BLAZE: Blazing Fast Privacy-Preserving Machine Learning

Arpita Patra*, Ajith Suresh*
*Indian Institute of Science, Bangalore
{arpita, ajith}@iisc.ac.in

***Abstract*—Machine learning tools have illustrated their potential in many significant sectors such as healthcare and finance, to aide in deriving useful inferences. The sensitive and confidential nature of the data, in such sectors, raise natural concerns for the privacy of data. This motivated the area of Privacy-preserving Machine Learning (PPML) where privacy of the data is guaranteed. Typically, ML techniques require large computing power, which leads clients with limited infrastructure to rely on the method of Secure Outsourced Computation (SOC). In SOC setting, the computation is outsourced to a set of specialized and powerful cloud servers and the service is availed on a pay-per-use basis. In this work, we explore PPML techniques in the SOC setting for widely used ML algorithms– Linear Regression, Logistic Regression, and Neural Networks.**

We propose BLAZE, a blazing fast PPML framework in the three server setting tolerating one malicious corruption over a ring ($\mathbb{Z}_{2^\ell}$). BLAZE achieves the stronger security guarantee of fairness (all honest servers get the output whenever the corrupt server obtains the same). Leveraging an *input-independent* pre-processing phase, BLAZE has a fast input-dependent online phase relying on efficient PPML primitives such as: (i) A dot product protocol for which the communication in the online phase is *independent* of the vector size, the first of its kind in the three server setting; (ii) A method for truncation that shuns evaluating expensive circuit for Ripple Carry Adders (RCA) and achieves a constant round complexity. This improves over the truncation method of ABY3 (Mohassel et al., CCS 2018) that uses RCA and consumes a round complexity that is of the order of the depth of RCA (which is same as the underlying ring size).

An extensive benchmarking of BLAZE for the aforementioned ML algorithms over a 64-bit ring in both WAN and LAN settings shows massive improvements over ABY3. Concretely, we observe improvements up to $333\times$ for Linear Regression, $53\times$ for Logistic Regression and $276\times$ for Neural Networks over WAN. Similarly, we show improvements up to $2610\times$ for Linear Regression, $54\times$ for Logistic Regression and $278\times$ for Neural Networks over LAN.

I. Introduction

Machine learning (ML) is increasingly becoming one of the dominant research fields. Advancement in the domain has myriad real-life applications– from smart keyboard predictions to more involved operations such as self-driving cars. It also finds useful applications in impactful fields such as healthcare and medicine, where ML tools are being used to assist healthcare specialists in better diagnosing abnormalities. This surge in interest in the field is bolstered by the availability of a large amount of data with the rise of companies such as Google and Amazon. This is also due to improved, more robust and accurate ML algorithms in use today. With better machinery and tools such as deep learning and reinforcement learning, ML techniques are starting to beat humans at some difficult tasks such as classifying echocardiograms [1].

In order to be deployed in practice, ML models face numerous challenges. The primary challenge is to provide a high level of accuracy and robustness, as it is imperative for the functioning of some mission-critical fields such as health care. Accuracy and robustness are contingent on a high amount of computing power and availability of data from more varied sources. Accumulating data from different and various sources is not practical for a single company/stake-holder to realize. Moreover, policies like the European Union General Data Protection Regulation (GDPR) or the EFF's call for information fiduciary rules for businesses have made it difficult and even illegal for companies to share datasets with each other without the prior consent of customers. In some cases, it might even be infeasible for companies to share their data with each other as it is proprietary information and sharing it may give rise to concerns such as competitive advantage. While in other cases, the data might be too sensitive, such as medical and financial records, that a breach of privacy cannot be tolerated. It is also possible that the companies providing ML services to clients risk leaking the model parameters rendering its services redundant, and the individual client's or company's data no longer private. In the light of huge interest in using ML and simultaneous requirement of security of data, the field of privacy-preserving machine learning (PPML) has emerged as a flourishing research area. These techniques can be used to ensure that no information about the query or dataset is leaked other than what is permissible by the algorithm, which in some cases might be only the prediction output.

Many everyday end-users are not equipped with computing infrastructure capable of efficiently executing compute-intensive ML algorithms. It is economical and convenient for end-users to outsource an ML task to more powerful and specialized systems. To ensure data privacy while outsourcing, we use Secure Outsourced Computation (SOC) as a potential solution. SOC allows end-users to securely outsource computation to a set of specialized and powerful cloud servers and avail its services on a pay-per-use basis. SOC guarantees that individual data of the end-users remain private, tolerating reasonable collusion amongst the servers.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24202
www.ndss-symposium.org

PPML, both for training and inference, can be realized in the SOC setting. Firstly, an end-user posing as a model-owner can host its trained machine learning model, in a secret-shared way, to a set of (untrusted) servers. An end-user as a customer can secret-share its query amongst the same servers to allow the prediction to be computed in a shared fashion and to finally obtain the prediction result. Secondly, multiple data-owners can host their datasets in a shared way amongst a set of (untrusted) servers and can train a common model on their joint datasets while keeping their individual dataset private. Recently, many works [2]–[6], solve the aforementioned goals using the techniques of Secure Multiparty Computation (MPC) where the untrusted servers are taken as the participants (or parties) of the MPC. The corrupt server(s) can collude with an arbitrary number of data-owners in case of training and with either the model-owner or the customer in case of inference. Privacy of the end-users is ensured leveraging the security guarantees of MPC.

MPC is arguably the standard-bearer problem in cryptography. It allows n mutually distrusting parties to perform computations together on their private inputs, so that an adversary controlling at most t parties, can not learn any information beyond what is already known and permissible by the computation. MPC for a small number of parties in the *honest majority* setting, specifically the setting of 3 parties with one corruption, has become popular over the last few years due to its spectacular performance [7]–[17], leveraging the presence of single corruption. Applications such as financial data analysis [18], email spam filtering [19], distributed credential encryption [12], privacy-preserving statistical studies [20] and popular MPC frameworks such as Sharemind [21], VIFF [22] involve 3 parties.

In an effort to improve the practical efficiency, many recent works divide their protocol into two phases, namely– i) *input-independent* preprocessing phase and ii) *input-dependent* online phase. This has become a prominent approach in both theoretical [23]–[28] and practical [4], [29]–[36] domains. The preprocessing phase is used to perform a relatively expensive computation that is independent of the input. In the online phase, once the inputs become available, the actual computation can be performed in a fast way making use of the pre-computed data. This paradigm suits scenarios analogous to our setting, where functions typically need to be evaluated a large number of times, and the function description is known beforehand.

There has been a recent paradigm shift of designing MPC over rings, considering the fact that computer architectures use rings of size 32 or 64 bits. Designing and implementing MPC protocols over rings can leverage CPU optimizations and have been proven to have a significant impact on efficiency [21], [34], [37]–[39]. Furthermore, operating over rings avoids the need to overload basic operations such as addition and multiplication during implementation, or rely on an external library as compared to working over prime order fields.

Although MPC techniques can be used to realize SOC, the current best MPC techniques cannot be directly plugged into ML algorithms, largely due to the following reasons. Firstly, in ML domain, most of the computation happens over decimal values, requiring us to embed the decimal values in 2's complement form over a ring ($\mathbb{Z}_{2^\ell}$). As a natural consequence of this embedding, repeated multiplications cause an overflow in the ring. The naive solution is to pick a ring large enough to avoid the overflow, but the number of sequential multiplications in a typical ML algorithm makes this solution impractical. The existing works [2], [5], [6] tackled this problem through a secure truncation, a very important primitive by now, which approximates the value by sacrificing the accuracy by an infinitesimal amount, performed after every multiplication. Secondly, certain functions such as comparison or the widely used activation such as ReLU or Sigmoid, requiring extraction of MSB in a privacy-preserving manner, needs involvement of the boolean world (over the ring $\mathbb{Z}_{2^1}$), while functions such as addition, dot product are more efficient when performed in the arithmetic domain (over the ring $\mathbb{Z}_{2^\ell}$). The ML algorithms involve a mix of operations, constantly alternating between these two worlds. As shown in some of the recent works [2], [5], [38], using mixed world computation is orders of magnitude more efficient as compared to most of the current best MPC techniques which operate only in either of the two worlds. Thirdly, while a typical MPC offers a way to tackle a multiplication gate, ML algorithms invoke its variant dot product. A naive way of doing privacy-preserving dot product would invoke the method of multiplication ℓ times, with ℓ being the size of the input vectors. With ML algorithms dealing with humongous size data vectors, the naive approach may turn expensive and so customized way of performing dot product that attains independence from the vector size in its complexity is called for. In other words, PPML would need customized privacy-preserving building blocks, such as dot product, truncation, comparison, ReLU, Sigmoid etc., rather than the typical building blocks such as addition and multiplication of MPC.

A. Related Work

In the regime of PPML using MPC, earlier works considered the widely-used ML algorithms such as Decision Trees [40], Linear Regression [41], [42], k-means clustering [43], [44], SVM Classification [45], [46], and Logistic Regression [47]. However, these solutions are far from practical due to the high overheads that they incur. SecureML [2] proposed a practically-efficient PPML framework in the two-server model using a mix of 2PC protocols that perform computation in **A**rithmetic, **B**oolean and **Y**ao style (aka. ABY framework [38]). One of their key contributions is a novel method for truncating decimal numbers. They consider training for linear regression, logistic regression, and neural network models. The work of Chameleon [4] considered a 2PC setting where parties availed the help of a semi-trusted third party and consider SVMs and Neural Networks. Both SecureML and Chameleon considered semi-honest corruption only. The ABY framework was extended to the three-party setting by ABY3 [5] and SecureNN [6] (the latter consider neural networks only). These works consider malicious security and demonstrate that the honest-majority setting can be leveraged to improve the performance by several orders of magnitude. Recently, ASTRA [48] furthered this line of work and improve upon ABY3. However, ASTRA presents a set of primitives to build protocols for Linear Regression and Logistic Regression inference. For the training of these ML algorithms and NN prediction, additional primitives like truncation, bit to arithmetic conversions are required, which are not considered in ASTRA.

B. Our Contribution

We propose an efficient PPML framework over the ring $\mathbb{Z}_{2^\ell}$ in a SOC setting, with three servers amongst which at most one can be maliciously corrupt. The framework consists of a range of ML tools realized in a privacy-preserving way which is ensured via running computation in a secret-shared fashion. We introduce a new secret-sharing semantics for three servers over a ring $\mathbb{Z}_{2^\ell}$ tolerating up to one malicious corruption, which is the basis for all our constructions. We use the sharing over both $\mathbb{Z}_{2^\ell}$ and its special instantiation $\mathbb{Z}_{2^1}$ and refer them as *arithmetic* and respectively *boolean* sharing.

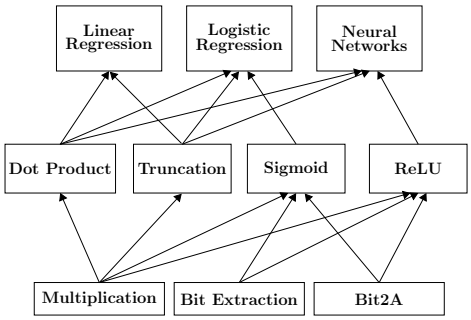


Fig. 1: Hierarchy of primitives in BLAZE Framework

Our framework, as depicted in Fig. 1, consists of three layers with the 3rd and final layer consisting of the privacy-preserving realization of various ML algorithms and forming the end goal of our framework– i) Linear Regression, ii) Logistic Regression, and iii) Neural Networks (NN). The 3rd layer builds upon the privacy-preserving realisation of 2nd layer primitives– (i) Dot Product: This is used to generate arithmetic sharing of $\vec{\mathbf{x}} \odot \vec{\mathbf{y}}$, given the arithmetic sharing of each element of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, (ii) Truncation: Given the arithmetic sharing of a value v, this generates the arithmetic sharing of truncated version of the value for a publicly known truncation parameter, and (iii) Non-linear Activation functions (Sigmoid and ReLU): Given the arithmetic sharing of a value, this is used to generate the arithmetic sharing of the resultant value obtained after applying the respective activation function on it. The 2nd layer builds upon the privacy-preserving realization of 1st layer primitives– (i) Multiplication: This is used to generate arithmetic sharing of $\mathsf{x} \cdot \mathsf{y}$, given the arithmetic sharing of values x and y, (ii) 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, and (iii) Bit to Arithmetic sharing Conversion (Bit2A): This is used to convert the boolean sharing of a single bit value to its arithmetic sharing. The above tools, designed with a focus on practical efficiency, are cast in *input-independent* preprocessing phase, and *input-dependent* online phase. Our contributions, presented in top-down fashion starting with the end-goals (3rd layer), can be summed up as follows.

Performance of PPML Algorithms (Layer-III).

– Relying on our efficient Layer-II building blocks, our framework BLAZE results in blazing fast PPML protocols for Linear Regression, Logistic Regression, and NN. We consider both training and inference for Linear Regression and Logistic Regression and inference alone for NN. Our 2nd layer primitives are enough to provide support to build all the above. Extending our framework for NN training will require Garbled circuit techniques, and seamless transitions across the three worlds arithmetic, boolean and Yao. We leave this as an interesting open direction.

We illustrate the performance of BLAZE via thorough benchmarking and compare with its closest competitors ABY3 [5] and ASTRA [48]. While the primitives built in ASTRA suffice for secure inference of Linear Regression and Logistic Regression, they do not suffice for secure training of the aforementioned algorithms and secure NN inference. These require additional tools such as truncation, bit to arithmetic conversion. Also, in ASTRA, the inference phase of Logistic Regression produces a boolean sharing of the output (as an efficiency optimization), while an arithmetic sharing is needed to continue with further computation in case of training. For these reasons, we apply the same optimizations as proposed in ASTRA while comparing the performance of Linear Regression and Logistic Regression inferences with ASTRA.

We provide benchmarking for both preprocessing and online phase separately over a 64-bit ring ($\mathbb{Z}_{2^\ell}$) in both WAN and LAN settings. We use *throughput* as the benchmarking parameter, which 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. For training, we benchmarked over batch sizes 128, 256 and 512 and feature size ranging from 100 to 900. For inference, in addition to the benchmarking over the aforementioned feature sizes, we benchmarked over real-world datasets as well. Table I summarizes the gain in throughput of our protocols over ABY3 for different ML algorithms.

Layer-III: PPML Algorithms				
Algorithm	Preprocessing		Online	
	WAN	LAN	WAN	LAN
Linear Regression (Training)	**4.01**× to **4.08**×	**4.01**× to **4.08**×	**18.54**× to **333.72**×	**138.89**× to **2610.76**×
Logistic Regression (Training)	**1.97**× to **2.96**×	**1.92**× to **2.98**×	**6.13**× to **53.19**×	**6.11**× to **53.21**×
Linear Regression (Inference)	**4.02**× to **5.00**×	**4.02**× to **5.21**×	**2.81**× to **194.86**×	**52.00**× to **3600.00**×
Logistic Regression (Inference)	**1.32**× to **2.36**×	**1.34**× to **2.41**×	**3.18**× to **27.52**×	**3.16**× to **27.04**×
Neural Networks (Inference)	**4.02**× to **4.07**×	**4.02**× to **4.07**×	**65.46**× to **276.31**×	**64.89**× to **276.84**×

TABLE I: Summary of BLAZE's Gain in Throughput over ABY3

In order to emphasise the improved communication of our protocols, we benchmarked over varied bandwidth from 25 to 75Mbps in WAN.

When compared with ASTRA, we observe improvements up to $194\times$ and $15\times$ over Linear Regression and Logistic Regression inference respectively over WAN. The respective improvements over LAN are $1800\times$ and $16\times$. Note that ASTRA has not considered the training of the above algorithms and NN inference.

Primary Building Blocks for PPML (Layer-II).

