Megha Byali, Harsh Chaudhari*, Arpita Patra, and Ajith Suresh

FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning

Abstract: Privacy-preserving machine learning (PPML) via Secure Multi-party Computation (MPC) has gained momentum in the recent past. Assuming a minimal network of pair-wise private channels, we propose an efficient four-party PPML framework over rings $\mathbb{Z}_{2^\ell}$, FLASH, the first of its kind in the regime of PPML framework, that achieves the strongest security notion of Guaranteed Output Delivery (all parties obtain the output irrespective of adversary's behaviour). The state of the art ML frameworks such as ABY3 by *Mohassel et.al* (ACM CCS'18) and SecureNN by *Wagh et.al* (PETS'19) operate in the setting of 3 parties with one malicious corruption but achieve the *weaker* security guarantee of *abort*. We demonstrate PPML with real-time efficiency, using the following custom-made tools that overcome the limitations of the aforementioned state-of-the-art– (a) *dot product*, which is independent of the vector size unlike the state-of-the-art ABY3, SecureNN and ASTRA by *Chaudhari et.al* (ACM CCSW'19), all of which have linear dependence on the vector size. (b) *Truncation*, which is constant round and free of circuits like Ripple Carry Adder (RCA), unlike ABY3 which uses these circuits and has round complexity of the order of depth of these circuits. We then exhibit the application of our FLASH framework in the secure server-aided prediction of vital algorithms– Linear Regression, Logistic Regression, Deep Neural Networks, and Binarized Neural Networks. We substantiate our theoretical claims through improvement in benchmarks of the aforementioned algorithms when compared with the current best framework ABY3. All the protocols are implemented over a 64-bit ring in LAN and WAN. Our experiments demonstrate that, for MNIST dataset, the improvement (in terms of throughput) ranges from 11× to 1395× over LAN and WAN together.

Keywords: Privacy, Machine Learning, Robust 4PC

Megha Byali: Indian Institute of Science, E-mail: megha@iisc.ac.in
***Corresponding Author: Harsh Chaudhari:** Indian Institute of Science, E-mail: chaudharim@iisc.ac.in

1 Introduction

Secure Multi-party Computation (MPC) [Yao82, BGW88, GMW87, IKNP03, DPSZ12] has evolved over the years in its pursuit of enabling a set of n mutually distrusting parties to compute a joint function f, in a way that no coalition of t parties can disrupt the true output of computation (correctness) or learn any information beyond what is revealed by the output of the computation (privacy). The area of secure MPC can be broadly categorized into honest majority [BGW88, MRZ15, ABF+16, BJPR18] and dishonest majority [Yao82, DPSZ12, DKL+13, MF06, GMW87]. Over the years, MPC has progressed from being simply of theoretical interest to providing real-time practical efficiency. In terms of efficient constructions, the special case of dishonest-majority setting, namely two-party computation (2PC) [Yao82, LP07, Lin16, NO16] has been in limelight over the last decade. However lately, the setting of three parties (3PC) [ABF+17, ABF+16, MRZ15, BJPR18] and four parties (4PC) [IKKPC15, BJPR18, GRW18] have drawn phenomenal attention due to the customization in techniques and efficiency that the constructions have to offer. In this direction, the area of MPC in a small domain with an honest majority is quite fascinating due to variety of reasons mentioned below.

First, the most widely known real-time applications such as Danish Sugar-Beet Auction [BCD+09], Distributed Credential Encryption [MRZ15], Fair-play MPC [BNP08], VIFF [Gei07], Sharemind [BLW08] explore MPC with 3 parties. Second, the expensive public-key primitives such as Oblivious Transfer (OT) known to be necessary for 2PC can be eliminated in the honest majority. Thus, the resulting constructions use only light-weight primitives and can even be information-theoretically secure. Third, the recent advances in se-

Arpita Patra: Indian Institute of Science. This author is supported by SERB Women Excellence Award 2017 (DSTO 1706). E-mail: arpita@iisc.ac.in
Ajith Suresh: Indian Institute of Science, E-mail: ajith@iisc.ac.in

BY-NC-ND

cure Machine Learning (ML) have indicated real-time applications involving a small number of parties [MZ17, MRSV18, WGC19, MR18, CCPS19, ÁMJ+18, AFS19]. Furthermore, the stronger security notions of fairness (the adversary gets the output if and only if the honest parties do) and robustness aka guaranteed output delivery (GOD) (all parties obtain the output irrespective of adversary's behaviour) are guaranteed only in the honest majority setting [Cle86].

In this work, we strongly motivate the need for robustness in privacy-preserving machine learning as a service (MLaaS) and then go on to explore the setting of 4PC and demonstrate that our constructions are highly efficient compared to the existing state of the art 3PC ML frameworks. The guarantee of robustness is of utmost importance in the area of MLaaS. Consider the following scenario where an entity owns a trained ML model and wants to provide prediction as a service. The model owner outsources her trained model parameters to a set of three servers, which uses one of the aforementioned 3PC ML frameworks for secure prediction. These frameworks keep the privacy of the model parameters and the queries of the clients intact even when one of the servers is maliciously corrupted, but cannot guarantee an output to a given client's query as the adversary can cause the protocol to abort. Thus in the practical setting, one simple strategy of the adversary would be to make the protocol abort for all the client queries. Eventually, this would steer the entity towards loss of monetary value and trust of the clients.

Motivation for 4PC Framework: The specific problem of MPC with 4-parties tolerating one corruption is of special interest to us. There are three primary motivations for us to consider this setting for achieving GOD– (a) avoid theoretical necessity of broadcast channel; (b) avoid expansive public-key primitives and (c) communication efficiency. We elaborate these points below. The popular setting of 3PC, when considered to achieve robustness, suffers from the necessity of an expensive robust broadcast channel as proven in the result of [CL14]. By moving to 4PC from 3PC, the need for a broadcast channel is removed, which results in highly efficient constructions [BJPR18] when compared to 3PC [CCPS19, MR18, WGC19]. Additionally in 4PC, for any message sent by a party that needs an agreement, a simple honest majority rule over the residual three parties suffices. Such a property cannot be counted on in 3PC which leads to the use of costly workarounds than 4PC. [GRW18] was the most recent work to propose guaranteed output delivery (robustness) in the 4PC setting. A major concern with GOD variant of multiplication protocol in [GRW18] was utilizing Digital Signatures and expensive public-key primitives: Broadcast and a PKI Setup. Since our end goal is an efficient and robust framework for ML, we let go their approach and propose a simple primitive coupled with a new secret sharing scheme which requires only symmetric-key primitives to achieve robustness.

Moreover, the state-of-the-art 3PC ML frameworks, like ABY3 and ASTRA, focused on highly efficient frameworks for machine learning in the semi-honest setting but suffered from efficiency loss for the primitives dot product, MSB extraction, and truncation in the malicious setting. For example, many of the widely used ML algorithms like Linear Regression, Logistic Regression, and Neural Networks use dot product computation as its building block. While the above frameworks incur a communication cost which is linearly dependent on the underlying size of the feature vector, we are able to eliminate this limitation and provide a dot product protocol whose communication is independent of the vector size. Additionally, we also make all our building blocks constant round and free of any circuits, unlike ABY3 which uses expensive non-constant round circuits like Parallel Prefix Adder (PPA) and Ripple Carry Adder (RCA) in their protocols.

Lastly, we choose build our framework over rings. Most of the computer architectures, Intel x64 for example, have their primitive data-types over rings. These architectures have specially designed hardware which can support fast and efficient arithmetic operations over rings. This led the way for efficient protocols over rings [BLW08, DOS18, ABF+17, EOP+19, CCPS19, BBC+19] as opposed to fields, which are usually 10-20x slower since they have to rely on external libraries. Thus, our protocols over rings give the additional advantage of faster performance when implemented in the real-world architectures.

1.1 Our Contribution

We propose FLASH, the first robust framework for privacy-preserving machine learning in the four party (4PC) honest majority setting over a ring $\mathbb{Z}_{2^\ell}$. We summarize our contributions below:

Robust 4PC protocol: We present an efficient and robust MPC protocol for four parties tolerating one malicious corruption. Concretely, for the multiplication operation, we require an overall communication of just 12 elements in the amortized sense. This is $\approx 2\times$ im-

Protocol	Equation	ABY3		ASTRA		FLASH	
		Rounds	Comm.	Rounds	Comm.	Rounds	Comm.
Multiplication	$[\![x]\!].[\![y]\!] \rightarrow [\![x.y]\!]$	5	21ℓ	7	25ℓ	5	12ℓ
Dot Product	$[\![\vec{x} \odot \vec{y}]\!] = [\![\sum_{i=1}^{d} x_i y_i]\!]$	5	$21m\ell$	7	$23m\ell + 2\ell$	5	12ℓ
MSB Extraction	$[\![x]\!] \rightarrow [\![\mathsf{msb}(x)]\!]^{\mathbf{B}}$	$\log \ell + 5$	63ℓ	6	$\approx 9\kappa\ell$	$\log \ell + 5$	28ℓ
Truncation	$[\![x]\!].[\![y]\!] \rightarrow [\![(xy)^t]\!]$	$2\ell - 1$	$\approx 108\ell$	–	–	5	14ℓ
Bit Conversion	$[\![b]\!]^{\mathbf{B}} \rightarrow [\![b]\!]$	6	42ℓ	–	–	5	14ℓ
Bit Insertion	$[\![b]\!]^{\mathbf{B}}[\![x]\!] \rightarrow [\![bx]\!]$	7	63ℓ	–	–	5	18ℓ

Table 1. Comparison of FLASH framework with ABY3 and ASTRA; ℓ, κ and m denote the ring size, security parameter and number of features respectively.

provement in terms of communication over the state-of-the-art protocol of [GRW18]. Moreover, our solution forgoes the need for Digital Signatures and expensive primitives like Broadcast and Public-Key Setup, unlike [GRW18]. The removal of this additional setup of Digital Signatures, PKI and Broadcast primarily comes from two factors – i) a new secret sharing scheme which we call as *mirrored*-sharing, enables two disjoint sets of parties to perform the computation and perform an effective validation in a single execution, and ii) a simple yet novel *bi-convey* primitive, which enables two designated parties, say S_1, S_2, to send a value to a designated party R with the help of a fourth party T.

The bi-convey primitive guarantees that if both S_1 and S_2 are honest, then party R will receive the value x for sure. If not, either the party R will be able to obtain x or both the parties R and T identify that one among S_1, S_2 is corrupt. Our construction for the bi-convey primitive requires a commitment scheme as the only cryptographic tool, which is considered inexpensive. Moreover, the commitments can be clubbed together for several instances and thus the cost of commitment gets amortized as well. Looking ahead, most of our constructions are designed in such a way that every message to be communicated will be made available to at least two parties and thus we can use the bi-convey primitive for the same.

Building Blocks for Machine Learning: We propose practically efficient building blocks that form the base for secure prediction. While ABY3 and SecureNN propose building blocks for security with abort, ASTRA elevates the security of these blocks from abort to fairness. We further strengthen the security and make all the building blocks robust. Additionally, we achieve significant efficiency improvements in all the building blocks due to the aid provided by an additional honest party in our setting. The improvements for each block are summarized as follows:

i) Dot Product: The aforementioned 3PC frameworks involve communication, linear in the order of vector size, we overcome this limitation with an efficient technique, independent of the vector size. This independence stems from the peculiar structure of our mirrored sharing alongside the multiplication protocol in 4PC.

ii) Truncation: Overflow caused by repeated multiplications may cause accuracy loss which can be prevented with truncation. Truncation has been expensive in the 3PC framework, especially ABY3 uses a Ripple Carry Adder (RCA) circuit which consumes around 108 ring elements to achieve MSB Extraction. We propose a simple yet efficient technique with a total of just 14 ring elements and does not require any circuits. The technical novelty comes from the specific roles played by the parties, in conjunction with the multiplication protocol of 4PC. We defer the detailed analysis of our truncation protocol and the corresponding roles of the parties to Section 5.4.

iii) MSB Extraction: Comparing two arithmetic values in a privacy-preserving manner is one of the major hurdles in realizing efficient privacy-preserving ML algorithms. The state of the art SecureML[MZ17] and ABY3 made an effort in this direction with the use of a garbled circuit technique and a boolean parallel prefix adder (PPA) respectively. We extend the technique proposed by ABY3 of using a boolean PPA circuit for our 4PC setting.

iv) Bit Conversion and Insertion: Operating interchangeably in the arithmetic and boolean worlds often demand conversion of a boolean bit to its arithmetic equivalent (*bit conversion*) or the multiplication of a boolean bit with an arithmetic value (*bit insertion*). We propose efficient techniques to achieve the same with innovations coming from our mirrored secret sharing and its linearity property. Ours is the first work in 4PC that proposes these transformations and is even superior to the state-of-the-art 3PC ML frameworks ABY3 and AS-

TRA, in terms of both efficiency and security guarantee. Table 1 provides a detailed comparison in terms of communication (Comm.) and rounds with ABY3 and ASTRA, where ℓ and m denote the ring size and number of features respectively.

Secure Prediction: We aim at secure prediction in a server-aided setting. Here, the model owner (M) holds a set of *trained* model parameters which are used to predict output to client's (C) input query, while preserving the privacy of the inputs of both the parties. The servers perform computation and reconstruct the output towards the client. Security is provided against a malicious adversary corrupting one server along with either model owner or client. We extend our techniques for vital machine learning algorithms namely: i) Linear Regression, ii) Logistic Regression, iii) Deep Neural Network (DNN) and iv) Binarized Neural Network (BNN). While Linear Regression is extensively used in Market Analytics, Logistic Regression is used in a variety of applications like customer segmentation, insurance fraud detection and so on. Despite being computationally cheap and smaller in size, the performance accuracy of BNNs is comparable to that of deep neural networks. They are the go-to networks for running neural networks on low-end devices. These use cases exhibit the importance of these algorithms in real-time and we make an effort to efficiently perform the secure evaluation for these algorithms.

ML Algorithm	**Setting**	
	LAN	**WAN**
Linear Regression	1395×	124×
Logistic Regression	400×	29×
Deep Neural Network	314×	13×
Binarized Neural Network	268×	11.5×

Table 2. Improvement over ABY3 in terms of throughput for MNIST dataset

We provide implementation results for all our protocols over a ring $\mathbb{Z}_{2^{64}}$. We summarize the efficiency gain of our protocols over the state-of-the-art ABY3 and ASTRA, albeit more elaborate details follow in Section 6. The latency and throughput (the number of operations per unit time) of the protocols are measured in the LAN (1Gbps) and WAN (20Mbps) setting while communication complexity is measured independent of the network. We compare the most crucial building blocks, namely i) Dot Product, ii) MSB Extraction and iii) Truncation of our framework with state-of-the-art and show the practical improvement observed in each of the building blocks. We also provide throughput comparisons (# queries per sec in LAN and # queries per min in WAN) for the aforementioned algorithms, over multiple real-world datasets. Table 2 below shows the improvement over ABY3 for MNIST dataset [LC10] which has 784 features. We omit comparison with ASTRA as ABY3 outperforms ASTRA in terms of total communication (ref. Table 1). The improvements for DNN and BNN stated in Table 2 are for a network having 2 hidden layers, each layer consisting of 128 nodes.

4PC Abort: As an extension, we also propose protocols for the weaker abort setting. The abort variant for the protocols are achieved by tweaking the bi-convey primitive present in the robust protocols. We give a detailed analysis and comparison with state-of-the-art works in Appendix A.

1.2 Related Work:

In the regime of MPC over a small domain, interesting works that achieve guaranteed output delivery have been carried out mainly in the class of low-latency (consisting of small constant number of rounds) protocols [PR18, BJPR18, BHPS19]. However, in the view of practical efficiency, high throughput (light in communication and computation complexity) is desirable. Yet the literature of high throughput protocols has seen limited work [GRW18] in guaranteeing security notions stronger than abort. The existing state-of-the-art includes notable works that are highly efficient, but trade security for efficiency [ABF+17, AFL+16, ABF+16, CGH+18, FLNW17, NV18]. In this work, we attempt to bridge the gap between the security achieved and the corresponding efficiency, thus providing highly efficient PPML framework using robust 4PC as the backbone. Below we summarize the contributions closest to our setting.

The study of MPC in high-throughput networks accelerated with the celebrated work of [DSZ15]. The works of [ABF+17, AFL+16, ABF+16, CGH+18, FLNW17, NV18] swiftly followed. These works focus on the evaluation of arithmetic circuits over rings or finite fields. [AFL+16] is semi-honest and operates over both rings and fields. The works of [ABF+17, FLNW17, DOS18] achieve abort security over rings with one malicious corruption. A compiler to transform semi-honest security to malicious-security was proposed by [CGH+18]. This conversion is obtained at twice the cost of the semi-honest protocol. The work of [GRW18] explores 4PC and the security notions of fairness and guaranteed output delivery. However, [GRW18] is dual execution based and relies on expensive public-key primitives and broadcast channel to achieve guaranteed output delivery. [NV18] improvises over [CGH$^+$18] by presenting a batch multiplication technique and additionally explores the notion of fairness.

