 3 elements. Fortunately, the switch does not cost anything, as both the step

(a) and (c) (as above) involve local computation and the cost simply reduces to a single run of a multiplication protocol. 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.

To leverage amortization, the *send* phase of jmp-send alone is executed on the fly and *verify* is performed once for multiple instances of jmp-send. Further, observe that P_1, P_2 possess the required shares in the online phase to compute the entire circuit. So, P_0 can come in only during *verify* of jmp-send towards P_1, P_2, which can be deferred towards the end. Hence, the jmp-send of $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0 (enabling computation of the verification information) can be performed once, towards the end, thereby requiring a single round for sending $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0 for multiple instances. Following this, the *verify* of jmp-send towards P_0 is performed first, followed by performing the *verify* of jmp-send towards P_1, P_2 in parallel.

We note that to facilitate a fast online phase for multiplication, our preprocessing phase leverages a robust multiplication protocol [9] 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 the communication cost, which can be obtained by instantiating the black-box with an efficient, robust protocol coupled with the fast online phase.

Reconstruction Protocol Protocol Π_{rec} 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. We defer the details to the full version [36].

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. A similar approach will be taken for 4PC and each PPML task, 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. 6) 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}. 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. 2). Finally, servers run the online steps of Π_{rec} on the output wires to reconstruct the function output. To leverage amortization, only *send* phases of all the jmp are run on the flow. At the end of preprocessing, 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* 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.

On the security of our framework: We emphasize that we follow the standard traditional (real-world / ideal-world based) security definition of MPC, according to which, in the 4-party setting with 1 corruption, exactly 1 party is assumed to be corrupt, and rest are *honest*. As per this definition, disclosing the honest parties's inputs to a selected *honest* party is *not* a breach of security. Indeed, in our framework, the data sharing and the computation on the shared data is done in a way that any malicious behaviour leads to establishment of a TTP who is enabled to receive all the inputs and compute the output on the clear. There has been a recent study on the additional requirement of hiding the inputs from a quorum of honest parties (treating them as semi-honest), termed as Friends-and-Foes (FaF) security notion [2]. This is a stronger security goal than the standard one and it has been shown that one cannot obtain FaF-secure robust 3PC. We leave FaF-secure 4PC for future exploration.

3.3 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 defer the communication analysis of our protocols and security proofs to the full version [36]. We begin by providing details of input sharing and reconstruction in the SOC setting.

Input Sharing and Output Reconstruction in the SOC Setting Protocol $\Pi^{\mathsf{SOC}}_{\mathsf{sh}}$ (Fig. 3) 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. 3) 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 the clear to the TTP, who computes the function output and sends it back to U.

MSB Extraction, Bit to Arithmetic Conversion and Bit Injection Protocols We provide a high-level overview of three protocols that involve working over arithmetic and boolean rings in a mixed fashion and are used in PPML primitives. The *bit extraction* protocol, Π_{bitext} allows servers to compute *boolean* sharing of the most significant bit (msb) of a value v from 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 the 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 2.2). Lastly, *Bit Injection* protocol, Π_{BitInj}, allows servers to compute the arithmetic sharing $[\![\mathsf{bv}]\!]$ from boolean sharing of a bit b ($[\![\mathsf{b}]\!]^{\mathbf{B}}$) and arithmetic sharing of v ($[\![\mathsf{v}]\!]$).

The core techniques used in these protocols follow from BLAZE [48], where multiplication calls are replaced with our new Π_{mult}, and private communications are replaced with jmp-send to ensure a successful run or TTP selection. These PPML building-blocks can be understood without details of the constructs and hence, are deferred to full version [36].

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 server 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 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 server sends $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ and commitments on $[\alpha_{\mathsf{v}}]_1, [\alpha_{\mathsf{v}}]_2$ and γ_{v} to U, who accepts the values forming majority.
- P_0, P_i for $i \in \{1,2\}$ open $[\alpha_{\mathsf{v}}]_i$ to U, while P_1, P_2 open γ_{v} to U.
- U accepts the consistent opening and computes $\mathsf{v} = (\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}) - [\alpha_{\mathsf{v}}]_1 - [\alpha_{\mathsf{v}}]_2 - \gamma_{\mathsf{v}}$.

Figure 3: 3PC: Input Sharing and Output Reconstruction

Dot Product Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol Π_{dotp} allows servers to generate $[\![\cdot]\!]$-sharing of $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ robustly. $[\![\cdot]\!]$-sharing of a vector $\vec{\mathbf{x}}$ of size n, means that each element $\mathsf{x}_i \in \mathbb{Z}_{2^{\ell}}$ of $\vec{\mathbf{x}}$, 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. Analogous to our multiplication protocol, our dot product offloads one call to a robust dot product protocol to the preprocessing. By extending techniques of [8, 9], we give an instantiation for the dot product protocol used in our preprocessing whose (amortized) communication cost is constant, thereby making our preprocessing cost also *independent* of n.