– *Dot Product*: Dot Product forms the fundamental building block for most of the ML algorithms and hence designing efficient constructions for the same are of utmost importance. We propose an efficient dot product protocol for which the communication in the online phase is independent of the size of the underlying vectors. Ours is the first solution in the three-party honest-majority and malicious setting, to achieve such

a result. Concretely, our solution requires communication of $3\mathsf{n}$ and 3 ring elements respectively in the preprocessing and online phases, where n denotes the size of the underlying vectors. When compared with the dot product protocol of ABY3, which requires communication of $12\mathsf{n}$ and $9\mathsf{n}$ ring elements in the preprocessing phase and online phase, our protocol results in the corresponding improvement of $4\times$ and $3\mathsf{n}\times$. Similar comparison with ASTRA [48], which requires communication of $21\mathsf{n}$ and $2\mathsf{n}+2$ ring elements in the preprocessing phase and online phase, our protocol results in respective improvements of $7\times$ and $\approx 0.67\mathsf{n}\times$.

Building Blocks	Ref.	Preprocessing		Online	
		R	C (ℓ)	R	C (ℓ)
Layer-II: Building Blocks for PPML					
Dot Product	ABY3	4	$12\mathsf{n}$	1	$9\mathsf{n}$
	ASTRA	6	$21\mathsf{n}$	**1**	$\approx 2\mathsf{n}$
	BLAZE	4	$\mathbf{3n}$	1	**3**
Dot Product with Truncation	ABY3	2ℓ - 2	$\approx 12\mathsf{n}+84$	2	$9\mathsf{n}+3$
	BLAZE	4	$\mathbf{3n+2}$	1	**3**
Sigmoid	ABY3	4	≈ 108	$\log \ell$ + 4	≈ 81
	BLAZE	5	$\approx 5\kappa+23$	**5**	$\approx \kappa+11$
ReLU	ABY3	4	60	$\log \ell$ + 3	45
	BLAZE	5	$\approx \mathbf{5}\kappa+\mathbf{14}$	**4**	$\approx \kappa+7$
Layer-I: Privacy-preserving Primitives					
Multiplication	ABY3	4	12	1	9
	ASTRA	6	21	1	4
	BLAZE	4	**3**	1	**3**
Bit Extraction	ABY3	4	24	$1+\log \ell$	18
	ASTRA	7	46	3	≈ 6
	BLAZE	4	**9**	$1+\log \ell$	**9**
	BLAZE	5	$\approx 5\kappa+2$	**2**	$\approx \kappa$
Bit2A	ABY3	4	24	2	18
	BLAZE	5	**9**	**1**	**4**

– Notations: ℓ - size of ring in bits, κ - computational security parameter, n - size of vectors for dot product, 'R' - number of rounds, 'C' - total communication in units of ℓ bits.
– ABY3, ASTRA and BLAZE requires an additional two rounds of interaction in the Online Phase for verification.

TABLE II: Comparison of ABY3 [5], ASTRA [48] and BLAZE in terms of Communication and Round Complexity

– *Truncation*: For ML applications where the inputs are floating-point numbers, the protocol for truncation plays a crucial role in determining the overall efficiency of the proposed solution. Towards this, we propose an efficient truncation protocol for the three server setting. When incorporated into our dot product protocol, our truncation method adds a very minimal overhead of just two ring elements in the preprocessing phase of the dot product protocol and more importantly keeps its online complexity intact. In contrast, the state-of-the-art protocol of ABY3 requires expensive Ripple Carry Adder (RCA) circuits in the preprocessing phase which consumes rounds proportional to the underlying ring size. Moreover, their solution demands an additional round of communication with 3 ring elements in the online phase.
– *Non-linear Activation functions*: We provide efficient instantiation for Sigmoid and ReLU activation functions. The former is used in Logistic Regression, while the latter is used in Neural Networks. Our constructions require only constant round of communication (≤ 4) in the online phase as opposed to ABY3. Moreover, we improve upon ABY3 in terms of online communication by a factor of $\approx 4.5\times$.

Layer-I: Secondary Building Blocks for PPML

– *Multiplication*: We propose a new and efficient multiplication protocol for the 3 server setting that can tolerate at most one malicious corruption. Our construction invokes the multiplication protocol of [17] (which uses distributed Zero Knowledge) in the preprocessing phase to facilitate an efficient online phase. Concretely, our protocol requires an amortized communication of 3 ring elements in both the preprocessing and online phases. Apart from the improvement in communication, the asymmetric nature of our protocol enables one among the three servers to be idle majority of the time during the input-dependent phase. This construct serves as the primary building block for our dot product protocol.
While the multiplication protocol of [17] performs better than ours with a communication complexity of 3 ring elements overall yet in an amortized sense, we choose our construct over it mainly due to the huge benefits it brings for the case of dot product protocol. The dot product for n-length vectors can be viewed as n multiplications. Using [17] for the same will result in a communication of $3\mathsf{n}$ (amortized) ring elements in the online phase. For the communication cost to get amortized, the protocol of [17] requires a large number of multiplications to be performed together, which cannot be guaranteed for several instances such as inference phases of Linear Regression and Logistic Regression. Furthermore, their protocol makes use of expensive public-key cryptography, which is undesirable in settings similar to ours, where practical efficiency is of utmost importance in the online phase.
On the other hand, our construct for multiplication when tweaked to obtain a dot product protocol requires communication of $3\mathsf{n}+3$ ring elements overall, where the preprocessing phase takes care of the expensive part involving invoking [17] and bearing heavy communication of $3\mathsf{n}$ elements. This results in a blazing fast online phase for dot product which requires communication of just 3 ring elements and symmetric key operations. Lastly, as our setting calls for the computation of many multiplication operations in the preprocessing phase, the protocol of [17] is used to perform them, and the communication cost gets amortized over many multiplication operations.
– *Bit Extraction*: We provide two constructions based on the solutions proposed by ASTRA [48] and ABY3 [5]. While the solution based on ASTRA results in constant round complexity, the one based on ABY3 requires $\log(\ell)$ rounds where ℓ denotes the size of the underlying ring in bits.
– *Bit to Arithmetic sharing Conversion (Bit2A)*: The arithmetic equivalent of a bit $\mathsf{b} = \mathsf{b}_1 \oplus \mathsf{b}_2$ can be written as $(\mathsf{b})^{\mathbf{A}} = (\mathsf{b}_1)^{\mathbf{A}} + (\mathsf{b}_2)^{\mathbf{A}} - 2(\mathsf{b}_1)^{\mathbf{A}}(\mathsf{b}_2)^{\mathbf{A}}$. Here $(\mathsf{b})^{\mathbf{A}}$ denotes the value of bit b in ring $\mathbb{Z}_{2^\ell}$. Thus the servers generate the arithmetic sharing of each of the shares of bit b and their product and use the aforementioned relation to compute the final result. Our protocol, when compared to ABY3, gives $3\times$ and $4\times$ improvement with respect to the communication cost, in the preprocessing and online phase, respectively.

II. Preliminaries and Definitions

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. We consider a static and Byzantine adversary, who can corrupt at most one of the three servers. In the case of ML training, many data-owners who wish to jointly train the model, secret-shares (as per schemes discussed

latter) their data amongst the three servers. In the case of ML inference, a model-owner and a client secret-share the trained model and query respectively among the three servers. Once all the inputs are available in shared fashion, servers perform the computation to generate the output in a shared format among them. For training, the output model is then reconstructed back to the data owners while for inference, the prediction result is reconstructed towards the client alone. We assume that an arbitrary number of data owners can collude with the corrupt server for training, while for inference, either the model-owner or the client can collude with the corrupt server. The same setting has been considered by ASTRA [48], ABY3 [5], and other related papers.

For a vector $\vec{\mathsf{x}}$, x_i denotes the i^{th} element in the vector. For two vectors $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$ of length n, the dot product is given by, $\vec{\mathsf{x}} \odot \vec{\mathsf{y}} = \sum_{i=1}^{\mathsf{n}} \mathsf{x}_i \mathsf{y}_i$. Given two matrices $\mathbf{X}, \mathbf{Y}$, the operation $\mathbf{X} \circ \mathbf{Y}$ denotes the matrix multiplication.

a) Input-independent and Input-dependent Phases: The protocols of this work are cast into two phases: *input-independent* preprocessing phase and *input-dependent* online phase. This approach is useful in outsourced setting where the servers execute several instances of an agreed-upon function. The preprocessing for multiple instances can be executed in parallel. It is plausible for some of the protocols to have empty input-independent phase.

b) Shared Key Setup: To facilitate non-interactive communication, parties use a one-time key setup that establishes pre-shared random keys for a pseudo-random function (PRF) among them. A similar setup for the three-party case was used in [4], [5], [8], [9], [48]. We model the above as functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 16) and all our proofs are cast in $\mathcal{F}_{\mathsf{setup}}$-hybrid model.

c) Basic Primitives: In our protocols, we make use of a *collision-resistant* hash function, denoted by $\mathsf{H}()$, to save communication. Also, we use a commitment scheme, denoted by $\mathsf{Com}()$, to boost the security of our constructions from abort to fairness. We defer the formal details of key setup, hash function, and the commitment scheme to §A.

We use real-world / ideal-world simulation based approach to prove the security of our constructions and the details appear in the full version of the paper [49].

III. Building Layer-I Primitives

In this section, we start with the sharing semantics that serve as the basis for all our primitives. The computation in each primitive is executed in shared fashion to obtain the privacy-preserving property.

A. Secret Sharing Semantics

We use three types of secret sharing, as detailed below.

a) $[\cdot]$-sharing: A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be $[\cdot]$-shared among servers P_1, P_2, if the servers P_1 and P_2 respectively hold the values $[\mathsf{v}]_1 \in \mathbb{Z}_{2^\ell}$ and $[\mathsf{v}]_2 \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = [\mathsf{v}]_1 + [\mathsf{v}]_2$.

b) $\langle\cdot\rangle$-sharing: A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $\langle\cdot\rangle$-shared among servers in $\mathcal{P}$, if
- there exist $[\lambda_\mathsf{v}]_1, [\lambda_\mathsf{v}]_2 \in \mathbb{Z}_{2^\ell}$ such that $\lambda_\mathsf{v} = [\lambda_\mathsf{v}]_1 + [\lambda_\mathsf{v}]_2$.
- P_0 holds $([\lambda_\mathsf{v}]_1, [\lambda_\mathsf{v}]_2)$, while P_i for $i \in \{1,2\}$ holds $([\lambda_\mathsf{v}]_i, \mathsf{v} + \lambda_\mathsf{v})$

c) $[\![\cdot]\!]$-sharing: A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be $[\![\cdot]\!]$-shared among servers in $\mathcal{P}$, if
- v is $\langle\cdot\rangle$-shared i.e. P_0 holds $([\alpha_\mathsf{v}]_1, [\alpha_\mathsf{v}]_2)$, while P_i for $i \in \{1,2\}$ holds $([\alpha_\mathsf{v}]_i, \beta_\mathsf{v})$ for $\alpha_\mathsf{v}, \beta_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ with $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ and $\alpha_\mathsf{v} = [\alpha_\mathsf{v}]_1 + [\alpha_\mathsf{v}]_2$
- additionally, there exists $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ such that P_1, P_2 hold γ_v, while P_0 holds $\beta_\mathsf{v} + \gamma_\mathsf{v}$.

The table below summarises the individual shares of the servers for the aforementioned secret sharings. $[\mathsf{v}]_i$, $\langle\mathsf{v}\rangle_i$ and $[\![\mathsf{v}]\!]_i$ respectively denote the ith share held by P_i for $[\mathsf{v}]$, $\langle\mathsf{v}\rangle$ and $[\![\mathsf{v}]\!]$.

	$[\mathsf{v}]$	$\langle\mathsf{v}\rangle$	$[\![\mathsf{v}]\!]$
P_0	$-$	$([\lambda_\mathsf{v}]_1, [\lambda_\mathsf{v}]_2)$	$([\alpha_\mathsf{v}]_1, [\alpha_\mathsf{v}]_2, \beta_\mathsf{v} + \gamma_\mathsf{v})$
P_1	$[\mathsf{v}]_1$	$([\lambda_\mathsf{v}]_1, \mathsf{v} + \lambda_\mathsf{v})$	$([\alpha_\mathsf{v}]_1, \beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}, \gamma_\mathsf{v})$
P_2	$[\mathsf{v}]_2$	$([\lambda_\mathsf{v}]_2, \mathsf{v} + \lambda_\mathsf{v})$	$([\alpha_\mathsf{v}]_2, \beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}, \gamma_\mathsf{v})$

TABLE III: Shares held by the parties under different sharings

d) Arithmetic and Boolean Sharing: We use the sharing over both $\mathbb{Z}_{2^\ell}$ and $\mathbb{Z}_{2^1}$ and refer them as *arithmetic* and respectively *boolean* sharing. The latter sharing is demarcated using a $\mathbf{B}$ in the superscript (e.g. $[\![\mathsf{b}]\!]^{\mathbf{B}}$).

e) Linearity of the secret sharing schemes: Given the $[\cdot]$-sharing of x, y and public constants c_1, c_2, servers can locally compute $[c_1\mathsf{x} + c_2\mathsf{y}]$ as $c_1[\mathsf{x}] + c_2[\mathsf{y}]$. Notice that linearity trivially extends to the case of $\langle\cdot\rangle$-sharing and $[\![\cdot]\!]$-sharing as well. Linearity allows the servers to perform the following operations *non-interactively*: i) addition of two shared values and ii) multiplication of the shared value with a public constant.

B. Secret Sharing and Reconstruction protocols

We dedicate this section to describe some of the secret sharing and reconstruction protocols that we need. A detailed communication complexity analysis of all the constructs appear in the full version of the paper [49].

a) Sharing Protocol: Protocol Π_{sh} (Fig. 2) enables server P_i to generate $[\![\cdot]\!]$-sharing of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. During the preprocessing phase, servers P_0, P_1 along with P_i together sample random value $[\alpha_\mathsf{v}]_1$, while servers P_0, P_2 and P_i sample $[\alpha_\mathsf{v}]_2$ using the shared randomness. This enables server P_i to obtain the entire α_v. Also, servers P_1, P_2 together sample a random $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$. For the case when $P_i = P_0$, we optimize the protocol by making P_0 sample the γ_v value along with P_1, P_2. This eliminates the need for servers P_1, P_2 to send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0 during the online phase. Furthermore, the sharing does not need to hide the input from P_0 (who is the input contributor) by keeping γ_v private.

During the online phase, P_i computes β_v and sends it to P_1, P_2 who then verify the sanity of the received value by

exchanging its hash with the fellow recipient. To complete the $[\![\cdot]\!]$-sharing, P_1 sends $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0 while P_2 sends a hash of the same to P_0, who aborts if the received values mismatch.

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

Preprocessing:

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

Online:

- P_i computes $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ and sends to both P_1 and P_2.
- If $P_i = P_0$, servers P_1, P_2 mutually exchange $\mathsf{H}(\beta_{\mathsf{v}})$ and `abort` if there is a mismatch.
- If $P_i \neq P_0$, P_1 computes and sends $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ to P_0 while P_2 sends a hash of the same to P_0, who `abort` if the received values are inconsistent.

Fig. 2: $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ by server P_i

In the outsourced setting, input sharing is performed by the parties and not the servers. Concretely, for the case of ML training, data owners perform the input sharing while for the case of ML inference, input sharing is performed by the model owner and the client. For a party P to perform the input sharing of value v, server P_j for $j \in \{1, 2\}$ sends $[\alpha_{\mathsf{v}}]_j$ to P while P_0 sends a hash of the same to P. Party P computes $\alpha_{\mathsf{v}} = [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$ if the received values are consistent and `abort` otherwise. P then computes $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ and sends to both P_1 and P_2. The rest of the protocol proceeds similar to Π_{sh} where servers P_1, P_2 mutually exchanges the hash of β_{v} and verifies the consistency of β_{v}.

b) Joint Sharing Protocol: Protocol $\Pi_{\mathsf{jsh}}(P_i, P_j, \mathsf{v})$ (Fig. 3) enables servers P_i, P_j (an unordered pair) to jointly generate $[\![\cdot]\!]$-sharing of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, known to both of them. Towards this, server P_i executes protocol Π_{sh} on the value v to generate its $[\![\cdot]\!]$-sharing. Server P_j helps in verifying the correctness of the sharing performed by P_i.