The influence of ML has found its way in a broad range of areas such as facial recognition [SKP15], banking, medicine [EKN$^+$17], recommendation systems and so on. Consequently, technology giants such as Amazon, Google are providing ML as a service (MLaaS) for both training and prediction purposes, where the parties outsource their computation to a set of servers. However, for confidential purposes, government regulations and competitive edge, such data cannot be made publicly available. Thus, there is a need for privacy of data while still enabling customers to perform training and prediction. This need for privacy has given rise to the culmination of MPC and ML. Recent works [MZ17, MR18, MRSV18, WGC19, CCPS19, RWT$^+$18] have shown the need of MPC in achieving efficient techniques for privacy-preserving machine learning in server aided setting, where parties outsource their data to a set of servers and the servers compute for purposes of training or classification. There have been works dedicated to linear regression [MR18, CCPS19, MZ17], logistic regression [MR18, CCPS19, MZ17] and neural networks [MR18, MZ17, WGC19, JVC18, RWT$^+$18] for both training and inference. Recent works have dived into variants of neural networks like Deep Neural Networks (DNNs) [MR18, MMH$^+$19, RWT$^+$18], Convolutional Neural Networks (CNNs) [WGC19, RWT$^+$18, JVC18], Binarized Neural Networks (BNNs) [KCY$^+$18] and Quantized Neural Networks (QNNs) [ADAM19, JBAP19]. DNNs and CNNs have become one of the most powerful machine learning models in recent history with amount of data available to train them and are one of the most widely considered models for training and prediction tasks for low power devices. MOBIUS [KCY$^+$18] was the first to explore secure prediction in BNNs for semi-honest 2PC.

2 Preliminaries and Definitions

We consider a set of four parties $\mathcal{P} = \{\mathsf{V}_1, \mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2\}$ connected by pair-wise private and authentic channels in a synchronous network. E_1, E_2 define the role of the parties as *evaluators* in the computation while parties V_1, V_2 enact the role of *verifiers* in the computation. We use $\mathbf{E}$ and $\mathbf{V}$ to denote the set of evaluators $\{\mathsf{E}_1, \mathsf{E}_2\}$ and verifiers $\{\mathsf{V}_1, \mathsf{V}_2\}$ respectively. The function f to be evaluated is expressed as a circuit ckt, with a publicly known topology and is evaluated over either an arithmetic ring $\mathbb{Z}_{2^\ell}$ or a Boolean ring $\mathbb{Z}_{2^1}$, consisting of 2-input addition and multiplication gates. d denotes the multiplicative depth of ckt.

We use a collision-resistant hash function, denoted by $\mathsf{H}()$ and a commitment scheme, denoted by $\mathsf{com}()$, in our protocols for practical efficiency. The details of the same can be found in Section B.1.

Security Model: For MPC, each party is modelled as a non-uniform probabilistic polynomial time (PPT) interactive Turing Machine. We operate in a static security model with an honest majority, where a PPT adversary $\mathcal{A}$ can corrupt a party at the onset of the protocol. $\mathcal{A}$ can be malicious in our setting i.e., the corrupt parties can arbitrarily deviate from the protocol specification. The computational security parameter is denoted by κ.

Robustness or Guaranteed Output Delivery: A protocol is said to be *robust* if all the parties can compute the output of the protocol irrespective of the behaviour of the adversary. The security of our protocols is proved in the standard real/ideal world paradigm. The details for the ideal world functionality $\mathcal{F}_{\mathsf{robust}}$ that realizes the same in the 4PC setting is presented in Fig 18.

Shared Key Setup: We adopt a one-time key setup to minimize the overall communication of the protocol. We use three types of key setup namely, between i) a pair of parties, ii) a committee of three parties and iii) all the four parties. In each type, the parties in consideration can run an MPC protocol to agree on a randomness and use it as the key for pseudo-random function (PRF) to derive any subsequent co-related randomness. We model the protocol for the shared key setup as functionality $\mathcal{F}_{\mathsf{setup}}$ and is presented in Fig 16.

3 Sharing Semantics

We use additive secret sharing of secrets over either an arithmetic ring $\mathbb{Z}_{2^\ell}$ or a Boolean ring $\mathbb{Z}_{2^1}$. We define two variants of secret sharing that are used in this work.

- **Additive sharing ($[\cdot]$-sharing):** A value x is additively shared between two parties if $x = x^1 + x^2$, where one party holds the first share x^1 while the other party holds x^2. We use $[x] = (x^1, x^2)$ to denote $[\cdot]$-sharing of x.

• **Mirrored sharing ($[\![\cdot]\!]$-sharing):** A value x is said to be $[\![\cdot]\!]$-shared among the parties in $\mathcal{P}$ if:

– There exist values σ_x, μ_x such that $\mu_x = x + \sigma_x$.

– σ_x is $[\cdot]$-shared among parties in $\mathbf{E}$ as $[\sigma_x]_{\mathsf{E}_1} = \sigma_x^1$ and $[\sigma_x]_{\mathsf{E}_2} = \sigma_x^2$, while parties in $\mathbf{V}$ hold both σ_x^1 and σ_x^2.

– μ_x is $[\cdot]$-shared among parties in $\mathbf{V}$ as $[\mu_x]_{\mathsf{V}_1} = \mu_x^1$ and $[\mu_x]_{\mathsf{V}_2} = \mu_x^2$, while parties in $\mathbf{E}$ hold both μ_x^1 and μ_x^2.

The shares of each party can be summarized as:

$\mathsf{E}_1 : [\![x]\!]_{\mathsf{E}_1} = (\sigma_x^1, \mu_x^1, \mu_x^2)$	$\mathsf{V}_1 : [\![x]\!]_{\mathsf{V}_1} = (\sigma_x^1, \sigma_x^2, \mu_x^1)$
$\mathsf{E}_2 : [\![x]\!]_{\mathsf{E}_2} = (\sigma_x^2, \mu_x^1, \mu_x^2)$	$\mathsf{V}_2 : [\![x]\!]_{\mathsf{V}_2} = (\sigma_x^1, \sigma_x^2, \mu_x^2)$

We use the notation $[\![x]\!] = ([\sigma_x], [\mu_x])$ to denote $[\![\cdot]\!]$-sharing of value x. Sharing techniques and protocols for the boolean variant ($\mathbb{Z}_{2^1}$) are identical to their arithmetic counterparts apart from addition and subtraction operations being replaced with XOR and multiplication with AND. We use $[\![\cdot]\!]^{\mathbf{B}}$ to denote the sharing over a boolean ring.

• **Linearity of $[\cdot]$-sharing and $[\![\cdot]\!]$-sharing:** Given $[x] = (x^1, x^2)$, $[y] = (y^1, y^2)$ and public constants $c_1, c_2 \in \mathbb{Z}_{2^\ell}$, we have

$$[c_1 x + c_2 y] = (c_1 x^1 + c_2 y^1, c_1 x^2 + c_2 y^2) = c_1[x] + c_2[y]$$

Thus, $[c_1 x + c_2 y]$ and $c_1[x] + c_2[y]$ are equivalent and implies that parties can compute shares of any linear function of $[\cdot]$-shared values locally. It is easy to see that the linearity property extends to our $[\![\cdot]\!]$-sharing as well.

4 Robust 4PC

In this section, we present a robust and efficient 4PC protocol with security against one malicious adversary. Our protocol incurs 12 ring elements per multiplication and removes the need for any additional setup of Broadcast, Digital Signatures, and Public-Key Setup, unlike [GRW18]. We begin this section by introducing a "bi-convey primitive", which forms the core for the majority of our constructions. As mentioned in the introduction, bi-convey primitive enables two designated parties to send a value x to the third party with the aid of the fourth party. The remainder of the section describes a high-level overview of our protocol which is divided into 3 stages– i) input sharing, ii) circuit evaluation and iii) output computation. We elaborate on our primitive and each of the stages below:

4.1 Bi-Convey Primitive

Bi-convey primitive enables either i) two parties, say S_1, S_2, to convey a value $x \in \mathbb{Z}_{2^\ell}$ to a designated party R or ii) allows party R to identify that one among S_1, S_2 is corrupt. The technical innovation of our construction for the 4 party case lies in using the fourth party available, say T, in an efficient manner. To elaborate, the protocol proceeds as follows. Parties S_1, S_2 both send the value x to R. In parallel, they send a commitment of the same ($\mathsf{com}(x)$) to the fourth party T. Note that the randomness used to prepare the commitment is picked from the common source of the randomness of S_1, S_2 and R. If the received copies of x match, party R accepts the value and sends `continue` to T, and discards any message received from T. If not, R will identify that one among (S_1, S_2) is corrupt and thus T is honest. She then sends her internal randomness to T and waits for a message from T. Note that, the internal randomness of R which is forwarded to T, in our setting are all the keys of R (established during the shared key setup phase) that are not available with T. Party T, on the other hand, first checks if the commitments received from S_1, S_2 match or not. If they match, she will forward $\mathsf{com}(x)$ to R else, she will identify that one among (S_1, S_2) is corrupt and thus sends her internal randomness to R. Now, if R receives $\mathsf{com}(x)$ from T, then she will accept the version of x that matches with the received $\mathsf{com}(x)$ and stops. If not, then both R and T have identified that one among (S_1, S_2) is corrupt.

$\mathcal{F}_{\mathsf{bic}}$ receives x, x', I_R and I_T from the parties S_1, S_2, R and T respectively. Here I_R and I_T denote the internal randomness of parties R and T respectively. $\mathcal{F}_{\mathsf{bic}}$ sets $\mathsf{msg}_{S_1} = \mathsf{msg}_{S_2} = \bot$.

– If $x = x'$, then $\mathcal{F}_{\mathsf{bic}}$ sets $\mathsf{msg}_T = \bot$ and $\mathsf{msg}_R = x$. Else it sets $\mathsf{msg}_T = \mathsf{I}_R$ and $\mathsf{msg}_R = \mathsf{I}_T$.

– $\mathcal{F}_{\mathsf{bic}}$ sends $\mathsf{msg}_{S_1}, \mathsf{msg}_{S_2}, \mathsf{msg}_R$ and msg_T to parties S_1, S_2, R and T respectively.

Fig. 1. Functionality $\mathcal{F}_{\mathsf{bic}}$: Ideal Functionality for party R to receive value x from S_1 and S_2.

The formal protocol appears in Fig 2 and the details for corresponding ideal world functionality $\mathcal{F}_{\mathsf{bic}}$ appears in Fig 1.

• **Input:** Parties S_1, S_2, R and T input x, x, I_R and I_T respectively.

• **Output:** Parties S_1, S_2 receive $\bot$. Parties R and T receive x and $\bot$ as outputs respectively, when S_1, S_2 are honest. For the case when one among S_1, S_2 is corrupt, party R obtains either x or I_T, while party T obtains either I_R or $\bot$, depending on the adversary's strategy.

– Parties S_1, S_2 send the value x to party R. In parallel, S_1, S_2 compute commitment of x, $\mathsf{com}(x)$, using shared randomness known to R as well (sampled from the key shared amongst S_1, S_2 and R established during the shared key setup phase) and send it to T.
– If the received values match, party R sets $\mathsf{msg}_R = \texttt{continue}$, accept the value x and discard any further message from T. Else, he sets $\mathsf{msg}_R = \mathsf{I}_R$, where I_R denotes the internal randomness of R.
– If the received commitments match, party T sets $\mathsf{msg}_T = \mathsf{com}(x)$, else sets $\mathsf{msg}_T = \mathsf{I}_T$, where I_T denotes the internal randomness of T.
– Parties R and T mutually exchange the msg values.
– If $\mathsf{msg}_R = \mathsf{I}_R$ and $\mathsf{msg}_T = \mathsf{com}(x)$, then R accepts the value x that is consistent with $\mathsf{com}(x)$.

Fig. 2. $\Pi_{\mathsf{bic}}(S_1, S_2, x, R, T)$: Protocol for S_1, S_2 to convey a value x to R with the help of T

We now provide a brief motivation for the need of bi-convey primitive in our framework. Looking ahead, the bi-convey primitive is used as a black-box in almost all of our subsequent protocol constructions. Consider the case where a call to this primitive from the outer protocol results in exchange of internal randomness among two parties. This implies both the parties conclude one among the remaining parties is corrupt and can safely trust each other. Thus both the honest parties combined, act as a single trusted party and use the received randomness to compute the inputs of all the parties in clear. Note that, both the honest parties together are able to compute the inputs in clear primarily because of the specific design of our mirrored sharing format (Section 3) where two parties together posses all the shares to reconstruct the inputs of the circuit. The honest parties then compute the final circuit output and send it to the remaining two parties ensuring guaranteed output delivery. We give a more detailed explanation of a use case of bi-convey primitive fitting in a larger protocol in Section 4.3.

4.2 Input Sharing

The goal is to robustly generate a $[\![\cdot]\!]$-sharing of a party's input. We call a party who wants to share the input as a Dealer. On a high level, if a dealer D wants to share a value x, parties start by locally sampling σ_x^1, σ_x^2 and μ_x^1, according to the defined sharing semantics. The dealer then sets the last share as $\mu_x^2 = x + \sigma_x - \mu_x^1$. In case when the dealer is a verifier (say V_1), we enforce V_1 to send μ_x^2 to both the evaluators and $\mathsf{com}(\mu_x^2)$ to V_2. Now, all parties except V_1, exchange $\mathsf{com}(\mu_x^2)$ and compute the majority. If there exists no majority then V_1 is known to be corrupt and eliminated from the computation. The remaining parties can then run a semi-honest three-party protocol to compute the output. A similar idea follows for the case when the dealer is an evaluator. We provide the formal details of our Π_{sh} in Fig 3 below.

- **Input:** Party D inputs value x while others input $\perp$.
- **Output:** Parties obtain $[\![x]\!]$ as the output.

– **If $\mathsf{D} = \mathsf{E}_1$:** Parties in $\mathbf{V}$ and E_1 locally sample σ_x^1, while all the parties in $\mathcal{P}$ locally sample σ_x^2. Parties in $\mathbf{V}$ and E_1 locally compute $\sigma_x = \sigma_x^1 + \sigma_x^2$. Similar steps are done for $D = \mathsf{E}_2$.
– **If $\mathsf{D} = \mathsf{V}_i$ for $i \in \{1,2\}$:** Parties in $\mathbf{V}$ and E_1 locally sample σ_x^1, while parties in $\mathbf{V}$ and E_2 locally sample σ_x^2. Parties in $\mathbf{V}$ locally compute $\sigma_x = \sigma_x^1 + \sigma_x^2$.
– **If $\mathsf{D} = \mathsf{V}_1$:** Party V_1 computes $\mu_x = x + \sigma_x$. Parties in $\mathbf{E}$ and V_1 locally sample μ_x^1. Party V_1 computes and sends $\mu_x^2 = \mu_x - \mu_x^1$ to parties in $\mathbf{E}$ and V_2. Parties in $\mathbf{E}$ and V_2 exchange the received copy of μ_x^2. If there exists no majority, then they identify V_1 to be corrupt and engage in semi-honest 3PC excluding V_1 (with default input for V_1). Else, they set μ_x^2 to the computed majority. Similar steps are done for $D = \mathsf{V}_2$.
– **If $\mathsf{D} = \mathsf{E}_i$ for $i \in \{1,2\}$:** Party E_i computes $\mu_x = x + \sigma_x$. Parties in $\mathbf{E}$ and V_1 locally sample μ_x^1. Party E_i computes and sends $\mu_x^2 = \mu_x - \mu_x^1$ to V_2 and the co-evaluator. E_i sends $\mathsf{com}(\mu_x^2)$ to V_1. Parties other than the dealer exchange the commitment of μ_x^2 to compute majority (the co-evaluator and V_2 also exchange their copies of μ_x^2). If no majority exists, then they identify E_i to be corrupt and engage in semi-honest 3PC excluding E_i (with default input for E_i). Else, they set μ_x^2 to the computed majority.

Fig. 3. $\Pi_{\mathsf{sh}}(\mathsf{D}, x)$: Protocol to generate $[\![x]\!]$ by dealer D.

4.3 Circuit Evaluation

The circuit is evaluated in topological order where for every gate g the following invariant is maintained: given the $[\![\cdot]\!]$-sharing of the inputs, the output is generated in the $[\![\cdot]\!]$-shared format. When g is an addition gate ($\mathsf{z} = \mathsf{x} + \mathsf{y}$), the linearity of $[\![\cdot]\!]$-sharing suffices to maintain this invariant.

- **Input:** Parties input their $[\![\mathsf{x}]\!]$ and $[\![\mathsf{y}]\!]$ shares.
- **Output:** Parties obtain $[\![\mathsf{z}]\!]$ as the output, where $\mathsf{z} = \mathsf{xy}$.

– Parties in $\mathbf{V}$ and E_1 collectively sample σ_{z}^1 and δ_{xy}^1, while parties in $\mathbf{V}$ and E_2 together sample σ_{z}^2.
– Verifiers $\mathsf{V}_1, \mathsf{V}_2$ compute $\delta_{\mathsf{xy}} = \sigma_{\mathsf{x}}\sigma_{\mathsf{y}}$, set $\delta_{\mathsf{xy}}^2 = \delta_{\mathsf{xy}} - \delta_{\mathsf{xy}}^1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{V}_2, \delta_{\mathsf{xy}}^2, \mathsf{E}_2, \mathsf{E}_1)$, which makes sure that E_2 receives δ_{xy}^2.