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 $i \in [n]$, 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} (ideal functionality for the same appears in Fig. 11), 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 $[\chi]_2$.

A trivial way to instantiate Π_{dotpPre} is to treat a dot product operation as n multiplications. However, this results in a communication cost that is linearly dependent on the feature size. Instead, we instantiate Π_{dotpPre} by a semi-honest dot product protocol followed by a verification phase to check the correctness. For the verification phase, we extend the techniques of [8, 9] to provide support for verification of dot product tuples. Setting the verification phase parameters appropriately gives a Π_{dotpPre} whose (amortized) communication cost is independent of the feature size.

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 [11, 14, 43, 45, 48] that re-adjusts the shares after multiplication so that FPA semantics are maintained. As shown in SecureML [45], the method of truncation would result in loss of information on the least significant bits and affect the accuracy by a very minimal amount.

For truncation, servers execute Π_{trgen} (Fig. 12) to generate $([\mathsf{r}], [\![\mathsf{r}^d]\!])$-pair, where r is a random ring element, and r^d is the truncated value of r, i.e the value r right-shifted by d bit positions. Recall that d denotes 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 as v^d, is computed as $\mathsf{v}^d = (\mathsf{v} - \mathsf{r})^d + \mathsf{r}^d$. The correctness and accuracy of this method was shown in ABY3 [43].

Protocol Π_{trgen} is inspired from [15, 43] and proceeds as follows to generate $([\mathsf{r}], [\![\mathsf{r}^d]\!])$. Analogous to the approach of ABY3 [43], servers generate a boolean sharing of an ℓ-bit value $\mathsf{r} = \mathsf{r}_1 \oplus \mathsf{r}_2$, non-interactively. Each server truncates its share of r locally to obtain a boolean sharing of r^d by removing the lower d bits. To obtain the arithmetic shares of $(\mathsf{r}, \mathsf{r}^d)$ from their boolean sharing, we do not, however, rely on the approach of ABY3 as it requires more rounds. Instead, we implicitly perform a *boolean to arithmetic conversion*, as was proposed in Trident [15], to obtain the arithmetic shares of $(\mathsf{r}, \mathsf{r}^d)$. This entails performing two dot product operations and constitutes the cost for Π_{trgen}. We defer details to §B.

Dot Product with Truncation Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol Π_{dotpt} allows servers to generate $[\![\mathsf{z}^d]\!]$, where z^d denotes the truncated value of $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$. A naive way is to compute the dot product using Π_{dotp}, followed by performing 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. 12) and the preprocessing corresponding to Π_{dotp}. In the online phase, servers enable P_1, P_2 to obtain $\mathsf{z}^\star - \mathsf{r}$ instead of $\beta^\star_{\mathsf{z}}$, where $\mathsf{z}^\star = \beta^\star_{\mathsf{z}} - \alpha_{\mathsf{z}}$. Using $\mathsf{z}^\star - \mathsf{r}$, 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 $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z} - \mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$. Formal details are deferred to the full version [36].

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}. In case an arithmetic sharing is desired, servers can apply Π_{bit2A} protocol on the outcome of Π_{bitext} protocol.

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 bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. Here $\overline{\mathsf{b}}$ denotes the complement of b. Given $[\![\mathsf{v}]\!]$, servers execute Π_{bitext} on $[\![\mathsf{v}]\!]$ to generate $[\![\mathsf{b}]\!]^{\mathbf{B}}$. $[\![\cdot]\!]^{\mathbf{B}}$-sharing of $\overline{\mathsf{b}}$ is locally computed by setting $\beta_{\overline{\mathsf{b}}} = 1 \oplus \beta_{\mathsf{b}}$. Servers 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 [14, 43, 45]. 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 in ReLU, and hence, we skip the details.

Maxpool, Convolution and Matrix Multiplication The goal of maxpool is to find the maximum value in a vector $\vec{\mathbf{x}}$ of m values. Maximum between two elements x_i, x_j can be computed by applying secure comparison, which returns a binary sharing of a bit b such that $\mathsf{b} = 0$ if $\mathsf{x}_i > \mathsf{x}_j$, or 1, otherwise, followed by computing $(\mathsf{b})^{\mathbf{B}}(\mathsf{x}_j - \mathsf{x}_i) + \mathsf{x}_i$, which can be performed using bit injection. To find the maximum value in vector $\vec{\mathbf{x}}$, the servers first group the values in $\vec{\mathbf{x}}$ into pairs and securely compare each pair to obtain the maximum of the two. This results in a vector of size $m/2$. This process is repeated for $\mathrm{O}(\log m)$ rounds to obtain the maximum value in the entire vector. Convolutions, which form another important building block in PPML tasks, can be cast into matrix multiplication. Our protocol to compute a matrix of dimension $p \times r$ after multiplication requires only $p \times r$ multiplications. The details appear in the full version of the paper [36].