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

- If $(P_i, P_j) = (P_1, P_0)$: Server P_1 executes protocol $\Pi_{\mathsf{sh}}(P_1, \mathsf{v})$. P_0 computes $\beta_{\mathsf{v}} = \mathsf{v} + [\alpha_{\mathsf{v}}]_1 + [\alpha_{\mathsf{v}}]_2$. P_0 then sends $\mathsf{H}(\beta_{\mathsf{v}})$ to P_2 who `aborts` if the received value is inconsistent with the same received from P_1.
- If $(P_i, P_j) = (P_2, P_0)$: Similar to the case above.
- If $(P_i, P_j) = (P_1, P_2)$: During the preprocessing phase, P_1, P_2 together sample random $\gamma_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$. Servers set $[\alpha_{\mathsf{v}}]_1 = [\alpha_{\mathsf{v}}]_2 = 0$ and $\beta_{\mathsf{v}} = \mathsf{v}$. P_1 computes and sends $\mathsf{v} + \gamma_{\mathsf{v}}$ to P_0 while P_2 sends corresponding hash to P_0, who `aborts` if the received values are inconsistent.

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

Protocol Π_{jsh} can be made non-interactive for the case when the value v is available to both P_i and P_j in the preprocessing phase. Towards this, servers in $\mathcal{P}$ sample random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and locally set their shares as described in Table IV.

Looking ahead, protocol Π_{jsh} offers tolerance against one active corruption, leveraging the fact that the secret to be shared is available amongst two servers, with one of them is guaranteed to be honest.

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

TABLE IV: The columns consider the three distinct possibility of input contributing pairs. The first row shows the assignment to various components of the sharing. The last row (with three sub-rows) specifies the shares held by the three servers.

c) Reconstruction Protocol: Protocol $\Pi_{\mathsf{rec}}(\mathcal{P}, [\![\mathsf{v}]\!])$ (Fig. 4) enables servers in $\mathcal{P}$ to reconstruct the secret v from its $[\![\cdot]\!]$-sharing. Towards this, each server receives her missing share from one of the other two servers and the hash of the same from the third one. If the received values are consistent, the server proceeds with the reconstruction and otherwise, it aborts. Reconstruction towards a single server P_i can be viewed as a special case of this protocol and we use $\Pi_{\mathsf{rec}}(P_i, \mathsf{v})$ to denote the same.

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

Online:

- P_1 receives $[\alpha_{\mathsf{v}}]_2$ and $\mathsf{H}([\alpha_{\mathsf{v}}]_2)$ from P_2 and P_0 respectively.
- P_2 receives $[\alpha_{\mathsf{v}}]_1$ and $\mathsf{H}([\alpha_{\mathsf{v}}]_1)$ from P_0 and P_1 respectively.
- P_0 receives β_{v} and $\mathsf{H}(\beta_{\mathsf{v}})$ from P_1 and P_2 respectively.

Server P_i for $i \in \{0, 1, 2\}$ `abort` if the received values are inconsistent. Else computes $\mathsf{v} = \beta_{\mathsf{v}} - [\alpha_{\mathsf{v}}]_1 - [\alpha_{\mathsf{v}}]_2$.

Fig. 4: Reconstruction of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ among servers in $\mathcal{P}$

In the outsourced setting where reconstruction happens towards the parties (data owners for ML training and client for ML inference), the servers will send their shares towards the parties directly. To reconstruct a value v towards party P, servers P_0, P_1 and P_2 sends $([\![\alpha_{\mathsf{v}}]\!]^{\mathbf{A}}, \mathsf{H}([\![\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}))$, $(\beta_{\mathsf{v}}, \mathsf{H}([\![\alpha_{\mathsf{v}}]\!]^{\mathbf{A}}))$ and $([\![\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}, \mathsf{H}(\beta_{\mathsf{v}}))$ respectively to P. Party P will accept the shares if the corresponding hash match and `abort` otherwise.

d) Fair Reconstruction Protocol: The security goal of fairness is well-motivated. Consider an outsourced setting where a machine learning service that is instantiated with a protocol *with abort* is offered against payment. Here, during the reconstruction of output, adversary can instruct the corrupt server to send inconsistent values (either shares or hash values) to honest parties and make them `abort`. At the same time, adversary will learn the output from the honest shares received on behalf of the corrupt parties. This leads to a situation where some parties who have control over the corrupt server obtain the protocol output, while the other honest parties obtain nothing. This is a strong deterrent for the honest parties to participate in the protocol in the future. On the other hand, a system with fairness property guarantees that the honest parties will get the output whenever the corrupt parties gets the output. In our 3PC setting, the presence of at least a single honest server ensures that all the participating honest parties

will eventually get the output. This will attract more people to participate in the protocol and is crucial to applications like ML training where more data leads to a better-trained model.

We use the techniques proposed by ASTRA [48] to achieve fairness and modify it for our sharing scheme. We defer formal details to §C.

C. Layer-I Primitives

We are now ready to describe our Layer-I primitives– Multiplication, Bit Extraction, and Bit2A.

a) Multiplication Protocol: Protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$ (Fig. 5) enables the servers in $\mathcal{P}$ to compute $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{xy}$, given the $[\![\cdot]\!]$-sharing of x and y. We begin with a protocol for the semi-honest setting, which is a slightly modified variant of the protocol proposed by ASTRA. 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

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

holds, servers P_1, P_2 locally compute $[\beta_\mathsf{z}]_j = (j-1)\beta_\mathsf{x}\beta_\mathsf{y} - \beta_x[\alpha_\mathsf{y}]_j - \beta_y[\alpha_\mathsf{x}]_j + [\Gamma_{\mathsf{xy}}]_j + [\alpha_\mathsf{z}]_j$ during the online phase and mutually exchange their shares to reconstruct β_z. Server P_1 then computes and 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 Equation 1.

In the malicious setting, we observe that the aforementioned protocol suffers from three issues:

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

While the first two issues in the above list are inherited from the protocol of ASTRA, the third one is due to our new sharing semantics (compared to ASTRA where γ_v and $\beta_\mathsf{v} + \gamma_\mathsf{v}$ were not part of the shares) that imposes an additional component of $\beta_\mathsf{z} + \gamma_\mathsf{z}$ held by P_0. We begin with solving the last issue first. In order to verify the correctness of $\beta_\mathsf{z} + \gamma_\mathsf{z}$ sent by P_1, server P_2 computes a hash of the same and send it to P_0, who `aborts` if the received values are inconsistent.

For the remaining two issues, though they are quite distinct in nature, we make use of the asymmetric roles played by the servers $\{P_0\}$ and $\{P_1, P_2\}$ to introduce a single check that solves both the issues at the same time. Though the check is inspired from the protocol of ASTRA, our technical innovation lies in the way in which the check is performed. In ASTRA, servers first execute the semi-honest protocol and the correctness of the computation is verified with the help of $\langle\cdot\rangle$-sharing of a multiplication triple generated in the preprocessing phase. Unlike ASTRA, we perform a single multiplication (and nothing additional) in the preprocessing phase to generate the correct preprocessing data required for a multiplication gate in the online phase. This brings down the communication in the preprocessing phase drastically from 21 ring elements to 3 ring elements. The details of our method are provided next.

To solve the second issue, where a corrupt P_1 (or P_2) sends an incorrect $[\cdot]$-share of β_z, we make use of server P_0 as follows: Using the values $\beta^\star_\mathsf{x} = \beta_\mathsf{x} + \gamma_\mathsf{x}$ and $\beta^\star_\mathsf{y} = \beta_\mathsf{y} + \gamma_\mathsf{y}$, P_0 computes $\beta^\star_\mathsf{z} = -\beta^\star_\mathsf{x}\alpha_\mathsf{y} - \beta^\star_\mathsf{y}\alpha_\mathsf{x} + 2\Gamma_{\mathsf{xy}} + \alpha_\mathsf{z}$. Now $\beta^\star_\mathsf{z}$ can be written as below:

$$\begin{aligned}\beta^\star_\mathsf{z} &= -\beta^\star_\mathsf{x}\alpha_\mathsf{y} - \beta^\star_\mathsf{y}\alpha_\mathsf{x} + 2\Gamma_{\mathsf{xy}} + \alpha_\mathsf{z} \\ &= -(\beta_\mathsf{x} + \gamma_\mathsf{x})\alpha_\mathsf{y} - (\beta_\mathsf{y} + \gamma_\mathsf{y})\alpha_\mathsf{x} + 2\Gamma_{\mathsf{xy}} + \alpha_\mathsf{z} \\ &= (-\beta_\mathsf{x}\alpha_\mathsf{y} - \beta_\mathsf{y}\alpha_\mathsf{x} + \Gamma_{\mathsf{xy}} + \alpha_\mathsf{z}) - (\gamma_\mathsf{x}\alpha_\mathsf{y} + \gamma_\mathsf{y}\alpha_\mathsf{x} - \Gamma_{\mathsf{xy}}) \\ &= (\beta_\mathsf{z} - \beta_\mathsf{x}\beta_\mathsf{y}) - (\gamma_\mathsf{x}\alpha_\mathsf{y} + \gamma_\mathsf{y}\alpha_\mathsf{x} - \Gamma_{\mathsf{xy}} + \psi) + \psi \quad \text{[Equation 1]} \\ &= (\beta_\mathsf{z} - \beta_\mathsf{x}\beta_\mathsf{y} + \psi) - \chi \quad \text{[where } \chi = \gamma_\mathsf{x}\alpha_\mathsf{y} + \gamma_\mathsf{y}\alpha_\mathsf{x} - \Gamma_{\mathsf{xy}} + \psi\text{]}\end{aligned}$$

Assuming that (a) $\psi \in \mathbb{Z}_{2^\ell}$ is a random value sampled together by P_1 and P_2 (and unknown to P_0) and (b) P_0 knows the value χ, P_0 can send $\beta^\star_\mathsf{z} + \chi$ to P_1 and P_2 who using the knowledge of $\beta_\mathsf{x}, \beta_\mathsf{y}$ and ψ can verify the correctness of β_z by computing $\beta_\mathsf{z} - \beta_\mathsf{x}\beta_\mathsf{y} + \psi$ and checking against the value $\beta^\star_\mathsf{z} + \chi$ received from P_0. Now we describe how to enable P_0 to obtain the value χ. Note that server P_j for $j \in \{1,2\}$ can locally compute $[\chi]_j = \gamma_\mathsf{x}[\alpha_\mathsf{y}]_j + \gamma_\mathsf{y}[\alpha_\mathsf{x}]_j - [\Gamma_{\mathsf{xy}}]_j + [\psi]_j$ where $[\psi]_j$ can be generated non-interactively by P_1, P_2 using shared randomness. P_1, P_2 can then send their $[\cdot]$-shares of χ to P_0 to enable him obtain the value χ. To verify if P_0 computed χ correctly, we leverage the following relation. The values $\mathsf{d} = \gamma_\mathsf{x} - \alpha_\mathsf{x}, \mathsf{e} = \gamma_\mathsf{y} - \alpha_\mathsf{y}$ and $\mathsf{f} = (\gamma_\mathsf{x}\gamma_\mathsf{y} + \psi) - \chi$ should satisfy $\mathsf{f} = \mathsf{de}$ if and only if χ is correctly computed, because:

$$\begin{aligned}\mathsf{de} &= (\gamma_\mathsf{x} - \alpha_\mathsf{x})(\gamma_\mathsf{y} - \alpha_\mathsf{y}) = \gamma_\mathsf{x}\gamma_\mathsf{y} - \gamma_\mathsf{x}\alpha_\mathsf{y} - \gamma_\mathsf{y}\alpha_\mathsf{x} + \Gamma_{\mathsf{xy}} \\ &= (\gamma_\mathsf{x}\gamma_\mathsf{y} + \psi) - (\gamma_\mathsf{x}\alpha_\mathsf{y} + \gamma_\mathsf{y}\alpha_\mathsf{x} - \Gamma_{\mathsf{xy}} + \psi) \\ &= (\gamma_\mathsf{x}\gamma_\mathsf{y} + \psi) - \chi = \mathsf{f}\end{aligned}$$

Therefore, the correctness of χ reduces to verifying if the triple $(\mathsf{d}, \mathsf{e}, \mathsf{f})$ is a multiplication triple or not. Interestingly, the same check suffices to resolve the first issue of corrupt P_0 generating incorrect $[\Gamma_{\mathsf{xy}}]$-sharing. This is because, if P_0 would have shared $\Gamma_{\mathsf{xy}} + \Delta$ where Δ denotes the error introduced, then $\mathsf{de} = \mathsf{f} + \Delta \neq \mathsf{f}$.

Equipped with the aforementioned observations (Table V), our final trick, that distinguishes BLAZE's multiplication from that of ASTRA's, is to compute a $\langle\cdot\rangle$-sharing of f starting with $\langle\cdot\rangle$-sharing of d, e using the efficient maliciously secure multiplication protocol of [17] referred to as Π_{mulZK} henceforth and described in §B for completeness, and extract out the values for $\Gamma_{\mathsf{xy}}, \psi$ and χ from f which are bound to be correct. This can be executed entirely in the preprocessing phase. Protocol Π_{mulZK} works over $\langle\cdot\rangle$-sharing (§III-A), which is why this part our computation is done over this type of sharing, and requires a per party communication of 1 ring element, when amortized over large circuits (ref. Theorem 1.4 of [17][1]). Concretely, given the $[\![\cdot]\!]$-sharing of the inputs x and y of the multiplication protocol, servers locally compute $\langle\cdot\rangle$-sharing of values d and e as follows. (The sharing semantics for $[\mathsf{v}]$ for any v is recalled below.)