– Parties in $\mathbf{V}$ and E_1 collectively sample Δ_1. Parties V_1 and E_1 compute $\mathsf{A}_1 = -\mu_{\mathsf{x}}^1\sigma_{\mathsf{y}}^1 - \mu_{\mathsf{y}}^1\sigma_{\mathsf{x}}^1 + \delta_{\mathsf{xy}}^1 + \sigma_{\mathsf{z}}^1 + \Delta_1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_1, \mathsf{A}_1, \mathsf{E}_2, \mathsf{V}_2)$, such that E_2 receives A_1.
– Similarly, parties in $\mathbf{V}$ and E_2 collectively sample Δ_2. Parties V_1 and E_2 compute $\mathsf{A}_2 = -\mu_{\mathsf{x}}^1\sigma_{\mathsf{y}}^2 - \mu_{\mathsf{y}}^1\sigma_{\mathsf{x}}^2 + \delta_{\mathsf{xy}}^2 + \sigma_{\mathsf{z}}^2 + \Delta_2$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_2, \mathsf{A}_2, \mathsf{E}_1, \mathsf{V}_2)$, such that E_1 receives A_2.
– Parties V_2 and E_1 compute $\mathsf{B}_1 = -\mu_{\mathsf{x}}^2\sigma_{\mathsf{y}}^1 - \mu_{\mathsf{y}}^2\sigma_{\mathsf{x}}^1 - \Delta_1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_1, \mathsf{B}_1, \mathsf{E}_2, \mathsf{V}_1)$. Similarly, V_2 and E_2 compute $\mathsf{B}_2 = -\mu_{\mathsf{x}}^2\sigma_{\mathsf{y}}^2 - \mu_{\mathsf{y}}^2\sigma_{\mathsf{x}}^2 - \Delta_2$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_2, \mathsf{B}_2, \mathsf{E}_1, \mathsf{V}_1)$.
– Evaluators compute $\mu_z = \mathsf{A}_1 + \mathsf{A}_2 + \mathsf{B}_1 + \mathsf{B}_2 + \mu_x\mu_y$ locally. Parties in $\mathbf{E}$ and V_1 collectively sample μ_z^1 followed by evaluators setting $\mu_z^2 = \mu_z - \mu_z^1$ and invoking $\Pi_{\mathsf{bic}}(\mathsf{E}_1, \mathsf{E}_2, \mu_z^2, \mathsf{V}_2, \mathsf{V}_1)$for V_2 to receive μ_z^2.

Fig. 4. $\Pi_{\mathsf{mult}}(\mathsf{x}, \mathsf{y}, \mathsf{z})$: Multiplication Protocol

For a multiplication gate g $(\mathsf{z} = \mathsf{xy})$, the goal is for the evaluators to robustly compute μ_{z} where

$$\begin{aligned}\mu_{\mathsf{z}} &= \mathsf{xy} + \sigma_{\mathsf{z}} = (\mu_{\mathsf{x}} - \sigma_{\mathsf{x}})(\mu_{\mathsf{y}} - \sigma_{\mathsf{y}}) + \sigma_{\mathsf{z}} \\ &= \mu_{\mathsf{x}}\mu_{\mathsf{y}} - \mu_{\mathsf{x}}\sigma_{\mathsf{y}} - \mu_{\mathsf{y}}\sigma_{\mathsf{x}} + \sigma_{\mathsf{x}}\sigma_{\mathsf{y}} + \sigma_{\mathsf{z}}\end{aligned}$$

followed by evaluators setting μ_z^2 share and robustly sending it to V_2. On a high level, we view the aforementioned equation of μ_z as: $\mu_z = \mu_{\mathsf{x}}\mu_{\mathsf{y}} + \mathsf{A} + \mathsf{B}$, where $\mathsf{A} = -\mu_{\mathsf{x}}^1\sigma_{\mathsf{y}} - \mu_{\mathsf{y}}^1\sigma_{\mathsf{x}} + \delta_{\mathsf{xy}} + \sigma_{\mathsf{z}} + \Delta$ is solely possessed by V_1 and $\mathsf{B} = -\mu_{\mathsf{x}}^2\sigma_{\mathsf{y}} - \mu_{\mathsf{y}}^2\sigma_{\mathsf{x}} - \Delta$ is possessed V_2. In order for evaluators to compute μ_z, E_1 and E_2 needs to robustly receive $\mathsf{A} + \mathsf{B}$. Note that $\mu_x\mu_y$ is already available with the evaluators. Thus A is further split into $\mathsf{A}_1 + \mathsf{A}_2$, such that each $\mathsf{A}_j \in \{1, 2\}$ is possessed by V_1 and E_j. Similarly, B is split such that each $\mathsf{B}_j \in \{1, 2\}$ is possessed by V_2 and E_j. Now parties need to simply invoke Π_{bic} protocol, one for each A_j and B_j with the co-evaluator acting as the receiving party. Thus evaluators are able to compute $\mathsf{A}+\mathsf{B}$ correctly. After computing μ_{z}, the evaluators set $\mu_{\mathsf{z}}^2 = \mu_{\mathsf{z}} - \mu_{\mathsf{z}}^1$ and call Π_{bic} protocol to send μ_{z}^2 to V_2, where μ_{z}^1 is collectively sampled by parties in $\mathbf{E}$ and V_1. We provide the formal details of our $\Pi_{\mathsf{mult}}(\mathsf{x}, \mathsf{y}, \mathsf{z})$ in Fig 4. For correctness of μ_z,

$$\begin{aligned}\mu_{\mathsf{z}} &= \mathsf{xy} + \sigma_{\mathsf{z}} = (\mu_{\mathsf{x}} - \sigma_{\mathsf{x}})(\mu_{\mathsf{y}} - \sigma_{\mathsf{y}}) + \sigma_{\mathsf{z}} \\ &= \mu_{\mathsf{x}}\mu_{\mathsf{y}} - \mu_{\mathsf{x}}\sigma_{\mathsf{y}} - \mu_{\mathsf{y}}\sigma_{\mathsf{x}} + \sigma_{\mathsf{x}}\sigma_{\mathsf{y}} + \sigma_{\mathsf{z}} \\ &= \ (-\mu_{\mathsf{x}}^1\sigma_{\mathsf{y}} - \mu_{\mathsf{y}}^1\sigma_{\mathsf{x}} + \delta_{\mathsf{xy}}^1 + \sigma_{\mathsf{z}}^1 + \Delta_1 + \Delta_2) \\ &\quad + (-\mu_{\mathsf{x}}^2\sigma_{\mathsf{y}} - \mu_{\mathsf{y}}^2\sigma_{\mathsf{x}} + \delta_{\mathsf{xy}}^2 + \sigma_{\mathsf{z}}^2 - \Delta_1 - \Delta_2) \\ &= \mu_{\mathsf{x}}\mu_{\mathsf{y}} + (\mathsf{A}_1 + \mathsf{A}_2) + (\mathsf{B}_1 + \mathsf{B}_2)\end{aligned}$$

where $\mathsf{A}_j = -\mu_{\mathsf{x}}^1\sigma_{\mathsf{y}}^j - \mu_{\mathsf{y}}^1\sigma_{\mathsf{x}}^j + \delta_{\mathsf{xy}}^j + \sigma_{\mathsf{z}}^j + \Delta_j$ and $\mathsf{B}_j = -\mu_{\mathsf{x}}^2\sigma_{\mathsf{y}}^j - \mu_{\mathsf{y}}^2\sigma_{\mathsf{x}}^j - \Delta_j$ for $j \in \{1, 2\}$. The evaluators receive $\mathsf{A}_1, \mathsf{A}_2, \mathsf{B}_1$ and B_2, whose correctness is guaranteed by Π_{bic} protocol. Thus the evaluators can correctly compute $\mu_{\mathsf{z}} = \mu_{\mathsf{x}}\mu_{\mathsf{y}} + (\mathsf{A}_1 + \mathsf{A}_2) + (\mathsf{B}_1 + \mathsf{B}_2)$. Verifier V_2 also correctly receives μ_z^2 share from the evaluators, by the underlying correctness guarantee of Π_{bic} protocol.

We now analyze how Π_{bic} primitive fits into the larger Π_{mult} protocol to make it robust. Consider Step 2 of the protocol Π_{mult} where parties invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{V}_2, \delta_{\mathsf{xy}}^2, \mathsf{E}_2, \mathsf{E}_1)$. As mentioned in Section 4.1, primitive Π_{bic} guarantees that either i) party E_2 receives the correct value δ_{xy}^2 or ii) both E_1 and E_2 identify that one among $(\mathsf{V}_1, \mathsf{V}_2)$ is corrupt. In the first case, parties can proceed with the execution of the protocol. For the second case, parties E_1 and E_2 mutually exchange their internal randomness (this includes the keys established during the shared key setup phase). Using the received randomness, both E_1 and E_2 can compute the missing part of her share corresponding to the $[\![\cdot]\!]$-sharing of the inputs and hence obtain all the inputs in clear. Given the inputs in clear, both E_1 and E_2 can compute the function output in clear and send it to the remaining two parties.

4.4 Output Computation

The output computation phase is comparatively simple. The missing share of the output with respect to each party is possessed by the remaining three parties. Thus two out of the three parties send the missing share and the third party sends the corresponding hash. Thus each party sets the missing share as the majority among the received values and reconstruct the output. The formal details of our robust output computation protocol Π_{oc} is given in Fig 5.

- **Input:** Parties input their $[\![\mathsf{z}]\!]$ shares.
- **Output:** Parties obtain z as the output.

– For $i, j \in \{1, 2\}$ and $i \neq j$, E_i receives σ_{z}^j from parties in $\mathbf{V}$ and $\mathsf{H}(\sigma_{\mathsf{z}}^j)$ from E_j .
– V_2 receives μ_{z}^1 from parties in $\mathbf{E}$ and $\mathsf{H}(\mu_{\mathsf{z}}^1)$ from V_1.
– V_1 receives μ_{z}^2 from parties in $\mathbf{E}$ and $\mathsf{H}(\mu_{\mathsf{z}}^2)$ from V_2.
– Each party sets the missing share as the majority among the received values and outputs $\mathsf{z} = \mu_{\mathsf{z}}^1 + \mu_{\mathsf{z}}^2 - \sigma_{\mathsf{z}}^1 - \sigma_{\mathsf{z}}^2$.

Fig. 5. Π_{oc}: Protocol for Robust Reconstruction

5 ML Building Blocks

In this section, we provide constructions for our crucial building blocks necessary to achieve secure training and prediction for algorithms namely– i) Linear Regression, ii) Logistic Regression, iii) Deep Neural Network (DNN) and iv) Binarized Neural Network (BNN). We provide the formal details of the corresponding lemmas and proofs to Appendix C.

5.1 Arithmetic/Boolean Couple Sharing Primitive

Two parties, either $\{\mathsf{V}_1, \mathsf{V}_2\}$ (set $\mathbf{V}$) or $\{\mathsf{E}_1, \mathsf{E}_2\}$ (set $\mathbf{E}$) own a common value x and want to create a $[\![\cdot]\!]$- sharing of x. We abstract out this procedure (Fig 6) and define it as *couple sharing* of a value.

Case 1: $(\mathbf{S} = \mathbf{E})$

- **Input:** E_1 and E_2 input x while others input $\perp$.
- **Output:** Parties obtain $[\![x]\!]$ as the output.

– Parties set $\sigma_x^1 = 0$ and $\sigma_x^2 = 0$. Parties in $\mathbf{E}$ and V_1 collectively sample random $\mu_x^1 \in \mathbb{Z}_{2^\ell}$.

– E_1 and E_2 set $\mu_x^2 = x - \mu_x^1$. Parties then execute $\Pi_{\mathsf{bic}}(\mathsf{E}_1, \mathsf{E}_2, \mu_x^2, \mathsf{V}_2, \mathsf{V}_1)$, such that V_2 receives μ_x^2.

Case 2: $(\mathbf{S} = \mathbf{V})$

- **Input:** V_1 and V_2 input x while others input $\perp$.
- **Output:** Parties obtain $[\![x]\!]$ as the output.

– Parties set $\mu_x^1 = 0$ and $\mu_x^2 = 0$. Parties in $\mathbf{V}$ and E_1 collectively sample random $\sigma_x^1 \in \mathbb{Z}_{2^\ell}$.

– V_1 and V_2 set $\sigma_x^2 = x - \sigma_x^1$. Parties then execute $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{V}_2, \sigma_x^2 5, \mathsf{E}_2, \mathsf{E}_1)$, such that E_2 receives σ_x^2.

Fig. 6. $\Pi_{\mathsf{cSh}}(\mathbf{S}, x)$: Protocol to generate couple sharing of x

On a high level when set $\mathbf{S} = \mathbf{E}$, in order to share a value x, parties set $\sigma_x^1 = \sigma_x^2 = 0$. A random μ_x^1 is collectively sampled and the owners of the value set μ_x^2 such that $\mu_x^1 + \mu_x^2 = x$ and send μ_x^2 to V_2 using Π_{bic} protocol. The shares of parties can be viewed as:

$$\begin{array}{ll} \mathsf{E_1} : [\![x]\!]_{\mathsf{E_1}} = (\mathbf{0}, \boldsymbol{\mu_x^1}, \boldsymbol{\mu_x^2}) & \mathsf{V_1} : [\![x]\!]_{\mathsf{V_1}} = (\mathbf{0}, \mathbf{0}, \boldsymbol{\mu_x^1}) \\ \mathsf{E_2} : [\![x]\!]_{\mathsf{E_2}} = (\mathbf{0}, \boldsymbol{\mu_x^1}, \boldsymbol{\mu_x^2}) & \mathsf{V_2} : [\![x]\!]_{\mathsf{V_2}} = (\mathbf{0}, \mathbf{0}, \boldsymbol{\mu_x^2}) \end{array}$$

For the case when set $\mathbf{S} = \mathbf{V}$ and value x, parties in $\mathbf{V}$ and E_1 collectively sample random σ_x^1 followed by $\mathbf{V}$ setting $\sigma_x^2 = -x - \sigma_x^1$ and robustly sending it to E_2.

$$\begin{array}{ll} \mathsf{E_1} : [\![x]\!]_{\mathsf{E_1}} = (\boldsymbol{\sigma_x^1}, \mathbf{0}, \mathbf{0}) & \mathsf{V_1} : [\![x]\!]_{\mathsf{V_1}} = (\boldsymbol{\sigma_x^1}, \boldsymbol{\sigma_x^2}, \mathbf{0}) \\ \mathsf{E_2} : [\![x]\!]_{\mathsf{E_2}} = (\boldsymbol{\sigma_x^2}, \mathbf{0}, \mathbf{0}) & \mathsf{V_2} : [\![x]\!]_{\mathsf{V_2}} = (\boldsymbol{\sigma_x^1}, \boldsymbol{\sigma_x^2}, \mathbf{0}) \end{array}$$

5.2 Dot Product

Given vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, each of size d, the goal is to compute the dot product $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}} = \sum_{i=1}^{d} \mathsf{x}_i \mathsf{y}_i$. The recent works of ABY3 and ASTRA have tackled dot product computation in the semi-honest setting with cost equal to that of a single multiplication thus, making the total cost independent of the vector size. However, in the malicious setting, their techniques become expensive, with cost dependent on the vector size. In this work, we remove this dependency and retain the cost to be the same as that of a single multiplication. This independence stems from the peculiar structure of our sharing and our robust multiplication method. On a high level, instead of calling Π_{bic} protocol for $\mathsf{A}_{1i}, \mathsf{A}_{2i}, \mathsf{B}_{1i}$ and B_{2i} corresponding to each product $\mathsf{z}_i = \mathsf{x}_i \mathsf{y}_i$, the parties add up their shares and then invoke Π_{bic} once for each of the summed up share. To facilitate this modification, verifiers also adjust $\delta_{\mathsf{xy}}^2 = \sum_{i=1}^{d} \delta_{\mathsf{x}_i \mathsf{y}_i} - \delta_{\mathsf{xy}}^1$ before sending to E_2. Formal details are presented in Fig 7 below.

- **Input:** Parties input their $[\![\vec{\mathbf{x}}]\!]$ and $[\![\vec{\mathbf{y}}]\!]$ shares.
- **Output:** Parties obtain $[\![\mathsf{z}]\!]$ as output, where $\mathsf{z} = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$.

– Parties in $\mathbf{V}$ and E_1 collectively sample σ_{z}^1 and δ_{xy}^1, while parties in $\mathbf{V}$ and E_2 together sample σ_{z}^2.

– Verifiers $\mathsf{V}_1, \mathsf{V}_2$ compute $\delta_{\mathsf{xy}} = \Sigma_{i=1}^{d} \sigma_{\mathsf{x}_i} \sigma_{\mathsf{y}_i}$, set $\delta_{\mathsf{xy}}^2 = \delta_{\mathsf{xy}} - \delta_{\mathsf{xy}}^1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{V}_2, \delta_{\mathsf{xy}}^2, \mathsf{E}_2, \mathsf{E}_1)$, such that E_2 receives δ_{xy}^2.

– Parties in $\mathbf{V}$ and E_1 collectively sample Δ_1. Parties V_1 and E_1 compute $\mathsf{A}_1 = \Sigma_{i=1}^{d}(-\mu_{x_i}^1 \sigma_{y_i}^1 - \mu_{y_i}^1 \sigma_{x_i}^1) + \sigma_{\mathsf{z}}^1 + \delta_{\mathsf{xy}}^1 + \Delta_1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_1, \mathsf{A}_1, \mathsf{E}_2, \mathsf{V}_2)$, such that E_2 receives A_1.

– Similarly, parties in $\mathbf{V}$ and E_2 collectively sample Δ_2. Parties V_1 and E_2 compute $\mathsf{A}_2 = \Sigma_{i=1}^{d}(-\mu_{x_i}^1 \sigma_{y_i}^2 - \mu_{y_i}^1 \sigma_{x_i}^2) + \sigma_{\mathsf{z}}^2 + \delta_{\mathsf{xy}}^2 + \Delta_2$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_2, \mathsf{A}_2, \mathsf{E}_1, \mathsf{V}_2)$, such that E_1 receives A_2.