4 Robust 4PC

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 (used in the preprocessing) altogether. Third, we achieve a dot product protocol with communication cost independent of the size of the vector, completely eliminating the complex machinery required as in the 3PC case. At the heart of our 4PC constructions lies an efficient 4-party jmp primitive, denoted as $\mathsf{jmp4}$, that allows two servers to send a common value to a third server robustly. While we provide details for the protocols that vary significantly from their 3PC counterpart in this section, the details for other protocols along with the communication analysis and security proofs, are deferred to the full version [36].

Secret Sharing Semantics For a value v, the shares for P_0, P_1 and P_2 remain the same as that for 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 $\mathsf{v} = \beta_{\mathsf{v}} - [\alpha_{\mathsf{v}}]_1 - [\alpha_{\mathsf{v}}]_2$.

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. 1) in spirit but is significantly optimized and free from broadcast calls. Similar to the 3PC counterpart, each server 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 to all the servers, who exchange it among themselves and agree on the bit that forms majority (1 indicates the presence of inconsistency, and 0 indicates consistency). The presence of an honest majority among P_i, P_j, P_l, guarantees agreement on the presence/absence of an inconsistency as conveyed by P_k. Observe that inconsistency can only be caused either due to a corrupt sender sending an incorrect value (or hash), or a corrupt receiver falsely announcing the presence of inconsistency. Hence, the fourth server, P_l, can safely be employed as TTP. The protocol appears in Fig. 4.

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

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

Send Phase: P_i sends v to P_k.
Verify Phase: P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.

- P_k sets $\mathsf{b}_k = 1$ if the received values are inconsistent or if the value is not received.
- P_k sends b_k to all servers. P_s for $s \in \{i,j,l\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.
- P_s for $s \in \{i,j,l\}$ mutually exchange their bits. P_s resets $\mathsf{b}_s = \mathsf{b}'$ where b' denotes the bit which appears in majority among $\mathsf{b}_i, \mathsf{b}_j, \mathsf{b}_l$.
- All servers set $\mathsf{TTP} = P_l$ if $\mathsf{b}' = 1$, terminate otherwise.

Figure 4: 4PC: Joint Message Passing Primitive

Notation 4.1. 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 [11]. 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 learn the TTP's identity unanimously. 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. On the efficiency front, jmp4 simply halves the communication cost of bi-convey, giving a 2× improvement.

Sharing Protocol To enable P_i to share a value v, protocol Π_{sh4} 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.

Multiplication Protocol Given the $[\![\cdot]\!]$-shares of x and y, protocol Π_{mult4} (Fig. 5) allows servers to compute $[\![\mathsf{z}]\!]$ with $\mathsf{z} = \mathsf{xy}$. When compared with the state-of-the-art 4PC GOD protocol of FLASH [11], 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 [15] that only provides security with fairness.

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

Preprocessing:

- P_0, P_3, P_j, for $j \in \{1,2\}$, sample random $[\alpha_{\mathsf{z}}]_j \in \mathbb{Z}_{2^\ell}$, and 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, and 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.

Figure 5: 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.

Reconstruction Protocol Given $[\![\mathsf{v}]\!]$, protocol Π_{rec4} 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, unlike 3PC, this protocol does not require commitments.

5 Applications and Benchmarking

In this section, we empirically show the practicality of our protocols for PPML. We consider training and inference for Logistic Regression and inference for 3 different Neural Networks (NN). NN training requires additional tools to allow mixed world computations, which we leave as future work. We refer readers to SecureML [45], ABY3 [43], BLAZE [48], FALCON [56] 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 [39] and CIFAR-10 [37] dataset. For training, we use a batch size of $B = 128$ and define 1 KB = 8192 bits.

In 3PC, we compare our results against the best-known framework BLAZE that provides fairness in the same setting. We observe that the technique of making the dot product cost independent of feature size can also be applied to BLAZE to obtain better costs. Hence, for a fair comparison, we additionally report these improved values for BLAZE. Further, we only consider the PPA circuit based variant of bit extraction for BLAZE since we aim for high throughput; the GC based variant results in huge communication and is not efficient for deep NNs. 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 [11] (which is robust) and Trident [15] (which is fair). Our results halve the cost of FLASH and are on par with Trident.

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[6], 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 40 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 as follows.

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

Software Details We implement our protocols using the publicly available ENCRYPTO library [21] 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.

Datasets We use the following datasets:

- MNIST [39] is a collection of 28×28 pixel, handwritten digit images along with a label between 0 and 9 for each image. It has $60,000$ and respectively, $10,000$ images in the training and test set. We evaluate logistic regression, and NN-1, NN-2 (cf. §5.2) on this dataset.
- CIFAR-10 [37] consists of 32×32 pixel images of 10 different classes such as dogs, horses, etc. There are $50,000$ images for training and $10,000$ for testing, with 6000 images in each class. We evaluate NN-3 (cf. §5.2) on this dataset.