Upon executing protocol $\Pi_{\mathsf{mulZK}}(\mathcal{P}, \mathsf{d}, \mathsf{e})$, servers obtain $\langle\mathsf{f}\rangle = ([\lambda_\mathsf{f}], \mathsf{f} + \lambda_\mathsf{f})$. To be precise, P_0 obtains $([\lambda_\mathsf{f}]_1, [\lambda_\mathsf{f}]_2)$

[1] https://eprint.iacr.org/2019/188

$\langle \mathsf{v} \rangle$	P_0 $([\lambda_\mathsf{v}]_1, [\lambda_\mathsf{v}]_1)$	P_1 $([\lambda_\mathsf{v}]_1, \mathsf{v}+\lambda_\mathsf{v})$	P_2 $([\lambda_\mathsf{v}]_2, \mathsf{v}+\lambda_\mathsf{v})$
$\langle \mathsf{d} \rangle$	$([\alpha_\mathsf{x}]_1, [\alpha_\mathsf{x}]_2)$	$([\alpha_\mathsf{x}]_1, \gamma_\mathsf{x})$	$([\alpha_\mathsf{x}]_2, \gamma_\mathsf{x})$
$\langle \mathsf{e} \rangle$	$([\alpha_\mathsf{y}]_1, [\alpha_\mathsf{y}]_1)$	$([\alpha_\mathsf{y}]_1, \gamma_\mathsf{y})$	$([\alpha_\mathsf{y}]_2, \gamma_\mathsf{y})$

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

while P_j for $j \in \{1,2\}$ obtains $([\lambda_\mathsf{f}]_j, \mathsf{f}+\lambda_\mathsf{f})$. Servers then map the values $[\chi]$ and $\gamma_\mathsf{x}\gamma_\mathsf{y}+\psi$ to $[\lambda_\mathsf{f}]$ and $\mathsf{f}+\lambda_\mathsf{f}$ respectively followed by extracting the required values as:

$$[\chi]_1 = [\lambda_\mathsf{f}]_1 \text{ and } [\chi]_2 = [\lambda_\mathsf{f}]_2 \quad \rightarrow \chi = [\lambda_\mathsf{f}]_1 + [\lambda_\mathsf{f}]_2$$
$$\gamma_\mathsf{x}\gamma_\mathsf{y} + \psi = \mathsf{f} + \lambda_\mathsf{f} \quad \rightarrow \psi = \mathsf{f} + \lambda_\mathsf{f} - \gamma_\mathsf{x}\gamma_\mathsf{y}$$
$$[\Gamma_{\mathsf{xy}}]_j = \gamma_\mathsf{x}[\alpha_\mathsf{y}]_j + \gamma_\mathsf{y}[\alpha_\mathsf{x}]_j + [\psi]_j - [\chi]_j \quad [j \in \{1,2\}]$$

where $[\psi]$ is generated non-interactively by servers P_1, P_2 by sampling a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ together and setting $[\psi]_1 = \mathsf{r}$ and $[\psi]_2 = \psi - \mathsf{r}$. We claim that after extracting the values as mentioned above, servers P_1, P_2 hold $[\Gamma_{\mathsf{xy}}] = [\alpha_\mathsf{x}\alpha_\mathsf{y}]$. To see this, note that

$$\begin{aligned}\Gamma_{\mathsf{xy}} &= \gamma_\mathsf{x}\alpha_\mathsf{y} + \gamma_\mathsf{y}\alpha_\mathsf{x} + \psi - \chi \\ &= (\mathsf{d}+\lambda_\mathsf{d})\lambda_\mathsf{e} + (\mathsf{e}+\lambda_\mathsf{e})\lambda_\mathsf{d} + (\mathsf{f}+\lambda_\mathsf{f}-\gamma_\mathsf{x}\gamma_\mathsf{y}) - \lambda_\mathsf{f} \\ &= (\mathsf{d}+\lambda_\mathsf{d})(\mathsf{e}+\lambda_\mathsf{e}) - \mathsf{de} + \lambda_\mathsf{d}\lambda_\mathsf{e} + (\mathsf{f}-\gamma_\mathsf{x}\gamma_\mathsf{y}) \\ &= \gamma_\mathsf{x}\gamma_\mathsf{y} - \mathsf{f} + \lambda_\mathsf{d}\lambda_\mathsf{e} + (\mathsf{f}-\gamma_\mathsf{x}\gamma_\mathsf{y}) = \lambda_\mathsf{d}\lambda_\mathsf{e} = \alpha_\mathsf{x}\alpha_\mathsf{y}\end{aligned}$$

This concludes the informal discussion.

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

Preprocessing:

- Servers P_0, P_j for $j \in \{1,2\}$ together sample a random $[\alpha_\mathsf{z}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample a random $\gamma_\mathsf{z} \in \mathbb{Z}_{2^\ell}$.
- Servers in $\mathcal{P}$ locally compute $\langle\cdot\rangle$-sharing of $\mathsf{d} = \gamma_\mathsf{x} - \alpha_\mathsf{x}$ and $\mathsf{e} = \gamma_\mathsf{y} - \alpha_\mathsf{y}$ by setting the shares as (as per Table V):

$$[\lambda_\mathsf{d}]_1 = [\alpha_\mathsf{x}]_1, \quad [\lambda_\mathsf{d}]_2 = [\alpha_\mathsf{x}]_2, \quad (\mathsf{d}+\lambda_\mathsf{d}) = \gamma_\mathsf{x}$$
$$[\lambda_\mathsf{e}]_1 = [\alpha_\mathsf{y}]_1, \quad [\lambda_\mathsf{e}]_2 = [\alpha_\mathsf{y}]_2, \quad (\mathsf{e}+\lambda_\mathsf{e}) = \gamma_\mathsf{y}$$

- Servers in $\mathcal{P}$ execute $\Pi_{\mathsf{mulZK}}(\mathcal{P}, \mathsf{d}, \mathsf{e})$ to generate $\langle \mathsf{f} \rangle = \langle \mathsf{de} \rangle$.
- P_0, P_j for $j \in \{1,2\}$ locally set $[\chi]_j = [\lambda_\mathsf{f}]_j$, while P_1, P_2 set $\psi = \mathsf{f} + \lambda_\mathsf{f} - \gamma_\mathsf{x}\gamma_\mathsf{y}$. P_0 then computes $\chi = [\chi]_1 + [\chi]_2$.
- P_1, P_2 sample random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and set $[\psi]_1 = \mathsf{r}, [\psi]_2 = \psi - \mathsf{r}$.
- P_j for $j \in \{1,2\}$ set $[\Gamma_{\mathsf{xy}}]_j = \gamma_\mathsf{x}[\alpha_\mathsf{y}]_j + \gamma_\mathsf{y}[\alpha_\mathsf{x}]_j + [\psi]_j - [\chi]_j$

Online:

- P_j for $j \in \{1,2\}$ computes and exchanges $[\beta_\mathsf{z}]_j = (j-1)\beta_\mathsf{x}\beta_\mathsf{y} - \beta_\mathsf{x}[\alpha_\mathsf{y}]_j - \beta_\mathsf{y}[\alpha_\mathsf{x}]_j + [\Gamma_{\mathsf{xy}}]_j + [\alpha_\mathsf{z}]_j$ to reconstruct $\beta_\mathsf{z} = [\beta_\mathsf{z}]_1 + [\beta_\mathsf{z}]_2$.
- P_0 computes $\beta_\mathsf{z}^\star = -(\beta_\mathsf{x}+\gamma_\mathsf{x})\alpha_\mathsf{y} - (\beta_\mathsf{y}+\gamma_\mathsf{y})\alpha_\mathsf{x} + \alpha_\mathsf{z} + 2\Gamma_{\mathsf{xy}} + \chi$ and sends $\mathsf{H}(\beta_\mathsf{z}^\star)$ to both P_1 and P_2.
- P_j for $j \in \{1,2\}$ `aborts` if $\mathsf{H}(\beta_\mathsf{z} - \beta_\mathsf{x}\beta_\mathsf{y} + \psi) \neq \mathsf{H}(\beta_\mathsf{z}^\star)$.
- P_1 sends $\beta_\mathsf{z} + \gamma_\mathsf{z}$ and P_2 sends the $\mathsf{H}(\beta_\mathsf{z} + \gamma_\mathsf{z})$ to P_0. P_0 will `abort` if it receives inconsistent values.

Fig. 5: Multiplication Protocol

Looking ahead, our multiplication protocol lends its technical strength to all our layer-II primitives, especially the dot product. The preprocessing phase of dot product protocol invokes its preprocessing phase in a black-box way many times that lets its optimal complexity (of 3 elements per multiplication) kick in. However, the online phase of dot product is not plain invocation of online phase of multiplication protocol in a black-box way. In fact, the tweaks here are crucial for achieving a complexity that is independent of the feature length. The other layer-II primitives use multiplication protocol in a block-box way.

b) Bit Extraction Protocol: Protocol $\Pi_{\mathsf{bitext}}(\mathcal{P}, [\![\mathsf{v}]\!])$ (Fig. 6) enables servers in $\mathcal{P}$ to compute the boolean sharing ($[\![\cdot]\!]^{\mathbf{B}}$) of most significant bit (msb) of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, given its arithmetic sharing $[\![\mathsf{v}]\!]$. The first approach is to use an optimized Parallel Prefix Adder (PPA) proposed by ABY3 [5]. The PPA circuit consists of 2ℓ AND gates and has a multiplicative depth of $\log(\ell)$. We refer readers to ABY3 for more details. The next approach is to use a garbled circuit that results in a constant round solution. We provide details for the latter approach below.

Let $GC = (\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3, \mathsf{u}_4, \mathsf{u}_5)$ denote a garbled circuit with inputs $\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3 \in \mathbb{Z}_{2^\ell}$ and $\mathsf{u}_4, \mathsf{u}_5 \in \{0,1\}$ and output $\mathsf{y} = \mathsf{msb}(\mathsf{u}_1 - \mathsf{u}_2 - \mathsf{u}_3) \oplus \mathsf{u}_4 \oplus \mathsf{u}_5$. Note that the MSB calculation portion of the circuit can be instantiated using the optimized PPA of ABY3. Let $\mathsf{u}_1 = \beta_\mathsf{v}, \mathsf{u}_2 = [\alpha_\mathsf{v}]_1$ and $\mathsf{u}_3 = [\alpha_\mathsf{v}]_2$ so that $\mathsf{u}_1 - \mathsf{u}_2 - \mathsf{u}_3 = \mathsf{v}$. Let $\mathsf{u}_4 = \mathsf{r}_1, \mathsf{u}_5 = \mathsf{r}_2$ where r_1 and r_2 denote random bits sampled by P_0, P_1 and P_0, P_2 respectively.

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

Let $GC = (\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3, \mathsf{u}_4, \mathsf{u}_5)$ denote a garbled circuit with inputs $\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3 \in \mathbb{Z}_{2^\ell}$ and $\mathsf{u}_4, \mathsf{u}_5 \in \{0,1\}$ and output $\mathsf{y} = \mathsf{msb}(\mathsf{u}_1 - \mathsf{u}_2 - \mathsf{u}_3) \oplus \mathsf{u}_4 \oplus \mathsf{u}_5$. Let $\mathsf{u}_1 = \beta_\mathsf{v}, \mathsf{u}_2 = [\alpha_\mathsf{v}]_1$ and $\mathsf{u}_3 = [\alpha_\mathsf{v}]_2$ so that $\mathsf{u}_1 - \mathsf{u}_2 - \mathsf{u}_3 = \mathsf{v}$.

Preprocessing:

- P_0, P_j for $j \in \{1,2\}$ sample random $\mathsf{r}_j \in \{0,1\}$ and execute Π_{jsh} on r_j to generate $[\![\mathsf{r}_j]\!]^{\mathbf{B}}$. Let $\mathsf{u}_4 = \mathsf{r}_1$ and $\mathsf{u}_5 = \mathsf{r}_2$.
- P_0, P_1 garbles the circuit GC and sends GC to P_2 along with the decoding information. Note that the values u_2 and u_4 are embedded in the GC itself since they are known to P_0, P_1.
- Corresponding to each bit of u_3, P_0, P_1 compute commitments for both the keys (zero key and one key) using common randomness and send these commitments to P_2. In addition, P_0 sends the decommitment of actual key for the bit to P_2 who `abort` if the values are inconsistent. Similar steps are executed for the bit $\mathsf{u}_5 = \mathsf{r}_2$.

Online:

- P_0, P_1 compute commitments for both the keys corresponding to the bits of u_1 similar to the case of u_3. P_1 opens the right commitment towards P_2.
- P_2 evaluates GC and obtains $\mathsf{v} = \mathsf{msb}(\mathsf{v}) \oplus \mathsf{r}_1 \oplus \mathsf{r}_2$ in clear. P_2 sends v to P_1 along with a hash of the key corresponding to v. P_1 `abort` if the received values are inconsistent.
- P_1, P_2 execute Π_{jsh} on v to generate $[\![\mathsf{v}]\!]^{\mathbf{B}}$. Servers locally compute $[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}} = [\![\mathsf{v}]\!]^{\mathbf{B}} \oplus [\![\mathsf{r}_1]\!]^{\mathbf{B}} \oplus [\![\mathsf{r}_2]\!]^{\mathbf{B}}$.

Fig. 6: Extraction of MSB bit of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$

On a high level, protocol proceeds as follows: P_0, P_1 garbles the circuit GC and send GC to P_2 along with the keys corresponding to the inputs and the decoding information. P_2 upon evaluating GC obtains $\mathsf{v} = \mathsf{msb}(\mathsf{v}) \oplus \mathsf{r}_1 \oplus \mathsf{r}_2$ in clear and sends v along with a hash of the actual key corresponding to v to P_1. P_1, P_2 then jointly generate $[\![\mathsf{v}]\!]^{\mathbf{B}}$. Servers then XOR $[\![\mathsf{v}]\!]^{\mathbf{B}}$ to $[\![\mathsf{r}_1]\!]^{\mathbf{B}}$ and $[\![\mathsf{r}_2]\!]^{\mathbf{B}}$ that are generated in the preprocessing phase to obtain the final result.

c) Bit2A: Protocol $\Pi_{\mathsf{bit2A}}(\mathcal{P}, [\![\mathsf{b}]\!]^{\mathbf{B}})$ (Fig. 7) enables servers in $\mathcal{P}$ to compute the arithmetic sharing of a single bit b, given its $[\![\cdot]\!]^{\mathbf{B}}$-sharing. We denote the value of bit b in the ring $\mathbb{Z}_{2^\ell}$ as $(\mathsf{b})^{\mathbf{A}}$. Now observing that $(\mathsf{b})^{\mathbf{A}} = (\beta_{\mathsf{b}} \oplus \alpha_{\mathsf{b}})^{\mathbf{A}} = (\beta_{\mathsf{b}})^{\mathbf{A}} + (\alpha_{\mathsf{b}})^{\mathbf{A}} - 2(\beta_{\mathsf{b}})^{\mathbf{A}}(\alpha_{\mathsf{b}})^{\mathbf{A}}$, we compute an arithmetic sharing of $(\beta_{\mathsf{b}})^{\mathbf{A}}$, $(\alpha_{\mathsf{b}})^{\mathbf{A}}$ and their product $(\beta_{\mathsf{b}})^{\mathbf{A}}(\alpha_{\mathsf{b}})^{\mathbf{A}}$ to obtain arithmetic sharing of $(\mathsf{b})^{\mathbf{A}}$. To compute an arithmetic sharing of $(\alpha_b)^{\mathbf{A}}$, we use $(\alpha_b)^{\mathbf{A}} = ([\alpha_b]_1 \oplus [\alpha_b]_2)^{\mathbf{A}} = ([\alpha_b]_1)^{\mathbf{A}} + ([\alpha_b]_2)^{\mathbf{A}} - 2([\alpha_b]_1)^{\mathbf{A}}([\alpha_b]_2)^{\mathbf{A}}$ and compute an arithmetic sharing of $([\alpha_b]_1)^{\mathbf{A}}$, $([\alpha_b]_2)^{\mathbf{A}}$ and their product $([\alpha_b]_1)^{\mathbf{A}}([\alpha_b]_2)^{\mathbf{A}}$ as follows. P_0, P_j for $j \in \{1,2\}$ execute Π_{jsh} on $([\alpha_b]_j)^{\mathbf{A}}$ to generate $[\![([\alpha_b]_j)^{\mathbf{A}}]\!]$. Servers then execute Π_{mult} on $[\![([\alpha_b]_1)^{\mathbf{A}}]\!]$ and $[\![([\alpha_b]_2)^{\mathbf{A}}]\!]$ to generate $[\![([\alpha_b]_1)^{\mathbf{A}}([\alpha_b]_2)^{\mathbf{A}}]\!]$, followed by locally computing the result. The computation of $[\![(\mathsf{b})^{\mathbf{A}}]\!]$ follows similarly.