– V_2 and E_1 compute $\mathsf{B}_1 = \Sigma_{i=1}^{d}(-\mu_{x_i}^2 \sigma_{y_i}^1 - \mu_{y_i}^2 \sigma_{x_i}^1) - \Delta_1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_1, \mathsf{B}_1, \mathsf{E}_2, \mathsf{V}_1)$. Similarly, V_2 and E_2 compute $\mathsf{B}_2 = \Sigma_{i=1}^{d}(-\mu_{x_i}^2 \sigma_{y_i}^2 - \mu_{y_i}^2 \sigma_{x_i}^2) - \Delta_2$ and execute $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_2, \mathsf{B}_2, \mathsf{E}_1, \mathsf{V}_1)$.

– Evaluators compute $\mu_z = \mu_x \mu_y + \mathsf{A}_1 + \mathsf{A}_2 + \mathsf{B}_1 + \mathsf{B}_2$ locally. Parties in $\mathbf{E}$ and V_1 collectively sample μ_z^1 followed by evaluators setting $\mu_z^2 = \mu_z - \mu_z^1$ and execute $\Pi_{\mathsf{bic}}(\mathsf{E}_1, \mathsf{E}_2, \mu_z^2, \mathsf{V}_2, \mathsf{V}_1)$ for V_2 to receive μ_z^2.

Fig. 7. $\Pi_{\mathsf{dp}}([\![\vec{\mathbf{x}}]\!], [\![\vec{\mathbf{y}}]\!])$: Dot Product of two vectors

5.3 MSB Extraction

All machine learning models which perform the task of classification require comparison between two values as a building block during their process of training and prediction. Efficient comparison of two arithmetic values u and v in a private fashion has been an ongoing challenging problem. Concretely, given the arithmetic shares $[\![\mathsf{u}]\!]$ and $[\![\mathsf{v}]\!]$, the goal is to check if $\mathsf{u} < \mathsf{v}$. In fixed point arithmetic setting, we check $\mathsf{msb}(\mathsf{a}) = 1$, if $\mathsf{a} = \mathsf{v} - \mathsf{u} < 0$ and vice-versa. Thus the goal of the parties reduces to computing the $[\![\cdot]\!]^{\mathbf{B}}$ shares of $\mathsf{msb}(\mathsf{a})$ given the $[\![\cdot]\!]$ shares of a. SecureML made an effort in this direction with the use of a garbled circuit technique to compute $\mathsf{msb}(\mathsf{a})$ in the 2PC setting. Later, ABY3 and ASTRA proposed protocols to tackle MSB extraction for the the 3PC setting. ASTRA proposed a constant round protocol but it required a garble circuit version of Parallel Prefix Adder (PPA) to perform the MSB extraction leading to a high communication cost (dependent on the security parameter κ), whereas ABY3 proposed a protocol which uses the boolean variant of the PPA circuit which trades off the rounds (dependent on the circuit depth) for a more efficient communication cost (independent of the security parameter κ). As our goal is to get a communication efficient protocol we trade-off the rounds and use the boolean variant of PPA circuit proposed by ABY3. The proposed PPA circuit requires 2ℓ AND gates leading to a total communication cost of 24ℓ bits and has a multiplicative depth of $\log \ell$ rounds. Concretely, given the shares $[\![\mathsf{u}]\!]$ and $[\![\mathsf{v}]\!]$, parties first locally compute $[\![\mathsf{a}]\!] = [\![\mathsf{u}]\!] - [\![\mathsf{v}]\!]$, where $\mathsf{a} = (\mu_{\mathsf{a}}^1 + \mu_{\mathsf{a}}^2) - (\sigma_{\mathsf{a}}^1 + \sigma_{\mathsf{a}}^2)$. We observe that, the optimized PPA circuit is a two input circuit which takes two inputs in boolean format and outputs the MSB of the sum of the two inputs. Thus, given $[\![\mathsf{a}]\!] = \{\sigma_{\mathsf{a}}^1, \sigma_{\mathsf{a}}^2, \mu_{\mathsf{a}}^1, \mu_{\mathsf{a}}^2\}$, we first need to prepare the following inputs: i) $[\![\mu_{\mathsf{a}}^1 + \mu_{\mathsf{a}}^2]\!]^{\mathbf{B}}$ and ii) $[\![-\sigma_{\mathsf{a}}^1 - \sigma_{\mathsf{a}}^2]\!]^{\mathbf{B}}$ for the PPA circuit inorder to obtain $[\![\mathsf{msb}(\mathsf{a})]\!]^{\mathbf{B}}$ as the output. This is achieved by parties executing $\Pi_{\mathsf{cSh}}^{\mathbf{B}}(\mathbf{E}, \mu_{\mathsf{a}}^1 + \mu_{\mathsf{a}}^2)$ and $\Pi_{\mathsf{cSh}}^{\mathbf{B}}(\mathbf{V}, -\sigma_{\mathsf{a}}^1 - \sigma_{\mathsf{a}}^2)$. Parties then input their respective shares to the PPA circuit, execute Π_{mult} protocol for each AND gate in the circuit and finally obtain the $[\![\cdot]\!]^{\mathbf{B}}$ sharing of $\mathsf{msb}(\mathsf{a})$. We encourage the readers to refer to ABY3 [MR18] for more details.

5.4 Truncation

We use ℓ-bit integers in signed $2'$s complement form to represent a decimal value where the sign of the decimal value is represented by the most significant bit (MSB). Consider a decimal value z represented in the signed 2's complement form. We use d_{z} to denote the least significant bits that represent its fractional part and $i_{\mathsf{z}} = \ell - d_{\mathsf{z}}$ to represent its integral part. It is observed that in the face of repeated multiplications, d_{z} and i_{z} needed to represent the output z keeps doubling with every multiplication and can eventually lead to an overflow. To avoid this multiplication overflow while preserving the accuracy and correctness, truncation is performed at the output of a multiplication gate. Truncation of a value z is defined as $\mathsf{z}^{\mathsf{t}} = \mathsf{z}/2^{d_{\mathsf{z}}}$, where the value z is *right arithmetic shifted* by d_{z} bits.

SecureML [MZ17] proposed an efficient truncation method for the two-party setting, where the parties locally truncate the shares after a multiplication. They showed that this technique introduces at most 1 bit error in the least significant bit (LSB) position and thus causes a minor reduction in the accuracy. Later ABY3 [MR18] showed that this idea cannot be trivially extended to three party setting and proposed an alternative technique to achieve truncation. Their main idea revolves around generating $([\![\mathsf{r}]\!], [\![\mathsf{r}^{\mathsf{t}}]\!])$ pair, where r is a random ring element and $\mathsf{r}^{\mathsf{t}} = \mathsf{r}/2^d$. Parties then compute $\mathsf{z} - \mathsf{r}$ in clear and locally truncate it to obtain $(\mathsf{z}-\mathsf{r})^{\mathsf{t}}$. This is followed by generating $[\![(\mathsf{z}-\mathsf{r})^{\mathsf{t}}]\!]$ and adding it to $[\![\mathsf{r}^{\mathsf{t}}]\!]$ to obtain $[\![\mathsf{z}^{\mathsf{t}}]\!]$. Similar to SecureML, this technique may also incur a one-bit error in the LSB position of z^{t}. To generate $([\![\mathsf{r}]\!], [\![\mathsf{r}^{\mathsf{t}}]\!])$, ABY3 requires two expensive circuit evaluations and leading to a total cost of more than 100 ring elements per multiplication. While we adopt ABY3's idea of using $(\mathsf{r}, \mathsf{r}^{\mathsf{t}})$ pair in our Π_{mult} protocol to achieve truncation, we remove the need of expensive circuits and maintain the total cost to 14 ring elements.

We begin with the generation of $(\mathsf{r}, \mathsf{r}^{\mathsf{t}})$ pair. Parties in $\mathbf{V}$ and E_1 sample random $\mathsf{r}_1 \in \mathbb{Z}_{2^\ell}$, while parties in $\mathbf{V}$ and E_2 sample r_2. Verifiers V_1 and V_2 set $\mathsf{r} = \mathsf{r}_1 + \mathsf{r}_2$. Then parties V_1 and V_2 locally truncate r to obtain r^{t} and execute Π_{cSh} to generate $[\![\mathsf{r}^{\mathsf{t}}]\!]$. Thus, the pair $([\mathsf{r}], [\![\mathsf{r}^{\mathsf{t}}]\!])$ is generated. Unlike Π_{mult} (Fig 4), evaluators instead reconstruct $(\mathsf{z}-\mathsf{r})$, followed by locally truncating it to obtain $(\mathsf{z}-\mathsf{r})^{\mathsf{t}}$. Evaluators execute Π_{cSh} to generate $[\![(\mathsf{z}-\mathsf{r})^{\mathsf{t}}]\!]$ followed by locally adding to $[\![\mathsf{r}^{\mathsf{t}}]\!]$ to obtain $[\![\mathsf{z}^{\mathsf{t}}]\!]$. The formal details of our protocol Π_{mulTr} appears in Fig 8 below.

- **Input:** Parties input their $[\![\mathsf{x}]\!]$ and $[\![\mathsf{y}]\!]$ shares.
- **Output:** Parties obtain $[\![\mathsf{z}^{\mathsf{t}}]\!]$ as output, where $\mathsf{z}^{\mathsf{t}} = (\mathsf{xy})^{\mathsf{t}}$.

– Parties in $\mathbf{V}$ and E_1 collectively sample σ_{z}^1 and r_1, while parties in $\mathbf{V}$ and E_2 together sample σ_{z}^2 and r_2.

– Verifiers set $\mathsf{r} = \mathsf{r}_1 + \mathsf{r}_2$ and truncate r by d bits to obtain r^{t}. Parties execute $\Pi_{\mathsf{cSh}}(\mathbf{V}, \mathsf{r}^{\mathsf{t}})$ to generate $[\![\mathsf{r}^{\mathsf{t}}]\!]$ sharing.

– Verifiers locally set $\delta_{xy} = \sigma_x \cdot \sigma_y$ and compute $\delta_{xy}^2 = \delta_{xy} - \delta_{xy}^1$, where δ_{xy}^1 is collectively sampled by parties in $\mathbf{V}$ and E_1. Parties then execute $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{V}_2, \delta_{\mathsf{xy}}^2, \mathsf{E}_2, \mathsf{E}_1)$, such that E_2 receives δ_{xy}^2.

– Parties in $\mathbf{V}$ and E_1 collectively sample Δ_1. Parties V_1 and E_1 compute $\mathsf{A}_1 = -\mu_{\mathsf{x}}^1\sigma_{\mathsf{y}}^1 - \mu_{\mathsf{y}}^1\sigma_{\mathsf{x}}^1 + \delta_{\mathsf{xy}}^1 - \mathsf{r}_1 + \Delta_1$ and execute $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_1, \mathsf{A}_1, \mathsf{E}_2, \mathsf{V}_2)$, such that E_2 receives A_1.

– Similarly, parties in $\mathbf{V}$ and E_2 collectively sample Δ_2. Parties V_1 and E_2 compute $\mathsf{A}_2 = -\mu_{\mathsf{x}}^1\sigma_{\mathsf{y}}^2 - \mu_{\mathsf{y}}^1\sigma_{\mathsf{x}}^2 + \delta_{\mathsf{xy}}^2 - \mathsf{r}_2 + \Delta_2$ and execute $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_2, \mathsf{A}_2, \mathsf{E}_1, \mathsf{V}_2)$, such that E_1 receives A_2.

– Parties V_2 and E_1 compute $\mathsf{B}_1 = -\mu_\mathsf{x}^2\sigma_\mathsf{y}^1 - \mu_\mathsf{y}^2\sigma_\mathsf{x}^1 - \Delta_1$ and execute $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_1, \mathsf{B}_1, \mathsf{E}_2, \mathsf{V}_1)$. Similarly, V_2 and E_2 compute $\mathsf{B}_2 = -\mu_\mathsf{x}^2\sigma_\mathsf{y}^2 - \mu_\mathsf{y}^2\sigma_\mathsf{x}^2 - \Delta_2$ and execute $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_2, \mathsf{B}_2, \mathsf{E}_1, \mathsf{V}_1)$.

– Evaluators compute $\mathsf{z} - \mathsf{r} = \mu_x\mu_y + \mathsf{A}_1 + \mathsf{A}_2 + \mathsf{B}_1 + \mathsf{B}_2$ and truncate it by d bits to obtain $(\mathsf{z} - \mathsf{r})^\mathsf{t}$.

– Parties execute $\Pi_{\mathsf{cSh}}(\mathbf{E}, (\mathsf{z} - \mathsf{r})^\mathsf{t})$ to generate $[\![(\mathsf{z} - \mathsf{r})^\mathsf{t}]\!]$ sharing and locally add to obtain $[\![\mathsf{z}^\mathsf{t}]\!] = [\![(\mathsf{z} - \mathsf{r})^\mathsf{t}]\!] + [\![\mathsf{r}^\mathsf{t}]\!]$

Fig. 8. $\Pi^{\mathsf{A}}_{\mathsf{mulTr}}(\mathsf{x}, \mathsf{y})$: Truncation Protocol

5.5 Bit Conversion

Here, we describe a protocol to transform $[\![\cdot]\!]^{\mathbf{B}}$-sharing of bit b to its arithmetic equivalent. For this transformation, we use the following equivalence relation:

$$b = \sigma_b \oplus \mu_b = \mu_{b'} + \sigma_{b'} - 2\mu_{b'}\sigma_{b'}$$

where $\mu_{b'}$ and $\sigma_{b'}$ denote the bits μ_b and σ_b respectively over $\mathbb{Z}_{2^\ell}$. Parties who hold μ_b and σ_b in clear convert them to $\mu_{b'}$ and $\sigma_{b'}$ respectively. Parties generate $[\![\cdot]\!]$-sharing of $\sigma_{b'}$ and $\mu_{b'}$ by executing Π_{cSh} followed by multiplication of $[\![\mu_{b'}]\!]$ and $[\![\sigma_{b'}]\!]$. We call the resultant protocol as Π_{btr} and the formal details are given below.

- **Input:** Parties input their $[\![b]\!]^{\mathbf{B}}$ shares.
- **Output:** Parties obtain $[\![b]\!]$ as the output.

– Parties execute $\Pi_{\mathsf{cSh}}(\mathbf{V}, \sigma_{b'})$ and $\Pi_{\mathsf{cSh}}(\mathbf{E}, \mu_{b'})$ to generate $[\![\sigma_{b'}]\!]$ and $[\![\mu_{b'}]\!]$ respectively.

– Parties execute $\Pi_{\mathsf{mult}}([\![\mu_{b'}]\!], [\![\sigma_{b'}]\!])$ to generate $[\![\mu_{b'}\sigma_{b'}]\!]$, followed by locally computing $[\![b]\!] = [\![\mu_{b'}]\!] + [\![\sigma_{b'}]\!] - 2\,[\![\mu_{b'}\sigma_{b'}]\!]$.

Fig. 9. $\Pi_{\mathsf{btr}}([\![\mathsf{b}]\!]^{\mathbf{B}})$: Conversion of a bit to arithmetic equivalent

We observe that cost of multiplication in Π_{btr} can be reduced from 12ℓ to 10ℓ bits. Note that the value $\sigma_{\mu_{b'}}$ is set to zero, when Π_{cSh} is executed to generate $[\![\mu_{b'}]\!]$. This implies $\delta_{\mu_{b'}\sigma_{b'}} = 0$ and thus removes the extra call to Π_{bic} protocol.

5.6 Bit Insertion

Given a bit $b \in \{0, 1\}$ in $[\![\cdot]\!]^{\mathbf{B}}$-shared form and $x \in \mathbb{Z}_{2^\ell}$ in $[\![\cdot]\!]$-shared form, we have to compute $[\![bx]\!]$. A trivial solution is to convert $[\![b]\!]^{\mathbf{B}}$ to $[\![b]\!]$ using Π_{btr} followed by a multiplication with $[\![x]\!]$, which requires a total of 26 ring elements and 10 rounds. Instead, we propose a better solution that requires 18ℓ ring elements and 5 rounds in total. We can view the equation for bit insertion as follows:

$$\begin{aligned}
\mu_{bx} &= (\mu_b \oplus \sigma_b) \cdot (\mu_x - \sigma_x) + \sigma_{bx} \\
&= (\mu_{b'} + \sigma_{b'} - 2\mu_{b'}\sigma_{b'}) \cdot (\mu_x - \sigma_x) + \sigma_{bx} \\
&= \gamma_{b'x} - \mu_{b'}\sigma_x + (\mu_x - 2\gamma_{b'x})\sigma_{b'} + (2\mu_{b'} - 1)\delta_{b'x} + \sigma_{bx} \\
&= \gamma_{b'x} + (-\mu_{b'}^1\sigma_x + (\mu_x^1 - 2\gamma_{b'x}^1)\sigma_{b'} + (2\mu_{b'}^1 - 1)\delta_{b'x} + \sigma_{bx}) \\
&\quad + (-\mu_{b'}^2\sigma_x + (\mu_x^2 - 2\gamma_{b'x}^2)\sigma_{b'} + (2\mu_{b'}^2 - 1)\delta_{b'x}) \\
&= \gamma_{b'x} + (\mathsf{A}_1 + \mathsf{A}_2) + (\mathsf{B}_1 + \mathsf{B}_2)
\end{aligned}$$

where $\gamma_{b'x} = \mu_{b'}\mu_x$, $\delta_{b'x} = \sigma_{b'}\sigma_x$ and $\mu_{b'}$, $\sigma_{b'}$ represent μ_b and σ_b over $\mathbb{Z}_{2^\ell}$ respectively. In the above equation, we observe that, given the $[\cdot]$-shares of $\mu_{b'}$, $\sigma_{b'}$, $\gamma_{b'x}$ and $\delta_{b'x}$, parties can robustly compute $[\![\cdot]\!]$-sharing of μ_{bx}. The protocol proceeds as follows: Parties begin by generating $[\cdot]$-shares of $\mu_{b'}$, $\gamma_{b'x}$ towards set $\mathbf{V}$ and $\sigma_{b'}$, $\delta_{b'x}$ towards set $\mathbf{E}$, so that parties can compute A_1, A_2, B_1 and B_2. This is followed by parties executing Π_{bic} protocol for each A_i and B_i, so that E_1 and E_2 are able to compute μ_{bx}. The formal details appear in Fig 10.