Benchmarking Parameters We use *throughput* (TP) as the benchmarking parameter following BLAZE and ABY3 [43] 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. At the same time, in the latter, the model parameters are adjusted according to the difference in the computed output and the actual output. Inference can be viewed as one forward propagation of the algorithm alone. In addition to TP, we provide the online and overall communication and latency for all the benchmarked ML algorithms.

We observe that due to our protocols' asymmetric nature, the communication load is unevenly distributed among all the servers, which leaves several communication channels under-utilized. Thus, to improve the performance, we perform *load balancing*, where we run several parallel execution threads, each with roles of the servers changed. This helps in utilizing all channels and improving the performance. We report the communication and runtime of the protocols for online phase and total (= preprocessing + online).

5.1 Logistic Regression

In Logistic Regression, one iteration comprises updating the weight vector $\vec{\mathbf{w}}$ using the gradient descent algorithm (GD).

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

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 *i*th 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 3.

Setting	Ref.	Online (TP in $\times 10^3$)			Total	
		Latency (s)	Com [KB]	TP	Latency (s)	Com [KB]
3PC Training	BLAZE	0.74	50.26	4872.38	0.93	203.35
	SWIFT	1.05	50.32	4872.38	1.54	203.47
3PC Inference	BLAZE	0.66	0.28	7852.05	0.84	0.74
	SWIFT	0.97	0.34	6076.46	1.46	0.86
4PC Training	FLASH	0.83	88.93	5194.18	1.11	166.75
	SWIFT	0.83	41.32	11969.48	1.11	92.91
4PC Inference	FLASH	0.76	0.50	7678.40	1.04	0.96
	SWIFT	0.75	0.27	15586.96	1.03	0.57

Table 3: 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 inference is slightly lower compared to that of BLAZE. This is because the total number of rounds for the inference phase 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. Concretely, we observe a 2× improvement in TP for inference and a 2.3× improvement for training. For Trident [15], we observe a drop of 15.86% in TP for inference due to the extra rounds required for verification to achieve GOD. This loss is, however, traded-off with the stronger security guarantee. For training, we are on par with Trident as the effect of extra rounds becomes less significant for more number of rounds, as will also be evident from the comparisons for NN inference.

As a final remark, note that our 4PC sees roughly 2.5× improvement compared to our 3PC for logistic regression.

5.2 NN Inference

We consider the following popular neural networks for benchmarking. These are chosen based on the different range of model parameters and types of layers used in the network. We refer readers to [56] for a detailed architecture of the neural networks.

NN-1: This is a 3-layered fully connected network with ReLU activation after each layer. This network has around 118K parameters and is chosen from [43, 48].

NN-2: This network –LeNet [38]– contains 2 convolutional layers and 2 fully connected layers with ReLU activation after each layer, additionally followed by maxpool for convolutional layers. It has approximately 431K parameters.

NN-3: This network –VGG16 [52]– was the runner-up of ILSVRC-2014 competition. It has 16 layers in total and comprises of fully-connected, convolutional, ReLU activation and maxpool layers. It has about 138 million parameters.

Network	Ref.	Online			Total	
		Latency (s)	Com [MB]	TP	Latency (s)	Com [MB]
NN-1	BLAZE	1.92	0.04	49275.19	2.35	0.11
	SWIFT	2.22	0.04	49275.19	2.97	0.11
NN-2	BLAZE	4.77	3.54	536.52	5.61	9.59
	SWIFT	5.08	3.54	536.52	6.22	9.59
NN-3	BLAZE	15.58	52.58	36.03	18.81	148.02
	SWIFT	15.89	52.58	36.03	19.29	148.02

Table 4: 3PC NN Inference. TP is given in (#queries/min).

Table 4 summarises the results for 3PC NN inference. As illustrated, the performance of our 3PC framework is on par with BLAZE while providing better security guarantee.

Network	Ref.	Online			Total	
		Latency (s)	Com [MB]	TP	Latency (s)	Com [MB]
NN-1	FLASH	1.70	0.06	59130.23	2.17	0.12
	SWIFT	1.70	0.03	147825.56	2.17	0.06
NN-2	FLASH	3.93	5.51	653.67	4.71	10.50
	SWIFT	3.93	2.33	1672.55	4.71	5.40
NN-3	FLASH	12.65	82.54	43.61	15.31	157.11
	SWIFT	12.50	35.21	110.47	15.14	81.46

Table 5: 4PC NN Inference. TP is given in (#queries/min).