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

Preprocessing:

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

Online:

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

Fig. 7: Bit2A Protocol

IV. Building Layer-II Primitives

Since ML algorithms involve operating over decimals, we use signed two's complement form [2], [5], [48] over the ring $\mathbb{Z}_{2^\ell}$ to represent the decimal numbers. Here, the most significant bit (msb) denotes the sign and the last d bits are reserved for the fractional part. We choose $\ell = 64$ and $d = 13$, which leaves 50 bits for the integer part. The ℓ-bit strings are treated as elements of $\mathbb{Z}_{2^\ell}$. A product of two numbers from this domain requires d to be 26 bits if we do not want to compromise on the accuracy. However, for training tasks which require many sequential multiplications, this might lead to an overflow. Hence, a method for truncation is required in order to cast the product result back in the aforementioned format. Also, typically ML algorithms perform multiplication in the form of dot product. We present below protocols for– (a) dot product, (b) truncation, (c) dot product with truncation, (d) secure comparison, and (e) non-linear activation functions.

a) Dot Product: Protocol Π_{dotp} (Fig. 8) enables servers in $\mathcal{P}$ to generate $[\![\cdot]\!]$-sharing of $\vec{\mathsf{x}} \odot \vec{\mathsf{y}}$, given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$. By $[\![\cdot]\!]$-sharing of a vector $\vec{\mathsf{x}}$ of size n, we mean each element $\mathsf{x}_i \in \mathbb{Z}_{2^\ell}$ of it, for $i \in [\mathsf{n}]$, is $[\![\cdot]\!]$-shared. A naive solution is to view the problem as n instances of Π_{mult}, where the i^{th} instance computes $\mathsf{z}_i = \mathsf{x}_i \cdot \mathsf{y}_i$. The final result can then be obtained by locally adding the shares of z_i corresponding to all the instances. But this would require a communication that is linearly dependent on the size of the vectors (i.e. n). We make the communication of Π_{dotp} in the online phase independent of n as follows: Instead of reconstructing each β_{z_i} separately to compute β_{z} with $\mathsf{z} = \vec{\mathsf{x}} \odot \vec{\mathsf{y}}$, P_1, P_2 locally compute $[\beta_{\mathsf{z}}] = [\beta_{\mathsf{z}_1}] + \ldots + [\beta_{\mathsf{z}_\mathsf{n}}]$ and reconstruct β_{z}. Moreover, instead of sending $\beta^\star_{\mathsf{z}_i}$ for each $\mathsf{z}_i = \mathsf{x}_i \cdot \mathsf{y}_i$, P_0 can "combine" all the $\beta^\star_{\mathsf{z}_i}$ values and send a single $\beta^\star_{\mathsf{z}}$ to P_1, P_2 for verification. In detail, P_0 computes $\beta^\star_{\mathsf{z}} = \sum_{i=1}^{\mathsf{n}} \beta^\star_{\mathsf{z}_i}$ and sends a hash of the same to both P_1 and P_2, who then can cross check with a hash of $\beta_{\mathsf{z}} - \sum_{i=1}^{\mathsf{n}} (\beta_{\mathsf{x}_i} \cdot \beta_{\mathsf{y}_i} - \psi_i)$.

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

Preprocessing:

- Servers in $\mathcal{P}$ execute preprocessing phase of $\Pi_{\mathsf{mult}}(\mathcal{P}, \mathsf{x}_i, \mathsf{y}_i)$ for each pair $(\mathsf{x}_i, \mathsf{y}_i)$ where $i \in [\mathsf{n}]$ and $\mathsf{z}_i = \mathsf{x}_i\mathsf{y}_i$. P_0 obtains χ_i, while P_j, for $j \in \{1,2\}$, obtains $[\Gamma_{\mathsf{x}_i\mathsf{y}_i}]_j$ and ψ_i.
- P_0 computes $\chi = \sum_{i=1}^{\mathsf{n}} \chi_i, \Gamma_{\mathsf{xy}} = \sum_{i=1}^{\mathsf{n}} \Gamma_{\mathsf{x}_i\mathsf{y}_i}$, while P_j for $j \in \{1,2\}$ computes $[\Gamma_{\mathsf{xy}}]_j = \sum_{i=1}^{\mathsf{n}} [\Gamma_{\mathsf{x}_i\mathsf{y}_i}]_j, \psi = \sum_i \psi_i$.
- P_0, P_j for $j \in \{1,2\}$ compute $[\alpha_{\mathsf{z}}]_j = \sum_{i=1}^{\mathsf{n}} [\alpha_{\mathsf{z}_i}]_j$.

Online:

- P_j for $j \in \{1,2\}$ computes $[\beta_{\mathsf{z}}]_j = \sum_{i=1}^{\mathsf{n}} ((j-1)\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i} - \beta_{\mathsf{x}_i}[\alpha_{\mathsf{y}_i}]_j - \beta_{\mathsf{y}_i}[\alpha_{\mathsf{x}_i}]_j) + [\Gamma_{\mathsf{xy}}]_j + [\alpha_{\mathsf{z}}]_j$ and mutually exchanges $[\beta_{\mathsf{z}}]_j$ to reconstruct β_{z}.
- P_0 computes $\beta^\star_{\mathsf{z}} = -\sum_{i=1}^{\mathsf{n}} (\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})\alpha_{\mathsf{y}_i} - \sum_{i=1}^{\mathsf{n}} (\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})\alpha_{\mathsf{x}_i} + \alpha_{\mathsf{z}} + 2\Gamma_{\mathsf{xy}} + \chi$ and sends $\mathsf{H}(\beta^\star_{\mathsf{z}})$ to P_1, P_2.
- P_j for $j \in \{1,2\}$ `abort` if $\mathsf{H}(\beta^\star_{\mathsf{z}}) \neq \mathsf{H}(\beta_{\mathsf{z}} - \sum_{i=1}^{\mathsf{n}} \beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i} + \psi)$.
- P_1 sends $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ and P_2 sends the $\mathsf{H}(\beta_{\mathsf{z}} + \gamma_{\mathsf{z}})$ to P_0. P_0 will `abort` if it receives inconsistent values.

Fig. 8: Dot Product Protocol

b) Truncation: A truncation protocol enables the servers to compute $[\![\mathsf{v}^d]\!]$ from $[\![\mathsf{v}]\!]$, where v^d denotes the truncated value of v (right-shifted value of v by d bit positions, where d is the number of bits allocated for the fractional part). SecureML [2] proposed an efficient truncation method for 2 parties where the parties locally truncate their shares after every multiplication. ABY3 [5] showed that this method fails when extended to 3-party, and proposed an alternative way using a shared truncated pair $(\mathsf{r}, \mathsf{r}^d)$, for a random r, to achieve truncation. Their method of truncating the shares of the product after evaluating a multiplication gate preserves the underlying truncated value with very high probability. We follow the technique of ABY3 and primarily differ in the way in which $(\mathsf{r}, \mathsf{r}^d)$ is generated. With the random truncation pair $(\mathsf{r}, \mathsf{r}^d)$ and a value v to be truncated, both available in $[\![\cdot]\!]$-shared form, the truncated v in $[\![\cdot]\!]$-shared format can be obtained by opening $(\mathsf{v} - \mathsf{r})$, truncating it and then adding it to $[\![\mathsf{r}^d]\!]$. Below we present a protocol that prepares the random truncation pair.

Protocol $\Pi_{\mathsf{trgen}}(\mathcal{P})$ (Fig. 9) generates a pair $([\mathsf{r}], [\![\mathsf{r}^d]\!])$ for a random r. Servers P_0, P_j for $j \in \{1,2\}$ sample random value $R_j \in \mathbb{Z}_{2^\ell}$ followed by P_0 locally truncating $\mathsf{r} = R_1 + R_2$ to obtain r^d. Note that $\mathsf{r} = 2^d\mathsf{r}^d + \mathsf{r}_d$ where r_d denotes the ring element that has last d bits of r in the last d positions and 0 elsewhere. P_0 then generates $[\![\mathsf{r}^d]\!]$ by executing the sharing protocol Π_{sh}. To verify the correctness of sharing performed by P_0, servers P_1, P_2 compute a $[\cdot]$-sharing of $\mathsf{a} = (\mathsf{r} - 2^d\mathsf{r}^d + \mathsf{r}_d)$, given $([\mathsf{r}], [\![\mathsf{r}^d]\!])$ and checks if $\mathsf{a} = 0$. To optimize communication, P_1 sends a hash of his share $\mathsf{H}([\mathsf{a}]_1)$

to P_2, who aborts if the received hash value mismatches with $\mathsf{H}(-[\mathsf{a}]_1)$.

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

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

Fig. 9: Generating Random Truncated Pair $(\mathsf{r}, \mathsf{r}^d)$

To see the correctness, it suffices to show that $\mathsf{u} = \mathsf{v}$ where $\mathsf{u} = [\mathsf{r}]_1 - 2^d [\mathsf{r}^d]_1 - [\mathsf{r}_d]_1$ and $\mathsf{v} = 2^d [\mathsf{r}^d]_2 + [\mathsf{r}_d]_2 - [\mathsf{r}]_2$. We start from the observation that $\mathsf{r} = 2^d \mathsf{r}^d + \mathsf{r}_d$.

$$\mathsf{r} = 2^d \mathsf{r}^d + \mathsf{r}_d$$
$$[\mathsf{r}]_1 + [\mathsf{r}]_2 = 2^d([\mathsf{r}^d]_{P_1} + [\mathsf{r}^d]_{P_2}) + ([\mathsf{r}_d]_{P_1} + [\mathsf{r}_d]_{P_2})$$
$$[\mathsf{r}]_1 - 2^d [\mathsf{r}^d]_1 - [\mathsf{r}_d]_1 = 2^d [\mathsf{r}^d]_2 + [\mathsf{r}_d]_2 - [\mathsf{r}]_2$$
$$\mathsf{u} = \mathsf{v}$$

$\Pi_{\mathsf{trgen}}(\mathcal{P})$ can entirely be run in the preprocessing phase. Our dot product with truncation, presented below, will invoke it in the preprocessing phase.

c) Dot Product with Truncation: Protocol $\Pi_{\mathsf{dotpt}}(\mathcal{P}, \{[\![\mathsf{x}_i]\!], [\![\mathsf{y}_i]\!]\}_{i\in[\mathsf{n}]})$ (Fig. 10) enables servers in $\mathcal{P}$ to generate $[\![\cdot]\!]$-sharing of truncated value of $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ denoted as z^d, given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$. To achieve the goal, we modify our dot product protocol Π_{dotp} in a way that does not inflate the online cost. This is unlike ABY3, which requires an additional reconstruction in the online phase.

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

Preprocessing:

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

Online:

- P_j for $j \in \{1,2\}$ computes $[(\mathsf{z}-\mathsf{r})]_j = [\mathsf{z}]_j - [\mathsf{r}]_j$ where $[\mathsf{z}]_j = [\beta_\mathsf{z}]_j - [\alpha_\mathsf{z}]_j = \sum_{i=1}^{\mathsf{n}}((j-1)\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i} - \beta_{\mathsf{x}_i}[\alpha_{\mathsf{y}_i}]_j - \beta_{\mathsf{y}_i}[\alpha_{\mathsf{x}_i}]_j) + [\Gamma_{\mathsf{xy}}]_j$.
- P_j for $j \in \{1,2\}$ mutually exchange $[(\mathsf{z}-\mathsf{r})]_j$ to reconstruct $(\mathsf{z}-\mathsf{r})$, followed by locally truncating it to obtain $(\mathsf{z}-\mathsf{r})^d$.
- P_1, P_2 execute $\Pi_{\mathsf{jsh}}(P_1, P_2, (\mathsf{z}-\mathsf{r})^d)$ to generate $[\![(\mathsf{z}-\mathsf{r})^d]\!]$.
- Servers in $\mathcal{P}$ locally compute $[\![\mathsf{z}]\!] = [\![(\mathsf{z}-\mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$
- P_0 computes $\Psi = -\sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})\alpha_{\mathsf{y}_i} - \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})\alpha_{\mathsf{x}_i} + \alpha_\mathsf{z} + 2\Gamma_{\mathsf{xy}} - \mathsf{r}$, sets $(\mathsf{z}-\mathsf{r})^\star = \Psi + \chi$ and sends $\mathsf{H}((\mathsf{z}-\mathsf{r})^\star)$ to both P_1 and P_2.
- P_j for $j \in \{1,2\}$ aborts if $\mathsf{H}((\mathsf{z}-\mathsf{r}) - \sum_{i=1}^{\mathsf{n}}\beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i} + \psi) \neq \mathsf{H}((\mathsf{z}-\mathsf{r})^\star)$.

Fig. 10: Dot Product Protocol with Truncation

In the preprocessing phase, along with the steps of Π_{dotp}, the servers execute Π_{trgen} to generate a truncation pair $(\mathsf{r}, \mathsf{r}^d)$. In the online phase, the servers P_1, P_2 locally compute $[\cdot]$-sharing of $(\mathsf{z}-\mathsf{r})$ (instead of $[\beta_\mathsf{z}]$) where $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$. This is followed by P_1, P_2 locally truncating $(\mathsf{z}-\mathsf{r})$ to obtain $(\mathsf{z}-\mathsf{r})^d$ and generating $[\![\cdot]\!]$-sharing of the same by executing Π_{jsh} protocol. Finally, the servers locally compute $[\![\cdot]\!]$-sharing of z by adding the shares of $(\mathsf{z}-\mathsf{r})^d$ and $[\![\mathsf{r}^d]\!]$. To ensure the correctness of the computation, the steps of P_0 are modified such that P_0 will be computing $(\mathsf{z}-\mathsf{r})^\star$ instead of $\beta_\mathsf{z}^\star$.

d) Secure Comparison: Given two values $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ in $[\![\cdot]\!]$-shared format, secure comparison allows parties to check whether $\mathsf{x} < \mathsf{y}$ or not. In fixed-point arithmetic representation, this can be accomplished by checking the sign of $\mathsf{v} = \mathsf{x} - \mathsf{y}$, which is stored in its msb position. Towards this, servers locally compute $[\![\mathsf{v}]\!] = [\![\mathsf{x}]\!] - [\![\mathsf{y}]\!]$ followed by extracting the msb using protocol Π_{bitext} on $[\![\mathsf{v}]\!]$. For the cases that demand the result in arithmetic sharing format, servers can apply the Bit2A protocol Π_{bit2A} on the outcome of Π_{bitext}.

e) Activation Functions: We consider two widely used activation functions– i) Rectified Linear Unit (ReLU) and ii) Sigmoid (Sig).

– *ReLU:* The ReLU function, defined as $\mathsf{relu}(\mathsf{v}) = \mathsf{max}(0, \mathsf{v})$ can be viewed as $\mathsf{relu}(\mathsf{v}) = \overline{\mathsf{b}} \cdot \mathsf{v}$ where the bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. Here $\overline{\mathsf{b}}$ denotes the complement of bit b. Protocol $\Pi_{\mathsf{relu}}(\mathcal{P}, [\![\mathsf{v}]\!])$ enables servers in $\mathcal{P}$ to compute $[\![\cdot]\!]$-sharing of $\mathsf{relu}(\mathsf{v})$ given the $[\![\cdot]\!]$-sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$.

For this, servers first execute the msb extraction protocol Π_{bitext} on v to obtain $[\![\mathsf{b}]\!]^{\mathbf{B}}$. Given $[\![\mathsf{b}]\!]^{\mathbf{B}}$, servers locally compute $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}}$ by setting $\beta_{\overline{\mathsf{b}}} = 1 \oplus \beta_\mathsf{b}$. Servers then execute Bit2A protocol Π_{bit2A} on $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}}$ to generate $[\![\overline{\mathsf{b}}]\!]$. Lastly, servers execute multiplication protocol Π_{mult} on $\overline{\mathsf{b}}$ and v to generate $[\![\cdot]\!]$-sharing of the result.

– *Sig:* We use the MPC-friendly version of the Sigmoid function [2], [5], [48], which is defined as:

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

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 $\mathsf{v} - 1/2 < 0$. Protocol $\Pi_{\mathsf{sig}}(\mathcal{P}, [\![\mathsf{v}]\!])$ is similar to that of Π_{relu} and therefore we omit the details.