- **Input:** Parties input their $[\![b]\!]^{\mathbf{B}}$ and $[\![x]\!]$ shares.
- **Output:** Parties obtain $[\![bx]\!]$ as the output.

– Parties in $\mathbf{V}$ and E_1 collectively sample random $\sigma_{bx}^1 \in \mathbb{Z}_{2^\ell}$, while parties in $\mathbf{V}$ and E_2 together sample random σ_{bx}^2.

– Parties in $\mathbf{V}$ and E_1 collectively sample random $\sigma_{b'}^1$ followed by V_1 and V_2 setting $\sigma_{b'}^2 = \sigma_{b'} - \sigma_{b'}^1$. Parties then execute $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{V}_2, \sigma_{b'}^2, \mathsf{E}_2, \mathsf{E}_1)$, such that E_2 receives $\sigma_{b'}^2$. The same procedure is used for E_2 to receive $\delta_{b'x}^2$.

– Parties in $\mathbf{E}$ and V_1 collectively sample random $\mu_{b'}^1$ followed by E_1 and E_2 setting $\mu_{b'}^2 = \mu_{b'} - \mu_{b'}^1$. Parties then execute $\Pi_{\mathsf{bic}}(\mathsf{E}_1, \mathsf{E}_2, \mu_{b'}^2, \mathsf{V}_2, \mathsf{V}_1)$, such that V_2 receives $\mu_{b'}^2$. The same procedure is used for V_2 to receive $\gamma_{b'x}^2$.

– Parties in $\mathbf{V}$ and E_1 collectively sample Δ_1. Parties V_1 and E_1 compute $\mathsf{A}_1 = -\mu_{b'}^1\sigma_x^1 + (\mu_x^1 - 2\gamma_{b'x}^1)\sigma_{b'}^1 + (2\mu_{b'}^1 - 1)\delta_{b'x}^1 + \sigma_{bx}^1 + \Delta_1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_1, \mathsf{A}_1, \mathsf{E}_2, \mathsf{V}_2)$.

– Similarly, parties in $\mathbf{V}$ and E_2 collectively sample Δ_2. Parties V_1 and E_2 compute $\mathsf{A}_2 = -\mu_{b'}^1\sigma_x^2 + (\mu_x^1 - 2\gamma_{b'x}^1)\sigma_{b'}^2 + (2\mu_{b'}^1 - 1)\delta_{b'x}^2 + \sigma_{bx}^2 + \Delta_2$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_1, \mathsf{E}_2, \mathsf{A}_2, \mathsf{E}_1, \mathsf{V}_2)$.

– Parties V_2 and E_1 compute $\mathsf{B}_1 = -\mu_{b'}^2\sigma_x^1 + (\mu_x^2 - 2\gamma_{b'x}^2)\sigma_{b'}^1 + (2\mu_{b'}^2 - 1)\delta_{b'x}^1 - \Delta_1$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_1, \mathsf{B}_1, \mathsf{E}_2, \mathsf{V}_1)$. Similarly, V_2 and E_2 compute $\mathsf{B}_2 = -\mu_{b'}^2\sigma_x^2 + (\mu_x^2 - 2\gamma_{b'x}^2)\sigma_{b'}^2 + (2\mu_{b'}^2 - 1)\delta_{b'x}^2 - \Delta_2$ and invoke $\Pi_{\mathsf{bic}}(\mathsf{V}_2, \mathsf{E}_2, \mathsf{B}_2, \mathsf{E}_1, \mathsf{V}_1)$.

– Evaluators compute $\mu_{b'x} = \mathsf{A}_1 + \mathsf{A}_2 + \mathsf{B}_1 + \mathsf{B}_2 + \gamma_{b'x}$ locally. Parties in $\mathbf{E}$ and V_1 collectively sample $\mu_{b'x}^1$ followed by evaluators setting $\mu_{b'x}^2 = \mu_{b'x} - \mu_{b'x}^1$ and invoking $\Pi_{\mathsf{bic}}(\mathsf{E}_1, \mathsf{E}_2, \mu_{b'x}^2, \mathsf{V}_2, \mathsf{V}_1)$.

Fig. 10. $\Pi_{\mathsf{bin}}([\![\mathsf{b}]\!]^{\mathbf{B}}, [\![x]\!])$: Insertion of bit b in a value

6 Secure Prediction

In this section, we provide detailed protocols for the prediction phase of the following ML algorithms – i) Linear Regression, ii) Logistic Regression, iii) Deep Neural Network and iv) Binarized Neural Network, using the building blocks constructed earlier in Section 5.

6.1 Our Model

We consider a server-aided setting where both model owner M and client C outsource their trained model parameters and query to a set of four non-colluding servers $\{\mathsf{V}_1, \mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2\}$, in a $[\![\cdot]\!]$-shared fashion. The servers then compute the function using our 4PC protocol and finally reconstruct the result towards C. We assume the existence of a malicious adversary $\mathcal{A}$, who can corrupt either M or C and at most one among $\{\mathsf{V}_1, \mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2\}$. Recall that $\mathbf{E}$ and $\mathbf{V}$ denote the set of servers $\{\mathsf{E}_1, \mathsf{E}_2\}$ and $\{\mathsf{V}_1, \mathsf{V}_2\}$ respectively. We begin with the assumption that both M and C have already outsourced their input vectors to $\{\mathsf{V}_1, \mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2\}$.

Notations: We use bold smalls to denote a vector. Given a vector $\vec{\mathbf{a}}$, the i^{th} element in the vector is denoted by a_i. Model Owner M holds a vector of *trained* model parameters denoted by $\vec{\mathbf{w}}$. C's query is denoted by $\vec{\mathbf{z}}$. Both $\vec{\mathbf{w}}$ and $\vec{\mathbf{z}}$ are vectors of size d, where d denotes the number of features.

6.2 Linear Regression

In case of linear regression model, the output of the prediction phase for a query $\vec{\mathbf{z}}$ is given by $\vec{\mathbf{w}} \odot \vec{\mathbf{z}} = \sum_{i=1}^{d} \mathsf{w}_i \mathsf{z}_i$. Thus the prediction phase boils down to servers executing Π_{dp} protocol with inputs as $[\![\vec{\mathbf{w}}]\!]$ and $[\![\vec{\mathbf{z}}]\!]$, to obtain $[\![\cdot]\!]$ shares of $\vec{\mathbf{w}} \odot \vec{\mathbf{z}}$.

6.3 Logistic Regression

The prediction phase of logistic regression model for a query $\vec{\mathbf{z}}$ is given by $\mathsf{sig}(\vec{\mathbf{w}} \odot \vec{\mathbf{z}})$, where $\mathsf{sig}(\cdot)$ denotes the sigmoid function. The sigmoid function is defined as $\mathsf{sig}(\mathsf{u}) = \frac{1}{1+e^{-\mathsf{u}}}$. SecureML [MZ17] showed the drawbacks of using sigmoid function for a general MPC setting and proposed a MPC friendly approximation, defined as follows :

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

The above equation can also be viewed as, $\mathsf{sigx}(u) = \overline{b_1} b_2(\mathsf{u} + 1/2) + \overline{b_2}$, where bit $b_1 = 1$ if $\mathsf{u} + 1/2 < 0$, bit $b_2 = 1$ if $\mathsf{u} - 1/2 < 0$. Servers execute $\Pi_{\mathsf{msb}}(\mathsf{u} + 1/2)$ and $\Pi_{\mathsf{msb}}(\mathsf{u} - 1/2)$ to generate $[\![b_1]\!]^{\mathbf{B}}$ and $[\![b_2]\!]^{\mathbf{B}}$ respectively. Servers can locally compute $[\![\overline{b_i}]\!]^{\mathbf{B}}$ from $[\![b_i]\!]^{\mathbf{B}}$. After this, $\Pi^{\mathsf{B}}_{\mathsf{mult}}([\![\overline{b_1}]\!], [\![b_2]\!])$ is executed to generate $[\![b]\!]^{\mathbf{B}}$, where $b = \overline{b_1} b_2$. Servers then invoke Π_{bin} on $[\![b]\!]^{\mathbf{B}}$ and $[\![(\mathsf{u} + 1/2)]\!]$ to generate $[\![\overline{b_1} b_2(\mathsf{u} + 1/2)]\!]$, and $\Pi_{\mathsf{btr}}([\![\overline{b_2}]\!]^{\mathbf{B}})$ to generate $[\![\overline{b_2}]\!]$. Servers then locally add their shares to obtain $[\![\mathsf{sigx}(u)]\!]$. Thus the cost for one query prediction in a logistic regression model is the same as the cost of linear regression, plus the additional overhead of computing $\mathsf{sigx}(\vec{\mathbf{w}} \circ \vec{\mathbf{z}})$.

6.4 Deep Neural Networks (DNN)

All the techniques used to tackle the above models can be easily extended to support neural network prediction. We follow a similar procedure as ABY3, where each node across all layers, use ReLU ($\mathsf{rel}(\cdot)$) as its activation function. It comprises of computation of activation vectors for all the layers of the network. The activation vector for a given layer i of the network is defined as $\vec{\mathbf{a}}_i = \mathsf{rel}(\vec{\mathbf{u}}_i)$, where $\vec{\mathbf{u}}_i = \mathbf{W}_i \times \vec{\mathbf{a}}_{i-1}$ is a matrix multiplication of weight matrix $\mathbf{W}_i$ with the activation vector of the previous layer. Weight matrix $\mathbf{W}_i \in \mathbb{R}^{n_i \times n_{i-1}}$ contains all the weights connecting the nodes between layers i and $i-1$, where n_i represents the number nodes in layer i. We set matrix $\vec{\mathbf{a}}_0 = \vec{\mathbf{z}}$, where $\vec{\mathbf{z}}$ is the input query of the client. All the above operations, that are needed for prediction, are simply a composition of several multiplications, dot products along with the evaluation of many ReLU functions. We now define the ReLU function below and also explain how to tackle it in our setting.

ReLU: The ReLU function is given as $\mathsf{max}(0, \mathsf{u})$. We view it as $\mathsf{rel}(\mathsf{u}) = \bar{b}\mathsf{u}$, where bit $b = 1$ if $\mathsf{u} < 0$, and $\bar{b}$ is the complement of b. Servers execute $\Pi_{\mathsf{msb}}(\mathsf{u})$ to generate $[\![b]\!]^{\mathbf{B}}$. Servers locally compute $[\![\bar{b}]\!]^{\mathbf{B}}$ from $[\![b]\!]^{\mathbf{B}}$, followed by executing Π_{bin} on $[\![\bar{b}]\!]^{\mathbf{B}}$ and $[\![\mathsf{u}]\!]$ to generate $[\![\bar{b}\mathsf{u}]\!]$.

6.5 Binarized Neural Network (BNN)

MOBIUS [KCY+18] proposed a secure prediction protocol for BNN in two party setting with one semi-honest corruption over $\mathbb{Z}_{2^\ell}$. In the original work of BNN [HCS+16], a batch normalization operation is performed at the output of every hidden layer of the binarized network, which requires bit-shifting mechanism. Performing bit-shifting in two party setting is very expensive. As a countermeasure, MOBIUS proposed an alternate solution for batch normalization with cost equal to that of one multiplication. The alternate solution is as follows: Suppose x_l^i be the output of node i in the l^{th} hidden layer, instead of using bit-shifting to normalize x_l^i, they perform $x_l'^i = p_l^i x_l^i + q_l^i$, where $x_l'^i$ is the normalized output and p_l^i, q_l^i are the normalization batch parameters for node i of hidden layer l, which are provided by M. MOBIUS also showed that this method drops the accuracy by a negligible amount. Inspired from the ideas of MOBIUS, we now provide a secure prediction protocol for our setting. Note that, $[\![\cdot]\!]$-shares of the weight matrices $\mathbf{W}_l \in \{-1,1\}^{n_l \times n_{l-1}}$, batch normalization parameters $\vec{\mathbf{p}}_l$, $\vec{\mathbf{q}}_l$, $\forall l \in \{1,\ldots,l_{\mathsf{final}}\}$ and the query $\vec{\mathbf{z}}$ are already available among the servers.

We describe our protocol layer by layer. We use n_l to denote the number of nodes in layer l. The computation in each layer l consists of three stages: i) The first stage comprises of matrix multiplication $\vec{\mathbf{x}}_l = \mathbf{W}_l \times f(\vec{\mathbf{x}}'_{l-1})$, where $\vec{\mathbf{x}}'_{l-1}$ denotes an n_{l-1}-sized vector and $f(\vec{\mathbf{x}}'_{l-1})$ denotes the vector obtained by applying activation function f on it. The activation function for a given value a is defined as

$$f(\mathsf{a}) = \begin{cases} -1 & \mathsf{a} < 0 \\ 1 & \mathsf{a} \geq 0 \end{cases}$$

The matrix multiplication can be viewed as n_l dot product (protocol Π_{dp}) computations. ii) Servers, then perform batch normalization process on vector $\vec{\mathbf{x}}_l$ to obtain $\vec{\mathbf{x}}'_l = \vec{\mathbf{p}}_l \circ \vec{\mathbf{x}}_l + \vec{\mathbf{q}}_l$, where $\circ$ denotes element wise multiplication. As evident, we use n_l multiplications and additions to compute the $[\![\cdot]\!]$-sharing of $\vec{\mathbf{x}}'_l$. iii) This stage consists of passing the $\vec{\mathbf{x}}'_l$ through the activation function f to obtain $f(\vec{\mathbf{x}}'_l)$. To compute the activation function $f(\mathsf{a})$ in a $[\![\cdot]\!]$-shared fashion, servers execute Π_{msb} on $[\![\mathsf{a}]\!]$ to extract the MSB $\mathsf{msb}(\mathsf{a})$, followed by executing Π_{btr} on $[\![\mathsf{msb}(\mathsf{a})]\!]^{\mathbf{B}}$ to generate $[\![\mathsf{msb}(\mathsf{a})]\!]$. Finally, the servers locally compute $[\![f(\mathsf{a})]\!] = 2[\![\mathsf{msb}(\mathsf{a})]\!] - 1$. For the input layer ($l = 0$), servers set $f(\vec{\mathbf{x}}'_0) = \vec{\mathbf{z}}$. Note that stage three is not required at the output layer.

7 Implementation & Benchmarks

We show the practicality of our framework by providing implementation results and compare with ABY3, in their respective settings over a ring of $\mathbb{Z}_{2^{64}}$.

i) Experimental Setup: Our experiments have been carried out both in the LAN and WAN setting. In the LAN setting, our machines are equipped with Intel Core i7-7790 CPU with 3.6 GHz processor speed and 32 GB RAM. Each of the four cores were able to handle eight threads, resulting in a total of 32 threads. We had a bandwidth of 1Gbps and an average round-trip time (rtt) of $\approx 0.26ms$. In the WAN setting, we use Microsoft Azure Cloud Services (Standard D8s v3, 2.4 GHz Intel Xeon® E5-2673 v3 (Haswell), 32GB RAM, 8 vcpus) with machines located in North Central US (S_1), South East Asia (S_2), Australia East (S_3) and West Europe (S_4). Each of the eight cores was capable of handling 8 threads resulting in a total of 64 threads. The bandwidth was limited to 20Mbps and the average rtt times are as follows:

S_1-S_2	S_1-S_3	S_1-S_4	S_2-S_3	S_2-S_4	S_3-S_4
$161.76ms$	$197.03ms$	$97.32ms$	$116.36ms$	$225.34ms$	$236.56ms$

We build on the ENCRYPTO library [CaTD17], following the standards of C++11. Due to the unavailability of the code of ABY3 [MR18], we implement their framework for comparison. For our executions, we report the average values over a run of 15 times.

ii) Parameters for Comparison: We consider three parameters for comparison– a) Latency (calculated as the maximum runtime of the servers), b) Communication complexity and c) Throughput (number of operations per unit time). The latency and throughput are evaluated over both LAN and WAN settings. The communication complexity is measured independent of the network. For the aforementioned algorithms, the throughput is calculated as the number of queries that can be computed per second and min in LAN and WAN respectively.

iii) Server Assignment: We assign the roles to the servers to maximize the performance of each of the frameworks, that we use for benchmarking. The table below provides the assignment of roles to the corresponding servers. For the 4PC setting, $\mathsf{V}_1, \mathsf{V}_2$ represent the set of verifiers while $\mathsf{E}_1, \mathsf{E}_2$ represent the set of evaluators. P_0, P_1, P_2 represent the parties, in the 3PC setting. we omit comparison with ASTRA framework as ABY3 outperforms ASTRA in terms of total communication (ref. Table 1).