Table 5 summarises NN inference for 4PC setting. Here, we outperform FLASH in every aspect, with the improvement in TP being at least 2.5× for each NN architecture. Further, we are on par with Trident [15] because the extra rounds required for verification get amortized with an increase in the number of rounds required for computing NN inference. This establishes the practical relevance of our work.

As a final remark, our 4PC sees roughly 3× improvement over our 3PC for NN inference. This reflects improvements brought in by the additional honest server in the system.

Acknowledgements

We thank our shepherd Guevara Noubir, and anonymous reviewers for their valuable feedback.

Nishat Koti would like to acknowledge financial support from Cisco PhD Fellowship 2020. Mahak Pancholi would like to acknowledge financial support from Cisco MTech Fellowship 2020. Arpita Patra would like to acknowledge financial support from SERB MATRICS (Theoretical Sciences) Grant 2020 and Google India AI/ML Research Award 2020. Ajith Suresh would like to acknowledge financial support from Google PhD Fellowship 2019. The authors would also like to acknowledge the financial support from Google Cloud to perform the benchmarking.

References

[1] M. Abspoel, A. P. K. Dalskov, D. Escudero, and A. Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. In *ACNS*, 2021.

[2] B. Alon, E. Omri, and A. Paskin-Cherniavsky. MPC with Friends and Foes. In *CRYPTO*, pages 677–706, 2020.

[3] 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*, pages 843–862, 2017.

[4] 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*, pages 805–817, 2016.

[5] C. Baum, I. Damgård, T. Toft, and R. W. Zakarias. Better preprocessing for secure multiparty computation. In *ACNS*, pages 327–345, 2016.

[6] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[7] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. In *FC*, pages 325–343, 2009.

[8] 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*, pages 67–97, 2019.

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

[10] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *ACM CCS*, pages 486–497, 2007.

[11] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. FLASH: fast and robust framework for privacy-preserving machine learning. *PETS*, 2020.

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

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

[14] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*, 2019.

[15] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. *NDSS*, 2020.

[16] 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*, pages 34–64, 2018.

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

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

[19] 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*, pages 769–798, 2018.

[20] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. In *EUROCRYPT*, pages 596–613, 2003.

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

[22] A. Dalskov, D. Escudero, and M. Keller. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. Cryptology ePrint Archive, 2020. https://eprint.iacr.org/2020/1330.

[23] 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.

[24] 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*, pages 1–18, 2013.

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

[26] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

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

[28] W. Du and M. J. Atallah. Privacy-preserving cooperative scientific computations. In *IEEE CSFW-14*, pages 273–294, 2001.

[29] H. Eerikson, M. Keller, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin. Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security. In *ITC*, 2020.

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

[31] S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT*, pages 59–85, 2018.

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

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

[34] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, pages 158–189, 2018.

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

[36] N. Koti, M. Pancholi, A. Patra, and A. Suresh. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. Cryptology ePrint Archive, 2020. https://eprint.iacr.org/2020/592.

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

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

[39] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. http://yann.lecun.com/exdb/mnist/.

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

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

[42] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon. Secure parallel computation on national scale volumes of data. In *USENIX*, pages 2487–2504, 2020.

[43] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS*, pages 35–52, 2018.

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

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

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

[47] A. Patra and D. Ravi. On the exact round complexity of secure three-party computation. In *CRYPTO*, pages 425–458, 2018.

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

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

[50] 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*, pages 707–721, 2018.

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

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

[53] A. B. Slavkovic, Y. Nardi, and M. M. Tibbits. Secure logistic regression of horizontally and vertically partitioned distributed databases. In *ICDM*, pages 723–728, 2007.

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

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

[56] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PoPETS*, pages 188–208, 2021. https://arxiv.org/abs/2004.02229v1.

[57] H. Yu, J. Vaidya, and X. Jiang. Privacy-preserving SVM classification on vertically partitioned data. In *PAKDD*, pages 647–656, 2006.

A Preliminaries

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 $\mathbb{Z}_{2^\ell}$. Servers establish the following keys for 3PC: (i) one key shared between every pair– k_{01}, k_{02}, k_{12} for the servers $(P_0, P_1), (P_0, P_2)$and(P_1, P_2), respectively, (ii) 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. The key setup is modelled via a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 6) that can be realised using any secure MPC protocol. Analogously, key setup functionality for 4PC is given in Fig. 7.

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 $P_s \in \mathcal{P}$.

Figure 6: 3PC: Ideal functionality for shared-key setup

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

$\mathcal{F}_{\mathsf{setup4}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup4}}$ 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 $P_s \in \mathcal{P}$.

Figure 7: 4PC: Ideal functionality for shared-key setup

To generate a *3-out-of-3* additive sharing of 0 i.e. ζ_s for $s \in \{0,1,2\}$ such that P_s holds ζ_s, and $\zeta_0 + \zeta_1 + \zeta_2 = 0$, servers proceed as follows. Every pair of servers, $P_s, P_{(s+1)\%3}$, non-interactively generate r_s, as described earlier, and each P_s sets $\zeta_s = \mathsf{r}_s - \mathsf{r}_{(s-1)\%3}$.