V. Building PPML and Benchmarking

We consider three widely used ML algorithms for our benchmarking and compare with their closest competitors– i) Linear Regression (training and inference), ii) Logistic Regression (training and inference) and iii) Neural Networks (inference). Training for NN requires conversions to and from Garbled Circuits (for tackling some functions) which are not considered in this work. To obtain fairness in our protocols, the final outcome is reconstructed via fair reconstruction protocol $\Pi_{\mathsf{frec}}(\mathcal{P}, [\![\mathsf{v}]\!])$ (Fig. 20). In addition to the above, we also benchmark the dot product protocol separately as it is a major building block for PPML. We start with the experimental setup.

a) Benchmarking Environment: We use a 64-bit ring ($\mathbb{Z}_{2^{64}}$). The benchmarking is performed over a WAN of 75Mbps bandwidth. The WAN was instantiated using n1-standard-8 instances of Google Cloud[2] with machines located in East Australia (P_0), South Asia (P_1) and South East Asia (P_2). 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. The average round-trip time (rtt) was taken as the time for communicating 1 KB of data between a pair of parties. The rtt values for the pairs P_0-P_1, P_0-P_2 and P_1-P_2 are $152.3ms$, $60.19ms$ and $92.63ms$ respectively.

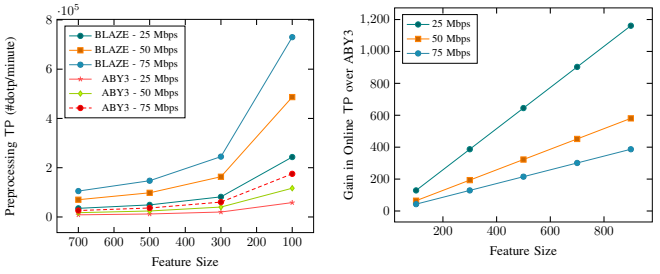
b) Software Details: We implement our protocols using the publicly available ENCRYPTO library [50] in C++17. We implemented the code of ABY3 [5] and ASTRA [48] in our environment since they were not publicly available. The collision-resistant hash function was instantiated using SHA-256. We have used 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.

c) Benchmarking Parameter: We use *throughput* (TP) as the benchmarking parameter following ABY3 [5] and ASTRA [48] 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. To analyse the performance of our protocols under various bandwidth settings, we report the performance under the following bandwidths: 25 Mbps, 50Mbps and, 75Mbps.

We defer the comparison with ASTRA to §D-B1. The benchmarking for the LAN setting is provided in the full version of the paper [49].

A. Dot Product

Here the throughput is computed as the number of dot products performed per minute (#dotp/min) and the same is computed for both preprocessing and online phases separately.



(a) Preprocessing TP (b) Online TP

Fig. 11: Throughput (TP) Comparison of ABY3 and BLAZE over varying Bandwidths

For the preprocessing phase, we plot the throughput of the dot product protocol of BLAZE (Fig. 11a) and ABY3 over vectors of length ranging from 100 to 1000. We note at least a gain of $4\times$, which is a consequence of $4\times$ improvement in communication, over ABY3. An interesting observation to be made here is that our protocol, over the bandwidth of 25Mbps, gives better throughput when compared to ABY3 even over a higher bandwidth of 75Mbps.

For the online phase (Fig. 11b), we plot the gain over ABY3 in terms of throughput. We observe an appreciable gain in throughput which is a direct corollary of the communication cost our protocol being independent of the vector size. Concretely, for a bandwidth of 50 Mbps, our gain ranges from $64\times$ to $580\times$. Note that, with an increase in bandwidth there is a drop in the gain. This is because even at a bandwidth of 25 Mbps the maximum attainable throughput cannot be handled by our processors. For a bandwidth of 8 Mbps, the maximum attainable throughput is within our processing capacity, where we observe throughput gain ranging from $400\times$ to $3600\times$. This showcases the practicality of our constructions over low-end networks.

In the preprocessing phase, over all the bandwidths under consideration, the maximum attainable throughput lies well within the processing capacity of our machines. Consequentially, we do not observe a drop in the throughput gain with increasing bandwidth, as is seen in the online phase. This is the reason why we choose to plot the actual throughput values instead of the gain in the case of the preprocessing phase. On increasing the processing capacity we expect a consistent gain in online throughput with increasing bandwidth.

B. ML Training

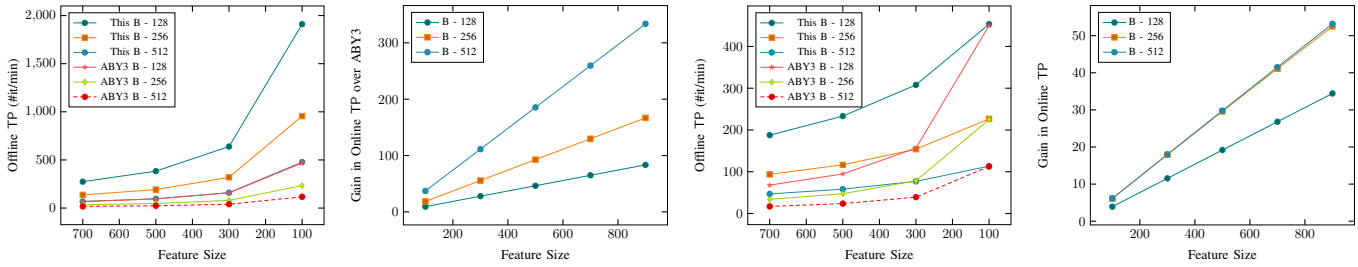
In this section, we explore the training phase of Linear Regression and Logistic Regression algorithms. The training phase can be divided into two stages– (i) a *forward propagation* phase, where the model computes the output given the input; (ii) a *backward propagation* phase, where the model parameters are adjusted according to the difference in the computed output and the actual output. For our benchmarking, we define one *iteration* in the training phase as one forward propagation followed by a backward propagation. Our performance improvement over ABY3 is reported in terms of the number of iterations over feature size varying from 100 to 1000, and a batch size of $B \in \{128, 256, 512\}$. Batching [2], [5] is a common optimization where n samples are divided into batches of size B and a combined update function is applied to the weight vectors. In order to analyse the performance over a wide range of features and batch sizes, we choose to benchmark over synthetic datasets following ABY3 [5].

a) **Linear Regression:** In Linear Regression, one iteration comprises of updating the weight vector $\vec{\mathbf{w}}$ using the gradient descent algorithm (GD). It is updated according to the following function:

$$\vec{\mathbf{w}} = \vec{\mathbf{w}} - \frac{\alpha}{B}\mathbf{X}_i^T \circ ((\mathbf{X}_i \circ \vec{\mathbf{w}}) - \mathbf{Y}_i)$$

where α denotes the learning rate and $\mathbf{X}_i$ denotes a subset of batch size B, randomly selected from the entire dataset in the ith iteration. The forward propagation involves computing $\mathbf{X}_i \circ \vec{\mathbf{w}}$, while the backward propagation consists of updating the weight vector. The update function requires computation of a series of matrix multiplications, which can be achieved

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



(a) Linear Reg. Preprocessing (b) Linear Reg. Online (c) Logistic Reg. Preprocessing (d) Logistic Reg. Online

Fig. 12: Throughput (TP) Comparison of ABY3 and BLAZE for ML Training

using dot product protocols. The update function, as mentioned earlier, can be computed entirely using $[\![\cdot]\!]$ shares as:

$$[\![\vec{\mathbf{w}}]\!] = [\![\vec{\mathbf{w}}]\!] - \frac{\alpha}{B}[\![\mathbf{X}_j^T]\!] \circ (([\![\mathbf{X}_j]\!] \circ [\![\vec{\mathbf{w}}]\!]) - [\![\mathbf{Y}_j]\!])$$

The operations of subtraction as well as multiplication by a public constant can be performed locally.

We compare the throughput for Linear Regression in Fig. 12. Fig. 12a depicts throughput in the preprocessing phase, and Fig. 12b illustrates the online throughput gain over ABY3. Since Linear Regression primarily involves computing multiple dot products, the underlying efficient dot product protocol improves the performance drastically. As a result, in the preprocessing phase, we observe a gain of $4\times$ and, in the online phase, performance gain for a batch size of 128 ranges from $9.2\times$ to $83.4\times$. The performance gain in the online phase increases significantly for larger batch sizes and goes all the way up to $333\times$ for a batch size of 512.

Algorithm	Ref.	Preprocessing		Online	
		TP	Gain	TP	Gain
Linear Regression	ABY3	61.02	**4.01×**	30.61	**145.35×**
	BLAZE	**244.74**		**4449.55**	
Logistic Regression	ABY3	60.71	**2.86×**	60.99	**30.01×**
	BLAZE	**173.36**		**1830.26**	

TABLE VI: Throughput (TP) for ML Training for a batch size B-128 and feature size n-784

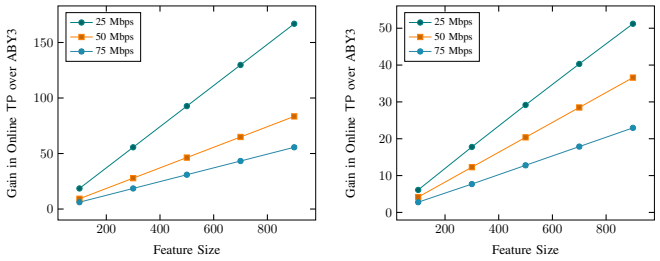
b) **Logistic Regression:** The training in Logistic Regression, is similar to the case of Linear Regression, with an additional application of sigmoid activation function over $\mathbf{X}_i \circ \vec{\mathbf{w}}$ in the forward propagation. Precisely, the update function for $\vec{\mathbf{w}}$ is as follows:

$$\vec{\mathbf{w}} = \vec{\mathbf{w}} - \frac{\alpha}{B}\mathbf{X}_i^T \circ (\mathsf{sig}(\mathbf{X}_i \circ \vec{\mathbf{w}}) - \mathbf{Y}_i)$$

The performance of the training phase in Logistic Regression is analysed in Fig. 12. The throughput in the preprocessing phase is depicted in Fig. 12c, while Fig. 12d showcases the online throughput gains. The improvements seen in Linear Regression are carried over to this case as well. An overall drop in the throughput is observed both in the preprocessing as well as in the online phase because of the overhead caused by the sigmoid activation function. In the preprocessing phase, our protocol for the largest batch size under consideration outperforms that of ABY3 over the smallest batch size. In the online phase, improvements range from $3.93\times$ to $34.43\times$ for the batch size of 128, compared to ABY3. The primary reason for this gain is our efficient method for msb extraction, which requires **2** rounds in the online phase, as opposed to $1+\log \ell$ rounds of communication in ABY3. Similar to Linear Regression, an increase in the online throughput gain can be observed for larger batch sizes.

Table VI provides concrete details for the training phase of Linear Regression and Logistic Regression algorithms over a batch size of 128 and a feature size of 784. More details are provided in §D.

c) Comparison over varying Bandwidths: Here, we analyse the performance of the training algorithms in the online phase over varying bandwidths. In Fig. 13a, we plot the gain in online throughput of Our protocol over ABY3 for the Linear Regression algorithm for the bandwidths– 25Mbps, 50Mbps, and 75Mbps. We observe that the improvement in communication cost is even more conspicuous for lower bandwidth. The gain over the bandwidth of 75Mbps ranges from $6.18\times$ to $55.62\times$ over various feature sizes, while over a bandwidth of 25Mbps it ranges from $18.54\times$ to $166.86\times$. This shows the practicality of our protocol over low bandwidths.



(a) Linear Regression (b) Logistic Regression

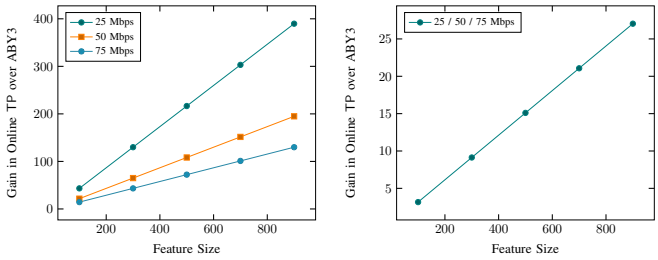
Fig. 13: Online Throughput (TP) Comparison of ABY3 and BLAZE over varying Bandwidths

A similar trend is observed for the case of Logistic Regression and the plot is presented in Fig. 13b. For a bandwidth of 75Mbps, the gain in online throughput ranges from $2.78\times$ to $22.95\times$, while for 25Mbps, the range is from 6.11 to 51.21.

C. ML Inference

In this section, we benchmark the inference phase of Linear Regression, Logistic Regression, and NN. For inference, the benchmarking parameter is the number of queries processed per minute (#queries/min). While the details for the online phase are presented here, we defer the details for the preprocessing phase to §D-B. Like ABY3 and SecureML [2], our method for truncation introduces a bit-error at the least significant bit position. The accuracy of the prediction itself ranges from 93.2% for linear regression to 97.8% for NN.

a) Linear Regression and Logistic Regression: Inference in the case of Linear Regression and Logistic Regression can be viewed as a single pass of the forward propagation phase. Below we provide the benchmarking for the same over the protocols of ABY3.



(a) Linear Regression (b) Logistic Regression

Fig. 14: Online Throughput (TP) Comparison of ABY3 and BLAZE for Linear Regression and Logistic Regression Inference

For Linear Regression, we observe that the gain in online throughput over ABY3 ranging from $14\times$ to $216\times$ across different bandwidths. The respective gain for Logistic Regression ranges from $3\times$ to $27\times$.

In ASTRA, the inference phase of Linear and Logistic Regression are optimized further. For instance, the output of Logistic Regression is the boolean sharing of a single bit. Hence, we benchmarked our protocol with the optimizations of ASTRA and the benchmarking appears in §D-B1.

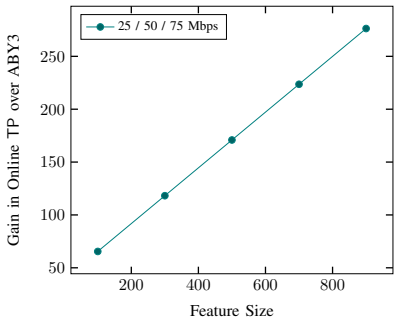


Fig. 15: Comparison of Online Throughput (TP) of BLAZE and ABY3 for Neural Network Inference

b) Neural Networks: Here we consider a NN with two hidden layers, each having 128 nodes followed by an output layer of 10 nodes. The activation function ReLU (relu) is applied after the evaluation of each layer. For a bandwidth of 25Mbps, our protocol could process the online phase of $16,866$ queries in a minute and the throughput goes all the way up to $50,602$ queries/min for a bandwidth of 75Mbps. ABY3, on the other hand, can process 70 and 210 queries/min for 25Mbps and 75Mbps, respectively.

Fig. 15 plots the gain in online throughput of BLAZE over ABY3 for varying feature sizes. Unlike Linear Regression and Logistic Regression, the gain is not dropped with the increase in bandwidth. This is because of the huge communication incurred for NN which makes the maximum attainable throughput within our processing capacity.

Table VII provides concrete details for the inference phase of the aforementioned algorithms for feature size of 784.

Algorithm	Ref.	Preprocessing		Online	
		TP ($\times 10^3$)	Gain	TP ($\times 10^3$)	Gain
Linear Regression	ABY3	15.57	**4.02**×	15.67	**169.75**×
	BLAZE	**62.61**		**2660.53**	
Logistic Regression	ABY3	15.41	**4.03**×	15.55	**23.57**×
	BLAZE	**62.13**		**366.68**	
Neural Networks	ABY3	0.10	**4.01**×	0.14	**245.74**×
	BLAZE	**0.41**		**33.74**	

TABLE VII: Throughput (TP) for ML Inference for a feature size of n-784

1) ML Inference on Real World Datasets: Here we benchmark the online phase of ML inference of all the three algorithms over real-world datasets (Table IX). The datasets are obtained from UCI Machine Learning Repository [51] and the details are provided in Table VIII.