Work	S_1	S_2	S_3	S_4
FLASH	E_1	E_2	V_1	V_2
ABY3	P_1	P_2	P_3	–

Table 3. Server Assignment for FLASH and ABY3 frameworks

iv) Datasets: We pick real-world datasets to measure the throughput for the prediction phase. The datasets we pick have features ranging from 13 to 784, which cover a range of feature sizes for a wide span of commonly used datasets.

ML Algorithm	Dataset	#features	#samples
Linear Reg.	Boston Housing Prices [HR78]	14	≈500
	Weather Conditions [NOA17]	31	≈119000
Logistic Reg.	Candy Power Ranking [Hic17]	13	≈85
	Food Recipes [Dar17]	680	≈20000
DNN & BNN	MNIST [LC10]	784	≈70000

Table 4. Real World datasets for Comparison

For Linear Regression, we use Boston Housing Prices Dataset (Boston) [HR78] and the dataset obtained from [NOA17] about the Weather Conditions in World War Two (Weather). The Boston dataset has ≈ 500 samples, each with 14 features, while the Weather dataset has $\approx 119,000$ samples with 31 features.

For Logistic Regression we use the dataset from [Dar17] which categorizes and gives the rating for recipes (Recipes) and Candy Power Ranking (Candy) dataset from [Hic17] which predicts the most popular Halloween candy. The Candy dataset is small with only 13 features and ≈ 85 samples whereas the Recipe dataset is large with 680 features and $\approx 20,000$ samples.

For Deep Neural Network and Binarized Neural Network, we use MNIST [LC10] which contains 784 pixel images of handwritten numbers, each of size 28×28. We also use synthetic datasets as it provides freedom to tune the number of features parameter and showcase the improvement with increasing feature size.

7.1 ML Building Blocks

We begin by comparing our protocols for some of the crucial ML building blocks, namely i) Dot Product, ii) MSB Extraction and iii) Truncation, against the state of the art protocols of ABY3 [MR18]. The comparison is mainly to show the substantial improvement we achieve in each building block when we shift from 3PC to 4PC setting, along with robustness guarantee. Later in Section 7.2 and 7.3 we show how the improvement in these blocks help us achieve massive improvements (Table.2) for our ML algorithms.

i) Dot Product: Dot Product is one of the vital building blocks for many machine learning algorithms like Linear Regression, Logistic Regression and Neural Network to name a few.

Work	LAN Latency (ms)	WAN Latency (s)
ABY3	3.55	1.10
FLASH	1.51	1.08

Table 5. Latency of 1 dot product computation for 784 features

Table 5 gives the comparison of our work with ABY3 with respect to the completion of one dot product computation for $d = 784$ features. We observe that for the LAN setting, even though the number of rounds required for completion of one dot product execution for both frameworks is 5 rounds, the latency of ABY3 is still twice of our FLASH. This discrepancy happens because the rtt of the network varies drastically with increase in the size of communication. In case of ABY3, due to their dot product protocol being dependent on the number of features the per party communication turns out to be 42.8KB, whereas our protocol incurs a tiny cost of 0.09KB. Such a discrepancy is not observed in WAN as the communication threshold to vary the rtt is very high, under which all our protocols operate. We also plot the number of dot product computations that can be performed per sec, for varying feature sizes.

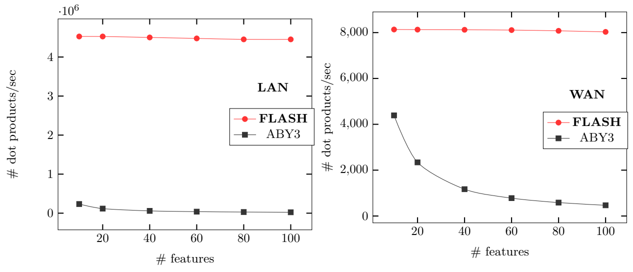

Fig. 11. # of dot product computations with increasing features.

It is clear from Figure.11 that varying the number of features has minimal impact on our throughput, since the communication cost of ours is independent of the feature size, while ABY3 suffers with increase in number of features. Thus for any machine learning algorithm which is heavily dependent on dot product computations, our protocol outperforms ABY3.

ii) MSB Extraction: MSB Extraction is the crux for many classification algorithms. Deep Neural Network and Binarized Neural Network where a large number of sequential comparisons are required. Table 6 gives the comparison of our work with ABY3, with respect to the completion of one MSB Extraction.

Work	LAN Latency (ms)	WAN Latency (s)
ABY3	3.53	2.22
FLASH	3.51	2.28

Table 6. Latency for single execution of MSB Extraction protocol

We also provide a latency graph with respect to the number of sequential comparisons.

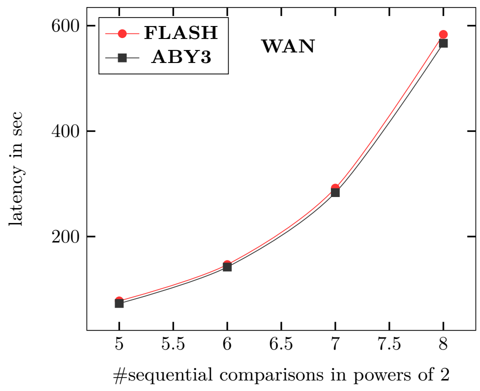

Fig. 12. Latency with increasing sequential comparisons

We observe from Figure 12 that the time taken for both protocols to complete a set of sequential comparisons are almost similar. Our completion time for sequential comparisons is slightly more than ABY3 in the WAN setting as the average rtts across the three servers (Table 3) in case of ABY3 is lesser as compared to ours where we require all four servers. Note that we omit the plot for the LAN setting as the average rtts between all servers are almost identical leading to both the plotted lines to practically overlap. Even though the average completion time for the both are almost identical, we require a communication cost of only $\approx$ 0.19KB per comparison as opposed to ABY3's cost of $\approx$ 0.33KB. Thus, for the prediction phase of an ML algorithm like Deep Neural Network, the gap in the communication cost will keep growing bigger with the increase in the number of hidden nodes in the neural network.

iii) Truncation: To showcase the effect of our efficient truncation protocol, we compare our protocol with that of ABY3. Table 7 gives the comparison with respect to the completion of a single execution of the protocol.

Work	LAN Latency (ms)	WAN Latency (s)
ABY3	1.52	1.11
FLASH	1.51	1.07

Table 7. Latency for a single execution of Truncation protocol

In the case of ABY3, though the truncation protocol takes $2\ell - 1$ rounds, the latency of both the frameworks in Table.7 are almost identical. This is because the goal of ABY3 was to have a high throughput framework, thus they compute $\approx 2^{20}$ parallel instances of $([\mathsf{r}], [\![\mathsf{r}^t]\!])$ pairs so that the amortized time for a single execution of truncation protocol reduces. If we consider only a single instance of $([\mathsf{r}], [\![\mathsf{r}^t]\!])$ pair, then the cost blows up to $\approx$ 760ms in the LAN and $\approx$ 26 sec in the WAN setting. On the flip side, we do not have any such restriction on the number of $([\mathsf{r}], [\![\mathsf{r}^t]\!])$ pair instances and the latency remains the same even if only one pair is required. Table 8 provides the throughput, measured as the number of multiplications with truncation performed, over both LAN (#mult/sec) and WAN (#mult/min) settings.

Work	LAN		WAN	
	#mult/sec	Improv.	#mult/min	Improv.
ABY3	0.45M	**8.8×**	4.76M	**8.81×**
FLASH	3.97M		0.54M	

Table 8. Throughput Comparison wrt # multiplications with truncation

We observe a minimum improvement of 8.8× over ABY3. The improvement comes from the fact that ABY3 requires $\approx$ 6300 bits per truncation as compared to 896 bits for our case, when instantiated over a 64 bit ring. Our protocol will outperform ABY3 for all the ML algorithms that require repeated multiplications in the prediction phase.

7.2 Linear and Logistic Regression

In this section, we compare the concrete improvement of our framework against ABY3, for Linear and Logistic Regression. The performance is reported in terms of throughput of the protocol, the units being # queries/sec over LAN and # queries/min over WAN. We begin by comparing our framework with ABY3 over synthesized datasets as it provides us the freedom to tune the number of features parameter and showcase the improvement with the increase in #features. Table 9 provides a throughput comparison for #features $d = 10, 100$ and 1000.

As mentioned earlier in Section.7.1, the increase in feature size changes the LAN latency for ABY3 from 1.68ms to 3.63ms and 5.59ms to 7.54ms for Linear and Logistic regression respectively, whereas our latency stays stable to $\approx$ 1.5ms and $\approx$ 5.36ms for the same. The reason for the stability in our latency is the underlying

Setting	# Features	Ref.	Linear Reg.	Logistic Reg.
LAN (ms)	10	ABY3	1.67	5.57
		FLASH	1.53	5.36
	100	ABY3	2.05	5.91
		FLASH	1.49	5.37
	1000	ABY3	3.61	7.55
		FLASH	1.54	5.39
WAN (sec)	10/100/1000	ABY3	1.12	3.77
		FLASH	1.09	3.73

Table 9. Latency of frameworks for Linear and Logistic Reg.

dot product which is independent of the feature size. We now test on real-world datasets as mentioned in Table 4 for Linear and Logistic Regression. Figures 13 and 14 provide a comparison with ABY3 in terms of the number of queries computed per second and minute in LAN and WAN setting respectively. For Linear Regression, we observe a minimum throughput gain of $\approx 35\times$. The improvement primarily comes from the underlying Π_{dp} protocol and its independence of feature size property. Similarly, for Logistic Regression, we observe a throughput gain of around $29\times$, where protocols Π_{dp} and Π_{msb} become the prime contributors for the improvements in Logistic Regression.

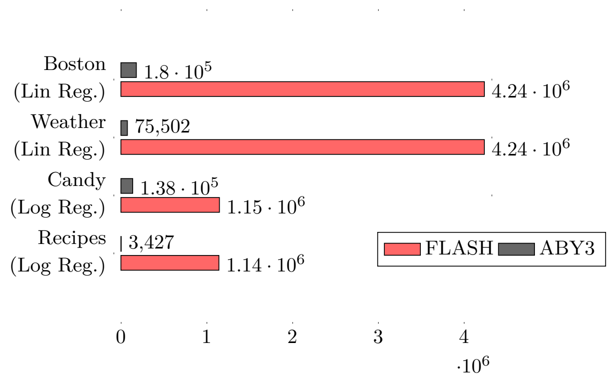

Fig. 13. Throughput Comparison (# queries/sec) for Linear and Logistic Regression in LAN setting

7.3 Deep and Binarized Neural Network

In this section, we compare our framework with ABY3, for DNN and BNN. The accuracy of our predictions has the same bit-error that ABY3 mentions due to the similarity in the approach to truncation. We begin by comparing (Table 10) over synthesized datasets and show the improvement in terms of latency for #features $d = 10, 100$ and 1000.

Figure 15 also shows how the depth of the neural network affects the throughput of the two frameworks.

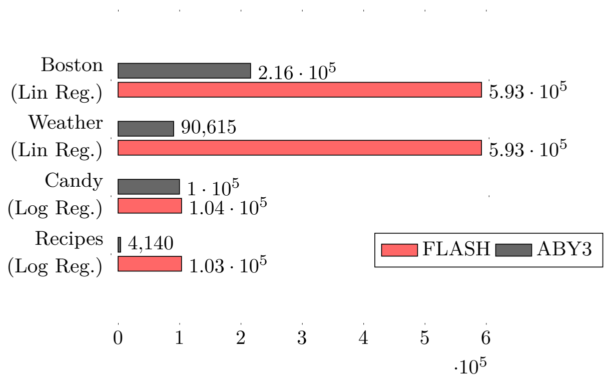

Fig. 14. Throughput Comparison (# queries/min) for Linear and Logistic Regression in WAN setting

Setting	# Features	Ref.	DNN	BNN
LAN (ms)	10	ABY3	58.98	59.18
		FLASH	28.78	31.46
	100	ABY3	67.79	67.83
		FLASH	28.86	31.71
	1000	ABY3	146.42	147.22
		FLASH	29.04	31.98
WAN (sec)	10/100/1000	ABY3	13.67	13.68
		FLASH	12.59	14.21

Table 10. Latency of frameworks for DNN and BNN

We consider a neural network with each hidden layer having 128 nodes and the final output layer having 10 nodes. The network is tested on MNIST dataset with $d = 784$ features.

It is clear from Figure 15, that we achieve impressive throughput gains of $\approx 155\times$ and $\approx 8.5\times$ for LAN and WAN setting respectively, even when the depth of the neural network goes up to 8 hidden layers. Such massive improvements primarily come from amalgamation of the improvements observed in the underlying building blocks (Section 7.1). Similar to DNN, we also achieve similar massive improvements for the case of BNN due to the aforementioned reasons. When tested on MNIST dataset ($d = 784$ features) for a BNN having 2 hidden layers, we observed throughput gains of $\approx 268\times$ in LAN and $\approx 11.5\times$ in WAN setting.

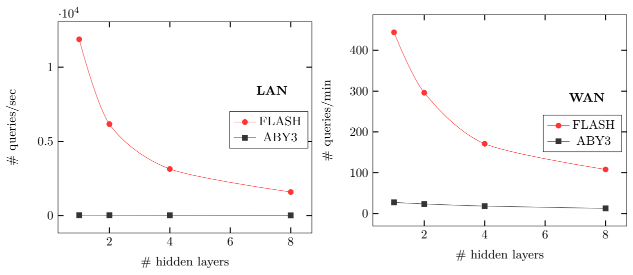

Fig. 15. Throughput Comparison for DNN with increasing number of hidden layers.

Acknowledgment: We thank Ananth Raghunathan, Yupeng Zhang and the anonymous reviewers of PETS 2020 for their valuable comments, which helped us improve the paper.

References

[ABF+16] T. Araki, A. Barak, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. DEMO: high-throughput secure three-party computation of kerberos ticket generation. In *ACM CCS*, 2016.

[ABF+17] 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.

[ADAM19] A.Barak, D.Escudero, A.P.K.Dalskov, and M.Keller. Secure evaluation of quantized neural networks. *IACR Cryptology ePrint Archive*, 2019.

[AFL+16] 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.

[AFS19] A.Tueno, F.Kerschbaum, and S.Katzenbeisser. Private evaluation of decision trees using sublinear cost. In *PoPETs*, 2019.

[ÁMJ+18] Á.Kiss, M.Naderpour, J.Liu, N. Asokan, and T.Schneider. Sok: Modular and efficient private decision tree evaluation. In *PoPETs*, 2018.

[BBC+19] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. How to prove a secret: Zero-knowledge proofs on distributed data via fully linear pcps. *CRYPTO*, 2019.

[BCD+09] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure Multiparty Computation Goes Live. In *FC*, 2009.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *ACM STOC*, 1988.

[BHPS19] M. Byali, C. Hazay, A. Patra, and S. Singla. Fast actively secure five-party computation with security beyond abort. In *ACM CCS*, 2019.

[BJPR18] M. Byali, A. Joseph, A. Patra, and D. Ravi. Fast secure computation for small population over the internet. *ACM CCS*, 2018.

[BLW08] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[BNP08] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM CCS*, 2008.

[CaTD17] Cryptography and Privacy Engineering Group at TU Darmstadt. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils, 2017.

[CCPS19] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High-throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW*, 2019.

[CGH+18] 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.

[CL14] R. Cohen and Y. Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. In *ASIACRYPT*, 2014.

[Cle86] R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *ACM STOC*, 1986.

[Dar17] H. Darwood. Epicurious - recipes with rating and nutrition. 2017.

[DKL+13] 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.

[DOS18] I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. *CRYPTO*, 2018.

[DPSZ12] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*, 2012.

[DSZ15] D. Demmler, T. Schneider, and M. Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*, 2015.

[EKN+17] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 2017.

[EOP+19] H. Eerikson, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin. Use your brain! arithmetic 3pc for any modulus with active security. *IACR Cryptology ePrint Archive*, 2019.

[FLNW17] 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.

[Gei07] M. Geisler. Viff: Virtual ideal functionality framework, 2007.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, 1987.

[GRW18] S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT*, 2018.

[HCS+16] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *NIPS*, 2016.

[Hic17] W. Hickey. The ultimate halloween candy power ranking. 2017.

[HR78] D. Harrison and D. L Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, 1978.

[IKKPC15] Y. Ishai, R. Kumaresan, E. Kushilevitz, and A. Paskin-Cherniavsky. Secure computation with minimal interaction, revisited. In *CRYPTO*, 2015.

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, 2003.

[JBAP19] J.So, B.Guler, A.S.Avestimehr, and P.Mohassel. Codedprivateml: A fast and privacy-preserving framework for distributed machine learning. *CoRR*, 2019.

[JVC18] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX*, 2018.

[KCY+18] H. Kitai, J. P. Cruz, N. Yanai, N. Nishida, T. Oba, Y. Unagami, T. Teruya, N. Attrapadung, T. Matsuda, and G. Hanaoka. MOBIUS: model-oblivious binarized neural networks. *CoRR*, 2018.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[Lin16] Y. Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *J. Cryptology*, 2016.

[LP07] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, 2007.

[MF06] P. Mohassel and M. K. Franklin. Efficiency tradeoffs for malicious two-party computation. In *PKC*, 2006.

[MMH+19] M.S.Riazi, M.Samragh, H.Chen, K.Laine, K.E.Lauter, and F.Koushanfar. XONN: xnor-based oblivious deep neural network inference. 2019.

[MR18] P. Mohassel and P. Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*, 2018.