B 3PC Protocols

Joint Message Passing: The ideal functionality for jmp appears in Fig. 8.

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 server with smallest index. Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{TTP}$

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

Figure 8: 3PC: Ideal functionality for jmp primitive

Joint Sharing: The joint sharing protocol appears in Fig. 9.

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.

Figure 9: 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 6.

Multiplication: The ideal functionality for Π_{mulPre} appears in Fig. 10.

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 $P_s \in \mathcal{P}$.

Figure 10: 3PC: Ideal functionality for Π_{mulPre} protocol

	(P_1, P_2)	(P_1, P_0)	(P_2, P_0)
	$[\alpha_v]_1 = 0,\ [\alpha_v]_2 = 0$ $\beta_v = v, \gamma_v = r - v$	$[\alpha_v]_1 = -v,\ [\alpha_v]_2 = 0$ $\beta_v = 0, \gamma_v = r$	$[\alpha_v]_1 = 0,\ [\alpha_v]_2 = -v$ $\beta_v = 0, \gamma_v = r$
P_0	$(0,\ 0,\ r\ \ )$	$(-v,\ 0,\ r)$	$(0,\ -v,\ r)$
P_1	$(0,\ v,\ r - v)$	$(-v,\ 0,\ r)$	$(0,\ \ 0,\ r)$
P_2	$(0,\ v,\ r - v)$	$(\ 0,\ 0,\ r)$	$(0,\ -v,\ r)$

Table 6: The columns depict the three distinct possibilities 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.

Dot Product: The ideal world functionality for realizing Π_{dotpPre} is presented in Fig. 11.

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 $\vec{\mathbf{d}} = (\mathsf{d}_1, \ldots, \mathsf{d}_n), \vec{\mathbf{e}} = (\mathsf{e}_1, \ldots, \mathsf{e}_n)$ from the servers. Let $\mathsf{v}_{j,s}$ for $j \in [\mathsf{n}], s \in \{0,1,2\}$ denote the share of v_j such that $\mathsf{v}_j = \mathsf{v}_{j,0} + \mathsf{v}_{j,1} + \mathsf{v}_{j,2}$. 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 $\langle \mathsf{e}_j \rangle_s = (\mathsf{e}_{j,s}, \mathsf{e}_{j,(s+1)\%3})$ 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 $P_s \in \mathcal{P}$.

Figure 11: 3PC: Ideal functionality for Π_{dotpPre} protocol

Truncation: We now give details of how to generate $([\mathsf{r}], \llbracket \mathsf{r}^d \rrbracket)$. For this, servers proceed as follows: P_0, P_j for $j \in \{1,2\}$ sample random $\mathsf{r}_j \in \mathbb{Z}_{2^\ell}$. Recall that the bit at ith position in r is denoted as $\mathsf{r}[i]$. Define $\mathsf{r}[i] = \mathsf{r}_1[i] \oplus \mathsf{r}_2[i]$ for $i \in \{0, \ldots, \ell - 1\}$. For r defined as above, we have $\mathsf{r}^d[i] = \mathsf{r}_1[i+d] \oplus \mathsf{r}_2[i+d]$ for $i \in \{0, \ldots, \ell - d - 1\}$. Further,

$$\begin{aligned} \mathsf{r} &= \sum_{i=0}^{\ell-1} 2^i \mathsf{r}[i] = \sum_{i=0}^{\ell-1} 2^i (\mathsf{r}_1[i] \oplus \mathsf{r}_2[i]) \\ &= \sum_{i=0}^{\ell-1} 2^i \left((\mathsf{r}_1[i])^{\mathsf{R}} + (\mathsf{r}_2[i])^{\mathsf{R}} - 2(\mathsf{r}_1[i])^{\mathsf{R}} \cdot (\mathsf{r}_2[i])^{\mathsf{R}} \right) \\ &= \sum_{i=0}^{\ell-1} 2^i \left((\mathsf{r}_1[i])^{\mathsf{R}} + (\mathsf{r}_2[i])^{\mathsf{R}} \right) - \sum_{i=0}^{\ell-1} \left(2^{i+1} (\mathsf{r}_1[i])^{\mathsf{R}} \right) \cdot (\mathsf{r}_2[i])^{\mathsf{R}} \qquad (5) \end{aligned}$$

Similarly, for r^d we have the following,

$$\mathsf{r}^d = \sum_{i=d}^{\ell-1} 2^{i-d} \left((\mathsf{r}_1[i])^{\mathsf{R}} + (\mathsf{r}_2[i])^{\mathsf{R}} \right) - \sum_{i=d}^{\ell-1} \left(2^{i-d+1} (\mathsf{r}_1[i])^{\mathsf{R}} \right) \cdot (\mathsf{r}_2[i])^{\mathsf{R}} \qquad (6)$$