Algorithm	Dataset	#features	#samples
Linear Regression	**Superconductivity** Critical Temperature Data Set [52]	81	21263
Logistic Regression	**FMA** Music Analysis Dataset [53]	518	106574
Neural Networks	**Parkinson** Disease Classification Dataset [54]	754	754

TABLE VIII: Real World Datasets used for ML Inference

In Table IX, we observe that the online throughput of our protocols for the case of Linear Regression and Logistic Regression is not increasing with the increase in bandwidth. This can be justified as the processing capacity becomes the bottleneck and prevents our protocols from reaching the maximum attainable throughput even for a bandwidth of 25Mbps. This can be prevented by introducing more computing power to the environment.

Bandwidth	Linear Regression **(Superconductivity)**		Logistic Regression **(FMA)**		Neural Networks **(Parkinson)**	
	ABY3	BLAZE	ABY3	BLAZE	ABY3	BLAZE
25 Mbps	75852	**2660532**	11725	**183339**	70	**16867**
50 Mbps	151704	**2660532**	23450	**366678**	140	**33735**
75 Mbps	227556	**2660532**	35175	**550017**	210	**50603**

TABLE IX: Comparison of Online TP of ABY3 and BLAZE for Inference over Real World Datasets (Datasets are given in Brackets). Values are given in #queries/min.

VI. Conclusion

In this work, we presented a blazing fast framework, BLAZE, for PPML. Our framework, designed for three servers tolerating at most one malicious corruption, works seamlessly over a ring $\mathbb{Z}_{2^\ell}$. Cast in the preprocessing model, our constructs outperform the state-of-the-art solutions by several orders of magnitude both in the round and communication complexity. We showcased the application of our framework in Linear Regression, Logistic Regression, and Neural Networks. We leave open the problem of extending our framework to support training of Neural Networks. Another interesting line of work is to explore the potential of Trusted Execution Environments (TEE) [4] in improving the overall efficiency of the framework.

REFERENCES

[1] A. Madani, R. Arnaout, M. R. K. Mofrad, and R. Arnaout, "Fast and accurate classification of echocardiograms using deep learning," *npj Digital Medicine*, 2018.

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

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

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

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

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

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

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

[9] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier," in *IEEE S&P*, 2017, pp. 843–862.

[10] Y. Lindell and A. Nof, "A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority," in *ACM CCS*, 2017, pp. 259–276.

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

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

[13] Y. Ishai, R. Kumaresan, E. Kushilevitz, and A. Paskin-Cherniavsky, "Secure computation with minimal interaction, revisited," in *CRYPTO*, 2015, pp. 359–378.

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

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

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

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

[18] D. Bogdanov, R. Talviste, and J. Willemson, "Deploying secure multi-party computation for financial data analysis - (short paper)," in *FC*, 2012, pp. 57–64.

[19] J. Launchbury, D. Archer, T. DuBuisson, and E. Mertens, "Application-scale secure multiparty computation," in *ESOP*, 2014, pp. 8–26.

[20] D. Bogdanov, L. Kamm, B. Kubo, R. Rebane, V. Sokk, and R. Talviste, "Students and taxes: a privacy-preserving social study using secure computation," *PoPETs*, pp. 117–135, 2016.

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

[22] M. Geisler, "Viff: Virtual ideal functionality framework," 2007.

[23] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *CRYPTO*, 1991, pp. 420–432.

[24] ——, "Precomputing oblivious transfer," in *CRYPTO*, 1995, pp. 97–109.

[25] Z. Beerliová-Trubíniová and M. Hirt, "Efficient multi-party computation with dispute control," in *TCC*, 2006, pp. 305–328.

[26] ——, "Perfectly-secure MPC with linear communication complexity," in *TCC*, 2008, pp. 213–230.

[27] E. Ben-Sasson, S. Fehr, and R. Ostrovsky, "Near-linear unconditionally-secure multiparty computation with a dishonest minority," in *CRYPTO*, 2012, pp. 663–680.

[28] A. Choudhury and A. Patra, "An efficient framework for unconditionally secure multiparty computation," *IEEE Trans. Information Theory*, vol. 63, no. 1, pp. 428–468, 2017.

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

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

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

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

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

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

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

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

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

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

[39] 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*, pp. 1102–1120, 2019.

[40] Y. Lindell and B. Pinkas, "Privacy preserving data mining," in *CRYPTO*, 2000, pp. 36–54.

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

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

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

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

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

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

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

[48] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High-throughput 3PC over Rings with Application to Secure Prediction," in *ACM CCSW*, 2019, pp. 81–92.

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

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

[51] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[52] K. Hamidieh, "A data-driven statistical model for predicting the critical temperature of a superconductor," *Computational Materials Science*, pp. 346 – 354, 2018.

[53] K. Benzi, M. Defferrard, P. Vandergheynst, and X. Bresson, "FMA: A dataset for music analysis," *CoRR*, 2016.

[54] C. O. Sakar, G. Serbes, A. Gunduz, H. C. Tunc, H. Nizam, B. E. Sakar, M. Tutuncu, T. Aydin, M. E. Isenkul, and H. Apaydin, "A comparative analysis of speech signal processing algorithms for parkinson's disease classification and the use of the tunable q-factor wavelet transform," *Appl. Soft Comput.*, pp. 255–263, 2019.

APPENDIX A
PRELIMINARIES

a) Shared Key Setup: Let $F : 0,1^\kappa \times 0,1^\kappa \to X$ be a secure pseudo-random function PRF, with co-domain X being $\mathbb{Z}_{2^\ell}$. The set of keys established among the servers are:

– One key shared between every pair– k_{01}, k_{02}, k_{12} for the parties $(P_0, P_1), (P_0, P_2), (P_1, P_2)$ respectively.

– One shared key amongst all– $k_{\mathcal{P}}$.

If the servers P_0, P_1 wish to sample a random value $r \in \mathbb{Z}_{2^\ell}$ non-interactively, they invoke $F_{k_{01}}(id_{01})$ to obtain r, where id_{01} is a counter which the servers update locally after every PRF invocation. The key used to sample a value will be clear from the context (from the identities of the pair that samples or from the fact that it is sampled by all) and will be omitted. We model the key setup via a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 16) that can be realised using any secure MPC protocol.

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_i denote the keys corresponding to server P_i. Then

– $\mathsf{y}_i = (k_{01}, k_{02}$ and $k_{\mathcal{P}})$ when $P_i = P_0$.

– $\mathsf{y}_i = (k_{01}, k_{12}$ and $k_{\mathcal{P}})$ when $P_i = P_1$.

– $\mathsf{y}_i = (k_{02}, k_{12}$ and $k_{\mathcal{P}})$ when $P_i = P_2$.

Output to adversary: If $\mathcal{S}$ sends $\mathtt{abort}$, then send $(\mathsf{Output}, \perp)$ to all the servers. Otherwise, send $(\mathsf{Output}, \mathsf{y}_i)$ to the adversary $\mathcal{S}$, where y_i denotes the keys corresponding to the corrupt server.

Output to selected honest servers: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest servers. If an honest server P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, \mathsf{y}_i)$.

Fig. 16: Functionality for Shared Key Setup

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

c) Commitment Scheme: Let $\mathsf{Com}(x)$ denote the commitment of a value x. The commitment scheme $\mathsf{Com}(x)$ possesses two properties; *hiding* and *binding*. The former ensures that given just the commitment, privacy of value x is guaranteed. The latter prevents a corrupt party from opening the commitment to a different value $x' \neq x$. The commitment scheme can be instantiated using a hash function $\mathcal{H}()$, whose security can be proved in the random-oracle model (ROM). For instance, $(c, o) = (\mathcal{H}(x||r), x||r) = \mathsf{Com}(x; r)$.

APPENDIX B
MULTIPLICATION PROTOCOL OF [17]

In this section, we provide details of the multiplication protocol proposed by [17] on $\langle\cdot\rangle$-shared values. For a value v, the $\langle\cdot\rangle$-sharing is defined as:

$$\langle \mathsf{v} \rangle_0 = ([\lambda_\mathsf{v}]_1, [\lambda_\mathsf{v}]_2), \langle \mathsf{v} \rangle_1 = ([\lambda_\mathsf{v}]_1, \mathsf{v} + \lambda_\mathsf{v}), \langle \mathsf{v} \rangle_2 = ([\lambda_\mathsf{v}]_2, \mathsf{v} + \lambda_\mathsf{v})$$

Given the $\langle\cdot\rangle$-shares of d and e, $\Pi_{\mathsf{mulZK}}(\mathcal{P}, \langle \mathsf{d} \rangle, \langle \mathsf{e} \rangle)$ (Fig. 17) computes $\langle\cdot\rangle$-share of $\mathsf{f} = \mathsf{de}$.

Protocol $\Pi_{\mathsf{mulZK}}(\mathcal{P}, \langle \mathsf{d} \rangle, \langle \mathsf{e} \rangle)$

Computation:

– Servers P_0, P_j for $j \in \{1,2\}$ locally sample a random $[\lambda_\mathsf{f}]_j \in \mathbb{Z}_{2^\ell}$. Also, P_0, P_1 samples a random $[\lambda_{\mathsf{d},\mathsf{e}}]_1 \in \mathbb{Z}_{2^\ell}$

– P_0 computes $\lambda_{\mathsf{d},\mathsf{e}} = \lambda_\mathsf{d} \cdot \lambda_\mathsf{e}$ and sets $[\lambda_{\mathsf{d},\mathsf{e}}]_2 = \lambda_{\mathsf{d},\mathsf{e}} - [\lambda_{\mathsf{d},\mathsf{e}}]_1$. P_0 sends $[\lambda_{\mathsf{d},\mathsf{e}}]_2$ to P_2.

– Server P_j for $j \in \{1,2\}$ computes and mutually exchanges $[\mathsf{f} + \lambda_\mathsf{f}]_j = (j-1)(\mathsf{d}+\lambda_\mathsf{d})(\mathsf{e}+\lambda_\mathsf{e}) - [\lambda_\mathsf{d}]_j (\mathsf{e}+\lambda_\mathsf{e}) - [\lambda_\mathsf{e}]_j (\mathsf{d}+\lambda_\mathsf{d}) + [\lambda_{\mathsf{d},\mathsf{e}}]_j + [\lambda_\mathsf{f}]_j$ to reconstruct $(\mathsf{f} + \lambda_\mathsf{f})$.

Verification: Using distributed zero-knowledge, each server proves the correctness of the following statement to the other two:

– Server P_0: $\lambda_{\mathsf{d},\mathsf{e}} = \lambda_\mathsf{d} \cdot \lambda_\mathsf{e}$.

– Server P_j: $[\mathsf{f} + \lambda_\mathsf{f}]_j = (j-1)(\mathsf{d}+\lambda_\mathsf{d})(\mathsf{e}+\lambda_\mathsf{e}) - [\lambda_\mathsf{d}]_j (\mathsf{e}+\lambda_\mathsf{e}) - [\lambda_\mathsf{e}]_j (\mathsf{d}+\lambda_\mathsf{d}) + [\lambda_{\mathsf{d},\mathsf{e}}]_j + [\lambda_\mathsf{f}]_j$. Here $j \in \{1,2\}$.

Fig. 17: $\langle\cdot\rangle$-shared Multiplication Protocol of [17]

Now we explain the verification method of [17] in detail. The technique enables prover P to prove to the verifiers $\mathsf{V}_1, \mathsf{V}_2$ in zero knowledge that it knows w such that $(x, w) \in \mathcal{R}$. Let ckt denotes the circuit corresponding to the statement being verified such that $\mathsf{ckt}(x, w) = 0$ iff $(x, w) \in \mathcal{R}$. The statement x is shared among the verifiers; x_1 with V_1 and x_2 with V_2 such that $x = x_1||x_2$, where $||$ denotes concatenation, $|x_1| = n_1$, $|x_2| = n_2$ and $n = n_1 + n_2$. Let M be the number of multiplication gates in ckt.

Without loss of generality and for easy explanation, we consider the circuit given in Figure 18 and the prover and verifiers being P_0 and (P_1, P_2) respectively.

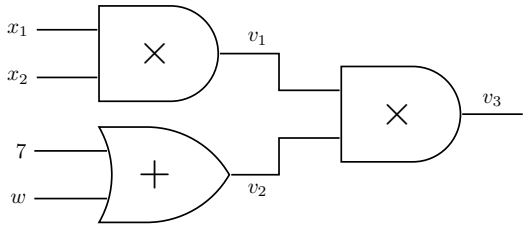


Fig. 18: Example circuit

The prover P_0 first constructs three polynomials $f()$, $g()$ and $h()$ on the values on left, right and output wires, respectively, of the multiplication gates. The constant terms of $f()$ and $g()$ are set as random ring elements z_1 and z_2

respectively, while the constant term for $h()$ is set as $z_1 z_2$. More precisely,

$$\begin{aligned} f(0) &= z_1, & g(0) &= z_2, & h(0) &= z_1 z_2 \\ f(1) &= x_2, & g(1) &= x_1, & h(1) &= x_1 x_2 \\ f(2) &= v_2 = 7 + w, & g(2) &= v_1 = x_1 x_2, & h(2) &= v_1 v_2 \end{aligned}$$

With the above values set, P_0 interpolates the polynomials $f()$, $g()$ and $h()$. While $f(), g()$ are polynomials of degree at most M, $h()$ is a polynomial of degree at most $d = 2M$. The proof π is then defined as $(w, z_1, z_2, c_h) \in \mathbb{Z}_{2^\ell}^{\sigma}$ where c_h denotes the coefficients of $h()$. The size of proof is denoted by $\sigma = s + d + 3$ where s is the size of the witness. P_0 then provides additive shares of π, denoted by π_i, to the verifiers; π_i to P_i for $i \in \{1, 2\}$. Note that if P_0 is honest then $\forall r \in \mathbb{Z}_{2^\ell}$, $h(r) = f(r)g(r)$ and $h(M) = 0$.

Verifiers P_1, P_2 together sample a random value $r \in \mathbb{Z}_{2^\ell} \setminus \{z_1, z_2\}$ and generate corresponding query vectors q_f, q_g and $q_h \in \mathbb{Z}_{2^\ell}^{n+\sigma}$. Each verifier P_i for $i \in \{1, 2\}$ then construct three query vectors from q_f, q_g and q_h. More precisely, corresponding to polynomial $f()$, verifier P_i constructs vector $Q_f^i \in \mathbb{F}^{n_i+\sigma}$ from q_f such that the first n_i positions are reserved for entries corresponding to x_i followed by q_f. P_i for $i \in \{1, 2\}$ then locally computes the dot product ($\odot$) of the vectors $(x_i || \pi_i)$ and Q_f^i as $f_i(r) = (x_i || \pi_i) \odot Q_f^i$ and sends it to verifier P_1. P_1 after receiving the shares of $f_i(r)$ computes the value $f(r) = f_1(r) + f_2(r)$. This comes from the fact that each query vector q defines a linear combination of the input x and proof π. Hence, the verifiers can form additive shares of the answers to the queries, which is the polynomials evaluated at r, using their parts of input x_i and additive share π_i. This comes from the fact that each query vector q defines a linear combination of the input x and proof π. Hence, the verifiers can form additive shares of the answers to the queries, which is the polynomials evaluated at r, using their parts of input x_i and additive share π_i. Similar steps are done for polynomials $g()$ and $h()$ which enables P_1 to obtain $g(r)$ and $h(r)$. P_1 `aborts` if $h(r) \neq f(r)g(r)$. A cheating prover P_0 will pass this check with probability at most $\frac{2M-1}{2^\ell - 2}$, which for a large enough ℓ is negligible.

Now for the second check, ie. $h(M) = 0$, verifiers generate query vector q in a similar fashion. More precisely, P_i for $i \in \{1, 2\}$ forms Q^i, computes $h_i(M)$ and sends his share of $h(M)$ to P_1. Verifier P_1 `abort` if $h(M) \neq h_1(M) + h_2(M) = 0$.