[MRSV18] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren. EPIC: efficient private image classification (or: Learning from the masters). *CT-RSA*, 2018.

[MRZ15] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and Secure Three-party Computation: Garbled Circuit Approach. In *CCS*, 2015.

[MZ17] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, 2017.

[NO16] J. B. Nielsen and C. Orlandi. Cross and clean: Amortized garbled circuits with constant overhead. In *TCC*, 2016.

[NOA17] NOAA. Weather conditions in world war two. 2017.

[NV18] P. S. Nordholt and M. Veeningen. Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification. In *ACNS*, 2018.

[PR18] A. Patra and D. Ravi. On the exact round complexity of secure three-party computation. *CRYPTO*, 2018.

[RWT+18] 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.

[SKP15] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE CVPR*, 2015.

[WGC19] S. Wagh, D. Gupta, and N. Chandran. Securenn: Efficient and private neural network training. *19th Privacy Enhancing Technologies Symposium*, 2019.

[Yao82] A. C. Yao. Protocols for Secure Computations. In *FOCS*, 1982.

A Comparison with [GRW18]

In this section we compare our work with state-of-the-art 4PC protocol of [GRW18] in detail for the Abort scenario.

A.1 4PC with Abort:

All the aforementioned robust protocols can be easily converted to the abort variant by tweaking the Bi-convey primitive (Section 4.1). In case of abort setting, parties S_1 and S_2 in the Bi-Convey primitive send x and $\mathsf{H}(x)$ respectively to R, who accepts x if the hashes match else aborts. Thus by swapping with the abort variant of the primitive, all the building blocks achieve security with abort. Table 11 provides round and communication complexity comparison of both the variants of the protocols.

Protocol	Equation	FLASH (Abort)		FLASH (Robust)	
		Rounds	Comm.	Rounds	Comm.
Multiplication	$\llbracket x \rrbracket . \llbracket y \rrbracket \rightarrow \llbracket x.y \rrbracket$	2	6ℓ	5	12ℓ
Dot Product	$\llbracket \vec{\mathbf{x}} \odot \vec{\mathbf{y}} \rrbracket = \llbracket \sum_{i=1}^{d} x_i y_i \rrbracket$	2	6ℓ	5	12ℓ
MSB Extraction	$\llbracket x \rrbracket \rightarrow \llbracket \mathsf{msb}(x) \rrbracket^{\mathbf{B}}$	$\log \ell + 4$	14ℓ	$\log \ell + 5$	28ℓ
Truncation	$\llbracket x \rrbracket . \llbracket y \rrbracket \rightarrow \llbracket (xy)^t \rrbracket$	2	7ℓ	5	14ℓ
Bit Conversion	$\llbracket b \rrbracket^{\mathbf{B}} \rightarrow \llbracket b \rrbracket$	2	7ℓ	5	14ℓ
Bit Insertion	$\llbracket b \rrbracket^{\mathbf{B}} \llbracket x \rrbracket \rightarrow \llbracket bx \rrbracket$	2	9ℓ	5	18ℓ

Table 11. Comparison of Abort and Robust variants in FLASH.

As observed in Table 11, for the abort setting our cost of multiplication protocol is 6 elements which turns out to be the same as [GRW18]. But from a practical viewpoint, if we cast ours and GRW18 multiplication protocol into the offline-online paradigm, where the offline phase generates the necessary offline values in order for a fast online phase to be executed when the client query becomes available, our protocol requires only 3 parties to be active (V_2, E_1 and E_2) in the online phase, whereas [GRW18] needs all parties to be active throughout the entire execution. This is helpful, because now the server associated with party V_1 is only needed to generate offline values and can be shut down for the entirety of the online phase which will, in turn, save a lot in terms of monetary cost for running the server on the cloud (WAN) setting. Hence, even though the communication and round complexity of both the works turns out to be the same with respect to a single multiplication, our work has better *practical* efficiency in terms of the number of servers required in the online phase. Table 12 provides a concrete comparison of our framework with [GRW18] below.

Work	Equation	Offline Phase		Online Phase	
		Rounds	Comm.	Rounds	Comm.
[GRW18]	$[\![x]\!].[\![y]\!] \rightarrow [\![x.y]\!]$	1	2ℓ	1	4ℓ
Ours		1	3ℓ	1	3ℓ

Table 12. Comparison of FLASH with [GRW18] for the Abort setting.

B Building Blocks and Security

B.1 Building Blocks

i) Collision Resistant Hash: Consider a hash function family $\mathsf{H} = \mathcal{K} \times \mathcal{L} \rightarrow \mathcal{Y}$. The hash function H is said to be collision resistant if for all probabilistic polynomial-time adversaries $\mathcal{A}$, given the description of H_k where $k \in_R \mathcal{K}$, there exists a negligible function $\mathsf{negl}()$ 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 $x_1, x_2 \in_R \{0,1\}^m$.

ii) Commitment Scheme: We use $\mathsf{com}(x)$ to denote commitment of a value x. The commitment scheme ($\mathsf{com}()$) possess two properties, namely – i) *hiding*, which ensures the privacy of value x given just the commitment, and ii) *binding*, which prevents a corrupt party from opening the commitment to a different value $x' \neq x$. The commitment scheme can be implemented via a hash function $\mathcal{H}()$, whose security can be proved in the random-oracle model (ROM). For example, $(c, o) = (\mathcal{H}(x||r), x||r) = Com(x; r)$.

B.2 Ideal World Functionalities

We prove the security of our protocols in the standard real/ideal world paradigm where we compare the view of the adversary in the real world and ideal world. In an ideal world execution, each party sends its input to an incorruptible trusted third party (TTP), who computes the given function $f()$ using the inputs received and sends back the respective output to each party.

Fig 16 denotes the ideal functionality $\mathcal{F}_{\mathsf{setup}}$ that establishes the shared randomness among the parties.

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys $k_{\mathbf{E}}, k_{\mathbf{V}}, k_{\mathbf{E},\mathsf{V}_1}, k_{\mathbf{E},\mathsf{V}_2}, k_{\mathbf{V},\mathsf{E}_1}, k_{\mathbf{V},\mathsf{E}_2}, k_{\mathcal{P}} \in \{0,1\}^{\kappa}$. Let y_i denote the keys corresponding to party P_i. Then

– $\mathsf{y}_i = (k_{\mathbf{V}}, k_{\mathbf{E},\mathsf{V}_1}, k_{\mathbf{V},\mathsf{E}_1}, k_{\mathbf{V},\mathsf{E}_2}, k_{\mathcal{P}})$ when $P_i = \mathsf{V}_1$.
– $\mathsf{y}_i = (k_{\mathbf{V}}, k_{\mathbf{E},\mathsf{V}_2}, k_{\mathbf{V},\mathsf{E}_1}, k_{\mathbf{V},\mathsf{E}_2}, k_{\mathcal{P}})$ when $P_i = \mathsf{V}_2$.
– $\mathsf{y}_i = (k_{\mathbf{E}}, k_{\mathbf{V},\mathsf{E}_1}, k_{\mathbf{E},\mathsf{V}_1}, k_{\mathbf{E},\mathsf{V}_2}, k_{\mathcal{P}})$ when $P_i = \mathsf{E}_1$.
– $\mathsf{y}_i = (k_{\mathbf{E}}, k_{\mathbf{V},\mathsf{E}_2}, k_{\mathbf{E},\mathsf{V}_1}, k_{\mathbf{E},\mathsf{V}_2}, k_{\mathcal{P}})$ when $P_i = \mathsf{E}_2$.

Output: $\mathcal{F}_{\mathsf{setup}}$ sends the keys y_i to party P_i.

Fig. 16. Functionality $\mathcal{F}_{\mathsf{setup}}$

B.3 4PC Protocol

We present the 4PC protocol in Fig 17 and the corresponding ideal functionality appears in Fig 18.

Input Sharing: For each input value x, parties execute $\Pi_{\mathsf{sh}}(\mathsf{D}, x)$, where D is the owner of value x.
Addition gate: For every addition gate $\mathsf{z} = \mathsf{x} + \mathsf{y}$ in the ckt, parties execute $\Pi_{\mathsf{add}}(\mathsf{x}, \mathsf{y}, \mathsf{z})$.
Multiplication gate: For every multiplication gate $(\mathsf{z} = \mathsf{xy})$ in the ckt, parties execute $\Pi_{\mathsf{mult}}(\mathsf{x}, \mathsf{y}, \mathsf{z})$.
Output Computation: For every output value z, parties execute Π_{oc}.

Fig. 17. Π_{4PC}: A 4PC Robust Protocol

$\mathcal{F}_{\mathsf{robust}}$ receives input (Input, x) from party $P \in \{\mathsf{V}_1, \mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2\}$. While honest parties send their input correctly, corrupt parties may send arbitrary inputs as instructed by the adversary $\mathcal{A}$.

– For every party P, $\mathcal{F}_{\mathsf{robust}}$ sets x to some pre-determined value if either $x = *$ or x is outside the domain of values allowed for input of P.

–$\mathcal{F}_{\mathsf{robust}}$ computes output $y = f(x'_1, x'_2, x'_3, x'_4)$ and sends (Output, y) to all the parties in $\{\mathsf{V}_1, \mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2\}$.

Fig. 18. Functionality $\mathcal{F}_{\mathsf{robust}}$ for 4PC protocol

C Lemmas and Proofs

C.1 4PC

Lemma C.1. *The designated receiver R either receives a given value x correctly in Π_{bic} or receiver R and helper T mutually exchange all their internal randomness.*

Proof. The case of R and T (who act as pair of honest parties) mutually exchanging their internal randomness occurs when when one of the senders (S_1, S_2) are corrupt and copies of x received by R and the hashes $\mathsf{H}(x)$ received by T mismatch. In all the other cases there always exists a majority among the copies of x received by R. Thus R is able to correctly obtain x in the remaining cases. □

Lemma C.2. Π_{bic} *protocol requires a communication cost (amortized) of* 2ℓ *bits and at most* 2 *rounds.*

Proof. For a given value x, the communication cost is equal to 2ℓ bits as the senders S_1, S_2 send x to the designated party R. Round complexity wise, in case of a corrupt sender, he/she can delay party R from receiving x by atmost 2 rounds. This case occurs when in the first round the copies of x received by R mismatch and the hashes $\mathsf{H}(x)$ received by party T match. The second round simply involves party T sending $\mathsf{H}(x)$ to R who accepts the copy which matches with the received hash. The case when R or T is corrupt, Π_{bic} will take exactly 1 round as S_1 and S_2 will always send the correct copies. □

Lemma C.3. *For a gate* $\mathsf{g} = (\mathsf{x}, \mathsf{y}, \mathsf{z})$*, given the* $[\![\cdot]\!]$*-shares of inputs* x *and* y*, protocols* Π_{add} *and* Π_{mult} *compute* $[\![\cdot]\!]$*-share of the output wire* z*.*

Proof. By linearity property of $[\![\cdot]\!]$-sharing, the addition gates preserve the $[\![\cdot]\!]$-sharing of their inputs. For every multiplication gate $\mathsf{g} = (\mathsf{z} = \mathsf{xy})$, the evaluators robustly compute μ_z, after which they set $\mu_z^2 = \mu_z - \mu_z^1$(μ_z^1 chosen non-interactively) for consistent $[\![\cdot]\!]$-sharing of z to preserve the invariant. The share μ_z^2 for every multiplication gate is later robuslty communicated to the verifier V_2 to maintain a consistent $[\![\cdot]\!]$-sharing for the entire circuit. □

Lemma C.4. Π_{mult} *protocol requires a communication cost (amortized) of* 12ℓ *bits and at most* 5 *rounds.*

Proof. Π_{bic} of $\delta_{xy}^2, \mathsf{A}_1$, A_2, B_1 and B_2 takes 10ℓ bits followed by Π_{bic} of μ_z^2 takes another 2ℓ bits. Round complexity wise, in case of a corrupt verifier, Π_{bic} of δ_{xy}^2 takes atmost 2 rounds. Π_{bic} of A_1, A_2, B_1 and B_2 also takes atmost 2 rounds followed by evaluators executing Π_{bic} of μ_z^2 consumes 1 round. A similar argument can be made when one of the evaluator is corrupt. □

Lemma C.5. *Each party either commits to his/her input in* Π_{sh} *or is identified to be corrupt.*

Proof. In Π_{sh}, the mirrored sharing of inputs by each party is as in Π_{sh} with an additional step of identifying the adversary in case of mismatch. The step of eliminating the adversary uses the computation of honest majority on the dispersed shares. Since only, one corruption can occur, an honest party's input always gets committed irrespective of the behaviour of the adversary. However, the case of no honest majority can occur only when the dealer is corrupt. Hence only a corrupt party is eliminated if she does not commit to her input and a default value is taken. The uniqueness of the share also follows from collision resistant hash. Else, the chosen input is committed. □

Lemma C.6. *The protocol* Π_{oc} *is correct.*

Proof. The correctness for output computation follows from the fact that each party receives 2 copies and a corresponding hash for its missing share from the remaining parties. Thus each party correctly reconstructs the output as a majority always exists. □

Lemma C.7. *The protocol* Π_{4PC} *is correct.*

Proof. We argue that the computed z corresponds to unique set of inputs. By Lemma C.5, a corrupt party either commits to its input in which case, we proceed to evaluation or is identified to be corrupt and eliminated in which case, the output is computed on default input of the corrupt party. In the evaluation step, the computation of addition gates is local by the linearity property. For a multiplication gate $\Pi_{\mathsf{mult}}(\mathsf{x}, \mathsf{y}, \mathsf{z})$, the correctness of $\mathsf{A}_1, \mathsf{A}_2, \mathsf{B}_1, \mathsf{B}_2$ and δ_{xy}^2 sharing follows from the correctness of Π_{bic} protocol. Hence evaluators correctly compute μ_z, and set $\mu_z^2 = \mu_z - \mu_z^1$. Verifier V_2 also correctly receives μ_z^2, from the underlying correctness of Π_{bic} protocol. The protocol Π_{4PC}, relies on the the routines $\Pi_{\mathsf{sh}}, \Pi_{\mathsf{mult}}$ and Π_{oc} and thus its correctness follows from their correctness. □

C.2 Privacy-Preserving Machine Learning

C.2.1 Arithmetic/Boolean Couple Sharing

C.2.1.1 i) Parties in E couple share

Lemma C.8. Π_{cSh} *protocol requires a communication cost (amortized) of* 2ℓ *bits and atmost* 2 *rounds.*

Proof. The communication cost of 2ℓ bits comes directly from the cost of Π_{bic} protocol as the rest of the steps are local, which includes collectively sampling μ_x^1. Round complexity argument also follow from Π_{bic} protocol. □

C.2.1.2 ii) Parties in V couple share

Lemma C.9. Π_{cSh} *protocol requires a communication cost (amortized) of* 2ℓ *bits and atmost* 2 *rounds.*

Proof. The communication cost of 2ℓ bits comes directly from the cost of Π_{bic} protocol as the rest of the steps are local, which includes collectively sampling σ_x^1. Round complexity argument also follow from Π_{bic} protocol. □

C.2.2 4PC Truncation

Lemma C.10. Π_{mulTr} *protocol requires a communication cost (amortized) of* 14ℓ *bits and atmost* 5 *rounds.*

Proof. Π_{cSh} of $[\![\mathsf{r}^{\mathsf{t}}]\!]$ and δ_{xy} takes 4ℓ bits in total. Π_{bic} of A_1, A_2, B_1 and B_2 takes 8ℓ bits followed by Π_{cSh} of $(\mathsf{z}-\mathsf{r})^{\mathsf{t}}$ takes another 2ℓ bits. Round complexity wise, in case of a corrupt verifier, Π_{cSh} of $[\![\mathsf{r}^{\mathsf{t}}]\!]$ and δ_{xy} takes atmost 2 rounds. Π_{bic} of A_1, A_2, B_1 and B_2 also takes atmost 2 rounds followed by Π_{cSh} of $(\mathsf{z}-\mathsf{r})^{\mathsf{t}}$ consumes 1 round. A similar argument can be made when one of the evaluator is corrupt.