The servers non-interactively generate $\llbracket \cdot \rrbracket$-shares (arithmetic shares) for each bit of r_1 and r_2 as in Table 6. Given their $\llbracket \cdot \rrbracket$-shares, the servers execute Π_{dotp} twice to compute $\llbracket \cdot \rrbracket$-share of $\mathsf{A} = \sum_{i=d}^{\ell-1} (2^{i-d+1} (\mathsf{r}_1[i])^{\mathsf{R}}) \cdot (\mathsf{r}_2[i])^{\mathsf{R}}$, and $\mathsf{B} = \sum_{i=0}^{\ell-1} (2^{i+1} (\mathsf{r}_1[i])^{\mathsf{R}}) \cdot (\mathsf{r}_2[i])^{\mathsf{R}}$. Using these values, the servers can locally compute the $\llbracket \cdot \rrbracket$-shares for $(\mathsf{r}, \mathsf{r}^d)$ pair following Equation 5 and 6, respectively. Note that servers need $[\cdot]$-shares of r and not $\llbracket \cdot \rrbracket$-shares. The $[\cdot]$-shares can be computed from the $\llbracket \cdot \rrbracket$-shares locally as follows. Let $(\alpha_{\mathsf{r}}, \beta_{\mathsf{r}}, \gamma_{\mathsf{r}})$ be the values corresponding to the $\llbracket \cdot \rrbracket$-shares of r. Since P_0 knows the entire value r in clear, and it knows α_{r}, it can locally compute β_{r}. Now, the servers set $[\cdot]$-shares as: $[\mathsf{r}]_1 = -[\alpha_{\mathsf{r}}]_1$ and $[\mathsf{r}]_2 = \beta_{\mathsf{r}} - [\alpha_{\mathsf{r}}]_2$. The protocol appears in Fig. 12.

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

– To generate each bit $\mathsf{r}[i]$ of r for $i \in \{0, \ldots, \ell - 1\}$, P_0, P_j for $j \in \{1,2\}$ sample random $\mathsf{r}_j[i] \in \mathbb{Z}_2$ and define $\mathsf{r}[i] = \mathsf{r}_1[i] \oplus \mathsf{r}_2[i]$.

– Servers generate $\llbracket \cdot \rrbracket$-shares of $(\mathsf{r}_j[i])^{\mathsf{R}}$ for $i \in \{0, \ldots, \ell - 1\}, j \in \{1,2\}$ non-interactively following Table 6.

– Define $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$ such that $\mathsf{x} = 2^{i-d+1} (\mathsf{r}_1[i])^{\mathsf{R}}$ and $\mathsf{y}_i = (\mathsf{r}_2[i])^{\mathsf{R}}$, respectively, for $i \in \{d, \ldots, \ell - 1\}$. Define $\vec{\mathbf{p}}$ and $\vec{\mathbf{q}}$ such that $\mathsf{p}_i = 2^{i+1} (\mathsf{r}_1[i])^{\mathsf{R}}$ and $\mathsf{q}_i = (\mathsf{r}_2[i])^{\mathsf{R}}$, respectively, for $i \in \{0, \ldots, \ell - 1\}$. Servers execute Π_{dotp} to compute $\llbracket \cdot \rrbracket$-shares of $\mathsf{A} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ and $\mathsf{B} = \vec{\mathbf{p}} \odot \vec{\mathbf{q}}$.

– Servers locally compute $\llbracket \mathsf{r}^d \rrbracket = \sum_{i=d}^{\ell-1} 2^{i-d} (\llbracket (\mathsf{r}_1[i])^{\mathsf{R}} \rrbracket + \llbracket (\mathsf{r}_2[i])^{\mathsf{R}} \rrbracket) - \llbracket \mathsf{A} \rrbracket$, and $\llbracket \mathsf{r} \rrbracket = \sum_{i=0}^{\ell-1} 2^i (\llbracket (\mathsf{r}_1[i])^{\mathsf{R}} \rrbracket + \llbracket (\mathsf{r}_2[i])^{\mathsf{R}} \rrbracket) - \llbracket \mathsf{B} \rrbracket$.

– P_0 locally computes $\beta_{\mathsf{r}} = \mathsf{r} + \alpha_{\mathsf{r}}$. P_0, P_1 set $[\mathsf{r}]_1 = -[\alpha_{\mathsf{r}}]_1$ and P_0, P_2 set $[\mathsf{r}]_2 = \beta_{\mathsf{r}} - [\alpha_{\mathsf{r}}]_2$.

Figure 12: 3PC: Generating Random Truncated Pair $(\mathsf{r}, \mathsf{r}^d)$

C 4PC Protocols

4PC Joint Message Passing Primitive: The ideal functionality for jmp4 primitive appears in Fig. 13.