[17] propose two variants of the above technique. The first variant gives a 2 round fully linear interactive oracle proofs with query complexity $\mathrm{O}(\sqrt{n})$, where n is the size of the input. The second variant gives $\mathrm{O}(\log(M))$ rounds fully linear interactive oracle proofs with query complexity of $\mathrm{O}(\log(M))$, where M is the number of multiplication gates in ckt. We use the former result in our work. We refer the readers to [17] for a more detailed description of the verification and its optimizations.

Lemma B.1 (Communication)**.** *Protocol* Π_{mulZK} *requires* 4 *rounds and an amortized communication of* 3ℓ *bits.*

The ideal-world functionality realising Π_{mulZK} protocol is presented in Fig. 19.

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

$\mathcal{F}_{\mathsf{mulZK}}$ interacts with the servers in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{mulZK}}$ receives the $\langle\cdot\rangle$-shares of values d and e from the servers where,

$$\begin{aligned} \langle \mathsf{d} \rangle_0 &= ([\lambda_\mathsf{d}]_1, [\lambda_\mathsf{d}]_2), & \langle \mathsf{d} \rangle_1 &= ([\lambda_\mathsf{d}]_1, \mathsf{d} + \lambda_\mathsf{d}), & \langle \mathsf{d} \rangle_2 &= ([\lambda_\mathsf{d}]_2, \mathsf{d} + \lambda_\mathsf{d}) \\ \langle \mathsf{e} \rangle_0 &= ([\lambda_\mathsf{e}]_1, [\lambda_\mathsf{e}]_2), & \langle \mathsf{e} \rangle_1 &= ([\lambda_\mathsf{e}]_1, \mathsf{e} + \lambda_\mathsf{e}), & \langle \mathsf{e} \rangle_2 &= ([\lambda_\mathsf{e}]_2, \mathsf{e} + \lambda_\mathsf{e}) \end{aligned}$$

If the functionality receives $\perp$ from $\mathcal{S}$, then send $\perp$ to every server, else do the following:

Computation of output: Compute $\mathsf{d} = (\mathsf{d} + \lambda_\mathsf{d}) - [\lambda_\mathsf{d}]_1 - [\lambda_\mathsf{d}]_2$ and $\mathsf{e} = (\mathsf{e} + \lambda_\mathsf{e}) - [\lambda_\mathsf{e}]_1 - [\lambda_\mathsf{e}]_2$ followed by computing $\mathsf{f} = \mathsf{de}$. Randomly select $[\lambda_\mathsf{f}]_1, [\lambda_\mathsf{f}]_2$ from $\mathbb{Z}_{2^\ell}$ and set $\lambda_\mathsf{f} = [\lambda_\mathsf{f}]_1 + [\lambda_\mathsf{f}]_2$. The output shares are set as

$$\langle \mathsf{f} \rangle_0 = ([\lambda_\mathsf{f}]_1, [\lambda_\mathsf{f}]_2), \quad \langle \mathsf{f} \rangle_1 = ([\lambda_\mathsf{f}]_1, \mathsf{f} + \lambda_\mathsf{f}), \quad \langle \mathsf{f} \rangle_2 = ([\lambda_\mathsf{f}]_2, \mathsf{f} + \lambda_\mathsf{f})$$

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the servers. Otherwise, send $(\mathsf{Output}, \langle \mathsf{f} \rangle_\mathcal{S})$ to the adversary $\mathcal{S}$, where $\langle \mathsf{f} \rangle_\mathcal{S}$ denotes the share of f corresponding to the corrupt server.

Output to selected honest servers: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest servers. If an honest server P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, \langle \mathsf{f} \rangle_i)$, where $\langle \mathsf{f} \rangle_i$ denotes the share of f corresponding to the honest server P_i.

Fig. 19: Functionality for Π_{mulZK}

APPENDIX C
FAIR RECONSTRUCTION PROTOCOL

Protocol $\Pi_{\mathsf{frec}}(\mathcal{P}, [\![\mathsf{v}]\!])$ (Fig. 20) ensures fair reconstruction of the secret v for servers in $\mathcal{P}$. This implies that the honest servers are guaranteed to obtain the secret v whenever the corrupt server obtains the same. The techniques for fair reconstruction introduced in ASTRA to achieve fairness are adapted for our sharing scheme.

The protocol proceeds as follows: In order to fairly reconstruct v, servers together commit to their common shares. Concretely, in the preprocessing phase, the servers P_0, P_1 commit $[\alpha_\mathsf{v}]_1$ to P_2 and P_0, P_2 commit $[\alpha_\mathsf{v}]_2$ to P_1. In the online phase, the servers P_1, P_2 commit β_v to P_0. The recipient in each case can abort if the received commitments do not match. In the case of no abort, P_0 signals P_1 and P_2 to start opening the commitments which provides each server with the missing share so that they can reconstruct v. It is fair because at least one honest party would have provided the missing share that would allow reconstruction. Lastly, if the protocol aborts before, then none receive the output. Note that a corrupt P_0 can send distinct signals to P_1 and P_2 (abort to one and continue to the other), breaching unanimity. To resolve this without relying on a broadcast channel, P_0, P_1 together commit a value r_1 to P_2 and P_0, P_2 together commit a common value r_2 to P_1 in the preprocessing phase. In the online phase, if P_0 aborts, it sends opening of r_2 to P_1 and r_1 to P_2, along with the abort signal. Now a server, say P_1 on receiving the abort can convince P_2 that it has indeed received abort from P_0, using r_2 as the *proof of origin* for the abort message. This is because P_1 can secure r_2 only via P_0. A single pair of (r_1, r_2) can be used as a proof of origin for multiple instances of reconstruction running in parallel.

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

Preprocessing:

– Servers P_0, P_j for $j \in \{1,2\}$ locally sample a random $r_j \in \mathbb{Z}_{2^\ell}$, prepare commitments of $[\alpha_\mathsf{v}]_j$ and r_j. P_0, P_j then send $(\mathsf{Com}([\alpha_\mathsf{v}]_j)\ ,\ \mathsf{Com}(r_j))$ to P_{2-j}.

– P_j for $j \in \{1,2\}$ `abort` if the received commitments mismatch.

Online:

– P_1, P_2 compute a commitment of β_v and send it to P_0.

– If the commitments do not match, P_0 sends $(\mathtt{abort}, o_j)$ to P_{2-j} for $j \in \{1,2\}$ and aborts, where o_j denotes opening information for the commitment of r_j. Else P_0 sends `continue` to both P_1 and P_2.

– P_1, P_2 exchange the messages received from P_0.

– P_1 aborts if he receives either (i) $(\mathtt{abort}, o_2)$ from P_0 and o_2 opens the commitment of r_2 or (ii) $(\mathtt{abort}, o_1)$ from P_2 and o_1 is the correct opening information of r_1. The case for P_2 is similar to that of P_1

– If no abort happens, servers obtain their missing share of v as follows:

– P_0, P_1 open $[\alpha_\mathsf{v}]_1$ towards P_2.

– P_0, P_2 open $[\alpha_\mathsf{v}]_2$ towards P_1.

– P_1, P_2 open β_v towards P_0.

– Servers reconstruct the value v using missing share that matches with the agreed upon commitment.

Fig. 20: Fair Reconstruction of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ among $\mathcal{P}$

In the outsourced setting, the fair reconstruction of a value v proceeds as follows: Servers execute all the steps of fair reconstruction protocol (Fig. 20) except the opening of the commitments in the online phase. If no `abort` happens, then each of the three servers sends the commitment of $[\![\alpha_\mathsf{v}]\!]^{\mathbf{A}}$, $[\![\alpha_\mathsf{v}]\!]^{\mathbf{B}}$, and β_v) to the party P towards which the output needs to be reconstructed. Since we are in the honest majority setting, there will be a majority value among each of the commitment which the party P accepts. In the next round, servers open the shares towards party P as follows: P_0, P_1 open $[\alpha_\mathsf{v}]_1$; P_0, P_2 open $[\alpha_\mathsf{v}]_2$; P_1, P_2 open β_v. For each of share, party P will accept the opening that matches with the commitment that it accepted.

APPENDIX D
MICRO BENCHMARKING OVER WAN

In this section, we provide detailed benchmarking of ML algorithms over the WAN setting.

A. ML Training

In Table X, we tabulate the performance in the preprocessing phase of the protocol of BLAZE and ABY3 for Linear Regression and Logistic Regression Training. The data for batch size B $\in \{128, 256, 512\}$ and feature sizes $\{100, 500, 900\}$ are provided. The values in the table shows the number of iterations in the preprocessing phase that can be completed in a minute. A higher value in the table corresponds a protocol with lower latency.

Similarly, in Table XI, we tabulate the performance in the online phase of the protocol of BLAZE and ABY3 for Linear Regression and Logistic Regression Training. The values in the table shows the number of online iterations that can be completed in a minute.

Algorithm	Batch Size	Ref.	Feature Size		
			n = 100	n = 500	n = 900
Linear Regression (#iterations/min)	128	ABY3	73.50	68.42	64.72
		BLAZE	97.08	97.47	91.72
	256	ABY3	72.38	64.36	55.96
		BLAZE	96.60	91.31	84.38
	512	ABY3	70.23	55.04	43.30
		BLAZE	95.65	83.67	70.55
Logistic Regression (#iterations/min)	128	ABY3	19.77	19.38	19.07
		BLAZE	32.89	32.47	32.25
	256	ABY3	19.68	19.04	18.23
		BLAZE	32.57	31.95	31.06
	512	ABY3	19.52	18.13	16.64
		BLAZE	31.73	30.29	28.38

TABLE X: Preprocessing Phase: Comparison of ABY3 and BLAZE for ML Training (higher = better)

Algorithm	Batch Size	Ref.	Feature Size		
			n = 100	n = 500	n = 900
Linear Regression (#iterations/min)	128	ABY3	97.57	95.07	89.94
		BLAZE	139.05	139.05	139.05
	256	ABY3	97.11	89.94	80.09
		BLAZE	139.05	139.05	139.05
	512	ABY3	95.08	80.10	68.94
		BLAZE	139.05	139.05	139.05
Logistic Regression (#iterations/min)	128	ABY3	20.54	20.43	20.18
		BLAZE	57.20	57.20	57.20
	256	ABY3	20.52	20.18	19.63
		BLAZE	57.14	57.14	57.14
	512	ABY3	20.40	19.61	18.87
		BLAZE	56.72	56.72	56.72

TABLE XI: Online Phase: Comparison of ABY3 and BLAZE for ML Training (higher = better)

B. ML Inference

Here we provide the details of the benchmarking done on the preprocessing phase of ML Inference. The details for Linear Regression, Logistic Regression, and Neural Networks appear in Fig. 21, Fig. 22, and Fig. 23 respectively.

For all of the algorithms above, we observe $\approx 4\times$ gain in the preprocessing throughput over ABY3.

1) Comparison with ASTRA: Here we compare Linear Regression and Logistic Regression inference of BLAZE and ASTRA. For a fair comparison, we apply the optimizations proposed by ASTRA in our protocols. Since Linear Regression inference essentially reduces to a dot product, the benchmarking for the former can be used to analyse the performance of dot product of BLAZE and ASTRA. Hence we omit a separate benchmarking for dot product.

In Fig. 24, we plot the online throughput of BLAZE and ASTRA for Linear Regression (Fig. 24a) and Logistic Regression (Fig. 24b) inference. Concretely, we plot the gain

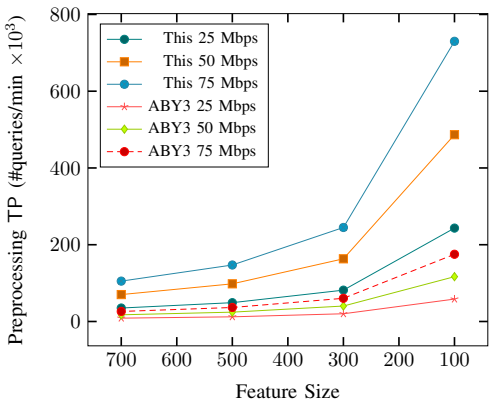


Fig. 21: Comparison of Preprocessing Throughput (TP) of BLAZE and ABY3 for Linear Regression Inference

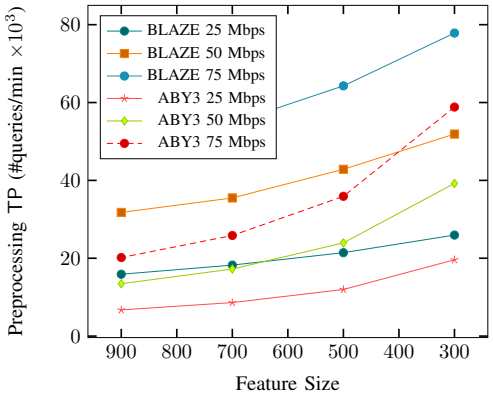


Fig. 22: Comparison of Preprocessing Throughput (TP) of BLAZE and ABY3 for Logistic Regression Inference

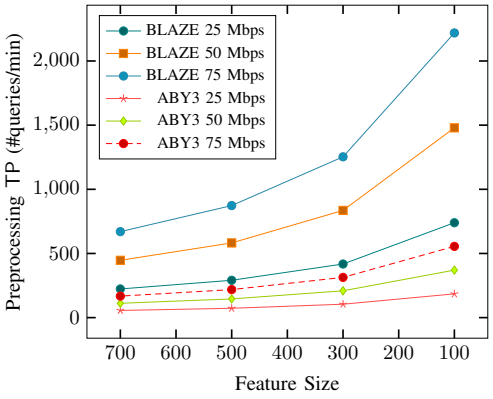
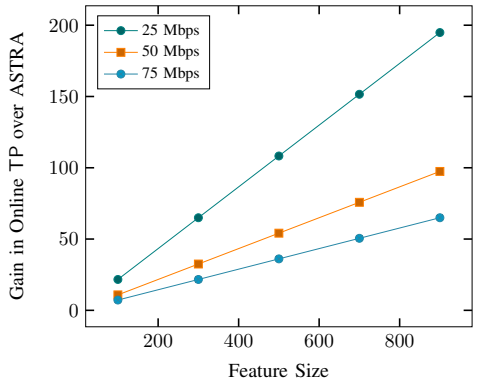


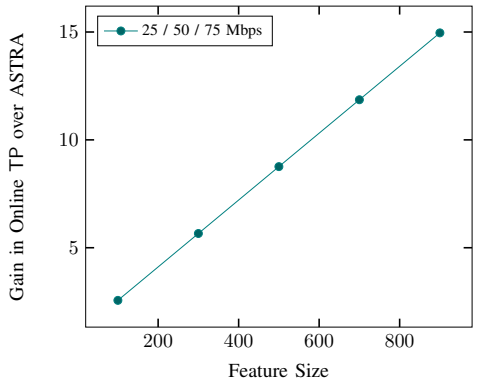
Fig. 23: Comparison of Preprocessing Throughput (TP) of BLAZE and ABY3 for Neural Networks Inference

in online throughput over ASTRA over different batch sizes and bandwidths. For the preprocessing phase, our protocols clearly outperforms that of ASTRA and hence we omit the plot for the same.

For both Linear Regression and Logistic Regression inference, we observe that the gain in online throughput over ASTRA drops with an increase in bandwidth. This is because of our limited processing capacity which prevents our protocols



(a) Linear Regression Inference



(b) Logistic Regression Inference

Fig. 24: Online Throughput (TP) Comparison of ASTRA and BLAZE for Linear Regression and Logistic Regression Inference

from attaining the maximum attainable throughput even for a bandwidth of 25 Mbps. On the other hand, the throughput of ASTRA increases with the increase in bandwidth. To see this, we limited the bandwidth further to 3Mbps. At 3Mbps, the gain in online throughput over ASTRA ranges from $180\times$ to $1623\times$ for Linear Regression inference.