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 $\mathsf{TTP} = P_l$ should be established.

Step 2: If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{ttp} = 0$, or if $\mathcal{S}$ has corrupted P_l, 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 $s \in \{0,1,2,3\}$.

Figure 13: 4PC: Ideal functionality for jmp4 primitive

Joint Sharing: Protocol Π_{jsh4} (Fig. 14) enables a pair of servers (P_i, P_j) to jointly generate a $\llbracket\cdot\rrbracket$-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.

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.

Figure 14: 4PC: $\llbracket\cdot\rrbracket$-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. 14). 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 $\llbracket\cdot\rrbracket$-sharing of v non-interactively when v is available with P_0, P_1, P_2. For 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.

Input Sharing and Output Reconstruction in SOC Setting We extend input sharing and reconstruction in the SOC setting as follows. To generate $\llbracket\cdot\rrbracket$-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) [49] to agree on u (or $\perp$). We refer the readers to [49] for the formal details of the agreement protocol. 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 $\llbracket\cdot\rrbracket$-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 $\llbracket\cdot\rrbracket$-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.

Dot Product Given $\llbracket\cdot\rrbracket$-shares of two n-sized vectors $\vec{\mathbf{x}}, \vec{\mathbf{y}}$, Π_{dotp4} enables servers to compute $\llbracket\mathsf{z}\rrbracket$ 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 helps to obtain a communication cost independent of the size of the vectors. Details are deferred to the full version [36].

Truncation Given the $\llbracket\cdot\rrbracket$-sharing of a value v, protocol $\Pi_{\mathsf{trgen4}}(\mathcal{P})$ enables the servers to compute the $\llbracket\cdot\rrbracket$-sharing of the truncated value v^d (right shifted value by, say, d positions). For this, given $\llbracket\mathsf{v}\rrbracket$ and a random truncation pair $([\mathsf{r}], \llbracket\mathsf{r}^d\rrbracket)$, the value $(\mathsf{v} - \mathsf{r})$ is opened, truncated and added to $\llbracket\mathsf{r}^d\rrbracket$ to obtain $\llbracket\mathsf{v}^d\rrbracket$. To generate the a random truncation pair, 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 $\mathsf{r} = R_1 + R_2$ while P_j sets $[\mathsf{r}]_j = R_j$. Then, P_0, P_3 locally truncate r to obtain r^d and execute $\Pi_{\mathsf{jsh4}}(P_0, P_3, \mathsf{r}^d)$ to generate $\llbracket\mathsf{r}^d\rrbracket$.

Dot Product with Truncation Protocol Π_{dotpt4} (Fig. 15) enables servers to generate $\llbracket\cdot\rrbracket$-sharing of the truncated value of $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$, denoted as z^d, given the $\llbracket\cdot\rrbracket$-sharing of n-sized vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$.

Protocol $\Pi_{\mathsf{dotpt4}}(\mathcal{P}, \{\llbracket\mathsf{x}_i\rrbracket, \llbracket\mathsf{y}_i\rrbracket\}_{i \in [\mathsf{n}]})$

Preprocessing :

– Servers execute the preprocessing phase of $\Pi_{\mathsf{dotp4}}(\mathcal{P}, \{\llbracket\mathsf{x}_i\rrbracket, \llbracket\mathsf{y}_i\rrbracket\}_{i \in [\mathsf{n}]})$.
– Servers execute $\Pi_{\mathsf{trgen4}}(\mathcal{P})$ to generate the truncation pair $([\mathsf{r}], \llbracket\mathsf{r}^d\rrbracket)$. P_0 obtains the value r in clear.

Online :

– 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 sets $[(\mathsf{z} - \mathsf{r})^\star]_j = [\Psi]_j + [\chi]_j$.
– P_1, P_0 jmp4-send $[(\mathsf{z} - \mathsf{r})^\star]_1$ to P_2 and P_2, P_0 jmp4-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 $\Pi_{\mathsf{jsh4}}(P_1, P_2, (\mathsf{z} - \mathsf{r})^d)$ to generate $\llbracket(\mathsf{z} - \mathsf{r})^d\rrbracket$.
– Servers locally compute $\llbracket\mathsf{z}^d\rrbracket = \llbracket(\mathsf{z} - \mathsf{r})^d\rrbracket + \llbracket\mathsf{r}^d\rrbracket$.

Figure 15: 4PC: Dot Product Protocol with Truncation

Special protocols Similar to 3PC, we consider the following special protocols for 4PC – i) Bit Extraction, ii) Bit2A, and iii) Bit Injection. These are elaborated in the full version of the paper [36]. Protocols for secure comparison, maxpool, convolution and matrix multiplication, follow a similar outline as described for the 3PC case, except that the underlying primitives used will be based on 4PC (defined in §4).