□

C.2.3 Dot Product

Lemma C.11. Π_{dp} *protocol requires a communication cost (amortized) of* 12ℓ *bits and atmost* 5 *rounds.*

Proof. The communication cost of 12ℓ bits comes directly from the cost of Π_{mult} protocol as the rest of the steps are local. Round complexity argument also follow from Π_{mult} protocol. □

C.2.4 Bit Conversion

Lemma C.12. Π_{btr} *protocol requires a communication cost (amortized) of* 14ℓ *bits and atmost* 5 *rounds.*

Proof. Firstly, the protocol Π_{cSh} used to generate the arithmetic equivalent $[\![\cdot]\!]$-sharing of bit σ_b and μ_b consumes 4ℓ bits in total. The optimized multiplication of $\mu_{b'}.\sigma_{b'}$ consumes 10ℓ bits in total as $\delta_{\mu_{b'}\sigma_{b'}}=0$ so Π_{cSh} is not required the same. In case of a corrupt verifier Π_{cSh} of σ_b can take atmost 2 rounds, followed by 3 rounds for optimized multiplication (as $\delta_{\mu_{b'}\sigma_{b'}}=0$) making the total rounds equal to 5. A similar argument can be made for the case when one of the evaluator is corrupt. □

C.2.5 Bit Insertion

Lemma C.13. Π_{bin} *protocol requires a communication cost (amortized) of* 18ℓ *bits and atmost* 5 *rounds.*

Proof. Four calls to Π_{bic} for $\sigma_{b'}^2, \mu_{b'}^2, \gamma_{b'x}$ and $\delta_{b'x}$ consumes 8ℓ bits in total. Again four calls to Π_{bic} each for A_1, A_2, B_1 and B_2 consumes another 8ℓ bits followed by evaluators invoking Π_{bic} of $\mu_{b'x}^2$ which consumes 2ℓ bits. Round complexity wise, in case of a corrupt verifier, Π_{bic} for $\sigma_{b'}^2, \mu_{b'}^2, \gamma_{b'x}$ and $\delta_{b'x}$ takes atmost 2 rounds, followed by Π_{bic} of A_1, A_2, B_1 and B_2 which consumes atmost 2 more rounds. Finally, Π_{bic} of $\mu_{b'x}^2$ which requires 1 round. A similar argument can be made when one of the evaluator is corrupt. □

C.2.6 MSB Extraction

Lemma C.14. Π_{msb} *protocol requires a communication cost (amortized) of* 28ℓ *bits and around log*$\ell + 5$ *rounds.*

Proof. To prepare the input for the optimized Parallel Prefix Adder (PPA) circuit takes 2 calls to Π_{cSh} protocol which takes 4ℓ bits and atmost 2 rounds. The remaining communication and round cost comes from computing the PPA circuit which requires computation of 2ℓ AND gates over a depth of $\log \ell + 3$ rounds. The communication cost of each AND gate is 12 bits (Lemma C.4), thus making the total cost of the circuit as $12 \times 2\ell = 24\ell$ bits. □

D Security Proofs

In this section, we provide detailed security proofs for all our aforementioned building blocks. We first discuss the general strategy of simulation for the entire circuit to tackle the corrupt party and then later provide the corresponding functionality and detailed simulation proof for each of the building block.

The simulator $\mathcal{S}$ for the entire circuit begins by simulating the $\mathcal{F}_{\mathsf{setup}}$ functionality and giving the keys to the adversary. This way the keys used in the PRF setup by the corrupt party during the course of circuit evaluation is also known to the simulator. During the input sharing phase the simulator on receiving the input shares from the corrupt party, on behalf of the honest parties, is able to extract the adversary's input using the keys given to him. This is possible because the inputs of each party are shared in mirrored sharing format. Additionally the simulator, on behalf of the honest parties set their inputs as 0. The simulator $\mathcal{S}$ now knows the inputs of all the parties and can compute all the intermediate values of each one of the building block in the circuit as well as the final output of the circuit in clear. Additionally the corrupt party receives only the input shares of the honest parties and hence cannot distinguish if the underlying value was 0 (received from the simulator) or the true values of the honest parties.

D.1 Security of Input Sharing

In this section, we first describe the ideal functionality followed by a detailed security proof for our Input Sharing Phase. The ideal functionality for the Input Sharing phase appears in Figure 19.

– $\mathcal{F}_{\mathsf{sh}}$ receives x from party/ dealer D who wants to generate $[\![\cdot]\!]$-sharing of x. Other parties input $\perp$ to the functionality.

– $\mathcal{F}_{\mathsf{sh}}$ randomly samples σ_x^1, σ_x^2 and $\mu_x^1 \in \mathbb{Z}_{2^\ell}$ and set $\mu_x^2 = x + \sigma_x^1 + \sigma_x^2 - \mu_x^1$.

– The output shares sent by $\mathcal{F}_{\mathsf{mul}}$ are as follows:

V_1: $(\sigma_x^1, \sigma_x^2, \mu_x^1)$, V_2: $(\sigma_x^1, \sigma_x^2, \mu_x^2)$

E_1: $(\sigma_x^1, \mu_x^1, \mu_x^2)$, E_2: $(\sigma_x^2, \mu_x^1, \mu_x^2)$

Fig. 19. Functionality $\mathcal{F}_{\mathsf{sh}}$: Ideal Functionality for Input Sharing of value x

We now describe the simulator for the case of a corrupt V_1 and a corrupt E_1. Other cases are similar to these and hence can be worked out in a similar way.

– If $\mathsf{D} = \mathsf{E}_1$, $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{V}_1}$ samples σ_x^1 on behalf of V_2 and E_1 and samples σ_x^2 on behalf of all honest parties respectively to compute $\sigma_x = \sigma_x^1 + \sigma_x^2$. Similar steps are done for $\mathsf{D} = \mathsf{E}_2$.
– If $\mathsf{D} = \mathsf{V}_1$, $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{V}_1}$ samples σ_x^1 and σ_x^2 on behalf of $\mathsf{V}_2, \mathsf{E}_1$ and $\mathsf{V}_2, \mathsf{E}_2$ respectively to compute $\sigma_x = \sigma_x^1 + \sigma_x^2$. Similar steps are done for $\mathsf{D} = \mathsf{V}_2$.
– If $\mathsf{D} = \mathsf{V}_1$, $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{V}_1}$ samples μ_x^1 on behalf of $\mathsf{E}_1, \mathsf{E}_2$. Receive μ_x^2 from V_1 on behalf of all honest parties. If the received copies have no majority, set $\mathsf{flag} = 1$.
 – If $\mathsf{flag} = 1$: $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{V}_1}$ sets the input of V_1 as $x = 0$ (default value) and executes a semihonest 3PC on behalf of the remaining three honest parties. $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{V}_1}$ then sends the final output to V_1.
 – Else If $\mathsf{flag} = 0$: $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{V}_1}$ extracts the input of V_1 by computing $x = \mu_x - \sigma_x$ and invokes $\mathcal{F}_{\mathsf{sh}}$ with input as x on behalf of V_1.
– If $\mathsf{D} = \mathsf{V}_2$, locally sample μ_x^1 on behalf of parties in $\mathbf{E}$ and V_2. Send $\mathsf{com}(\mu_x^2)$ to V_1 on behalf of $\mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2$ on a random μ_x^2. Similar steps are done for $\mathsf{D} = \mathsf{E}_i, i \in \{1, 2\}$.

Fig. 20. $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{V}_1}$: Simulator for corrupt V_1 in Π_{sh}

This completes the simulation for the case of a corrupt V_1. We now describe the simulator for the case of a corrupt E_1.

– If $\mathsf{D} = \mathsf{E}_1$, $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{E}_1}$ samples σ_x^1 on behalf of verifiers and samples σ_x^2 on behalf of verifiers and E_2 to compute $\sigma_x = \sigma_x^1 + \sigma_x^2$. Similar steps are done for $\mathsf{D} = \mathsf{E}_2$.
– If $\mathsf{D} = \mathsf{V}_i, i \in [2]$, sample σ_x^1 and σ_x^2 on behalf of verifiers and E_2 to compute $\sigma_x = \sigma_x^1 + \sigma_x^2$.
– If $\mathsf{D} = \mathsf{E}_1$, $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{E}_1}$ samples μ_x^1 on behalf of $\mathsf{V}_1, \mathsf{E}_2$ and receives μ_x^2 from E_1 on behalf of $\mathsf{V}_2, \mathsf{E}_2$ and $\mathsf{com}(\mu_x^2)$ on behalf of V_1. If there exists no majority, set $\mathsf{flag} = 1$.
 – If $\mathsf{flag} = 1$: $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{E}_1}$ sets the input of E_1 as $x = 0$ (default value) and executes a semihonest 3PC on behalf of the remaining three honest parties. $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{E}_1}$ then sends the final output to E_1.
 – Else if $\mathsf{flag} = 0$: $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{E}_1}$ extracts the input of E_1 by computing $x = \mu_x - \sigma_x$ and invokes $\mathcal{F}_{\mathsf{sh}}$ with input x on behalf of E_1.
– If $\mathsf{D} = \mathsf{E}_2$, locally sample μ_x^1 on behalf of parties in E_2 and V_2. Send μ_x^2 to E_1 on behalf of E_2 on a random μ_x^2. Also, send $\mathsf{com}(\mu_x^2)$ to E_1 on behalf of $\mathsf{V}_2, \mathsf{V}_1$. Similar steps are done for $\mathsf{D} = \mathsf{V}_i, i \in \{1, 2\}$.

Fig. 21. $\mathcal{S}_{\Pi_{\mathsf{sh}}}^{\mathsf{E}_1}$: Simulator for corrupt E_1 in Π_{sh}

D.2 Security of Bi-Convey

In this section, we provide a detailed security proof for our Bi-Convey Primitive (Π_{bic}), which forms the backbone for most of our constructions, in the stand-alone model. $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{P}$ denotes the simulator for the case of a corrupt party $P \in \{\mathsf{V}_1, \mathsf{V}_2, \mathsf{E}_1, \mathsf{E}_2\}$.

We begin with case of a corrupt S_1. Since party S_1 is not receiving any messages in the protocol Π_{bic}, there is no need for $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{S_1}$ to simulate any messages. Based on the messages received from S_1, simulator prepares the input value of corrupt S_1 and invoke the ideal functionality $\mathcal{F}_{\mathsf{bic}}$. A detailed description of $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{S_1}$ is given in Fig 22. Note that, $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{S_1}$ has the knowledge of input value x, since it plays the role of an honest S_2.

– $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{S_1}$ receives x' and $\mathsf{com}(x'')$ from S_1 on behalf of parties R and T respectively.
– If $x' \neq x$ or $\mathsf{com}(x'') \neq \mathsf{com}(x)$, $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{S_1}$ sets the input message of S_1 as $x_{S_1} = \perp$. Else it sets $x_{S_1} = x$.
– $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{S_1}$ invokes the ideal functionality $\mathcal{F}_{\mathsf{bic}}$ on behalf of S_1 with input x_{S_1}.

Fig. 22. $\mathcal{S}_{\Pi_{\mathsf{bic}}}^{S_1}$: Simulator for the case of corrupt S_1

It is easy to see that the view of the adversary $\mathcal{A}$ in the real and simulated worlds are indistinguishable. The case for a corrupt S_2 follows similarly.

We now consider the case of a corrupt R. For this, $\mathcal{S}^R_{\Pi_{\mathsf{bic}}}$ (Fig 23) samples a random value x on behalf of S_1, S_2 and prepares the commitment of x honestly. This is followed by sending the values x, x and $\mathsf{com}(x)$ to R on behalf of S_1, S_2 and T respectively.

- $\mathcal{S}^R_{\Pi_{\mathsf{bic}}}$ samples a random value x on behalf of S_1, S_2. It then prepares the commitment $\mathsf{com}(x)$ using a randomness shared with R.
- $\mathcal{S}^R_{\Pi_{\mathsf{bic}}}$ sends x, x and $\mathsf{com}(x)$ to R on behalf of S_1, S_2 and T respectively.
- $\mathcal{S}^R_{\Pi_{\mathsf{bic}}}$ invokes the simulator for ideal functionality $\mathcal{F}_{\mathsf{setup}}$ and obtains the internal randomness of R, I_R. $\mathcal{S}^R_{\Pi_{\mathsf{bic}}}$ invokes the ideal functionality $\mathcal{F}_{\mathsf{bic}}$ on behalf of R with I_R as the input.

Fig. 23. $\mathcal{S}^R_{\Pi_{\mathsf{bic}}}$: Simulator for the case of corrupt R

For the case of a corrupt T, $\mathcal{S}^T_{\Pi_{\mathsf{bic}}}$ (Fig 24) proceeds as follows: $\mathcal{S}^T_{\Pi_{\mathsf{bic}}}$ samples a random value x on behalf of S_1, S_2 and prepares the commitment of x honestly. This is followed by sending the values $\mathsf{com}(x), \mathsf{com}(x)$ and $\perp$ to T on behalf of S_1, S_2 and R respectively.

- $\mathcal{S}^T_{\Pi_{\mathsf{bic}}}$ samples a random value x on behalf of S_1, S_2. It then prepares the commitment $\mathsf{com}(x)$.
- $\mathcal{S}^T_{\Pi_{\mathsf{bic}}}$ sends $\mathsf{com}(x), \mathsf{com}(x)$ and `continue` to T on behalf of S_1, S_2 and R respectively.
- $\mathcal{S}^T_{\Pi_{\mathsf{bic}}}$ invokes the simulator for ideal functionality $\mathcal{F}_{\mathsf{setup}}$ and obtains the internal randomness of T, I_T. $\mathcal{S}^T_{\Pi_{\mathsf{bic}}}$ invokes the ideal functionality $\mathcal{F}_{\mathsf{bic}}$ on behalf of T with I_T as the input.

Fig. 24. $\mathcal{S}^T_{\Pi_{\mathsf{bic}}}$: Simulator for the case of corrupt T

In each of the cases, since the simulator behaves entirely as an honest party in the protocol simulation, the view of the adversary $\mathcal{A}$ in the real and simulated worlds are indistinguishable in a very straightforward manner. This concludes the proof.

D.3 Security of Multiplication

In this section, we describe the ideal functionality followed by a detailed security proof for our Multiplication phase and prove security in the standard model. The ideal functionality for the Multiplication phase appears in Figure 25.

Functionality $\mathcal{F}_{\mathsf{mul}}$ receives the inputs from the parties as follows:

- V_1: $[\![x]\!]_{\mathsf{V}_1}$, $[\![y]\!]_{\mathsf{V}_1}$ and internal randomness $\mathsf{I}_{\mathsf{V}_1}$.
- V_2: $[\![x]\!]_{\mathsf{V}_2}$, $[\![y]\!]_{\mathsf{V}_2}$ and internal randomness $\mathsf{I}_{\mathsf{V}_2}$.
- E_1: $[\![x]\!]_{\mathsf{E}_1}$, $[\![y]\!]_{\mathsf{E}_1}$ and internal randomness $\mathsf{I}_{\mathsf{E}_1}$.
- E_2: $[\![x]\!]_{\mathsf{E}_2}$, $[\![y]\!]_{\mathsf{E}_2}$ and internal randomness $\mathsf{I}_{\mathsf{E}_2}$.

On receiving the inputs $\mathcal{F}_{\mathsf{mul}}$ performs the following steps:

- $\mathcal{F}_{\mathsf{mul}}$ sets $\mathsf{flag} = 1$, if the copies σ^1_x received from $\mathsf{V}_1, \mathsf{V}_2$ and E_1 mismatch. $\mathcal{F}_{\mathsf{mul}}$ also performs similar checks for $\sigma^2_x, \mu^1_x, \mu^2_x$ and the shares of $[\![y]\!]$.
- If $\mathsf{flag} = 1$:
  - $\mathcal{F}_{\mathsf{mul}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
  - $\mathcal{F}_{\mathsf{mul}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
  - $\mathcal{F}_{\mathsf{mul}}$ sends the final output O to all the parties.
- Else If $\mathsf{flag} = 0$:
  - $\mathcal{F}_{\mathsf{mul}}$ computes $x = \mu^1_x + \mu^2_x - \sigma^1_x - \sigma^2_x$, $y = \mu^1_y + \mu^2_y - \sigma^1_y - \sigma^2_y$ and sets $z = xy$.
  - $\mathcal{F}_{\mathsf{mul}}$ randomly samples σ^1_z, σ^2_z and $\mu^1_z \in \mathbb{Z}_{2^\ell}$ and sets $\mu^2_z = z + \sigma^1_z + \sigma^2_z - \mu^1_z$.
  - The output shares sent by $\mathcal{F}_{\mathsf{mul}}$ are as follows:
    V_1: $(\sigma^1_z, \sigma^2_z, \mu^1_z)$, V_2: $(\sigma^1_z, \sigma^2_z, \mu^2_z)$
    E_1: $(\sigma^1_z, \mu^1_z, \mu^2_z)$, E_2: $(\sigma^2_z, \mu^1_z, \mu^2_z)$

Fig. 25. $\mathcal{F}_{\mathsf{mul}}$: Ideal Functionality for multiplication of two values x and y

We first begin by describing the simulator for the case of a corrupt V_1. Note that, $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ already has the knowledge of $\mathsf{I}_{\mathsf{V}_1}$, δ^2_{xy}, A_1 and A_2. Without loss of generality, we observe that only for the case of when V_1 acts as a sender in the Π_{bic} protocol, the output of Π_{bic} can lead to pair of honest parties exchanging their internal randomness with each other. Thus $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ emulates the $\mathcal{F}_{\mathsf{bic}}$ functionality on behalf of V_1 for each of δ^2_{xy}, A_1 and A_2. The simulator then checks if any of the output leads to exchange of internal randomness among two pair of honest parties, in which case $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_1} = (\perp, \perp, \perp)$ and $[\![\mathsf{y}]\!]_{\mathsf{V}_1} = (\perp, \perp, \perp)$ and invokes the $\mathcal{F}_{\mathsf{mul}}$ functionality on behalf of V_1. This will ensure that $\mathcal{F}_{\mathsf{mul}}$, on receiving the inputs, will find a mismatch in the copies of shares received and will directly compute the output of the entire circuit. A similar strategy is used in most of the other simulation proofs.

1) $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the sender, for δ^2_{xy}. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Similar steps are followed for the case of A_1 and A_2.
2) If $\mathsf{flag} = 0$:
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the helper T. The simulator also invokes $\mathcal{F}_{\mathsf{mul}}$ on behalf of V_1, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{V}_1}$, $[\![\mathsf{y}]\!]_{\mathsf{V}_1}$ and $\mathsf{I}_{\mathsf{V}_1}$.
3) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_1} = (\perp, \perp, \perp)$, $[\![\mathsf{y}]\!]_{\mathsf{V}_1} = (\perp, \perp, \perp)$ and invokes the ideal functionality $\mathcal{F}_{\mathsf{mul}}$ on behalf of V_1 .

– $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ sends the final circuit output O to V_1 on behalf of the pair of honest parties and discards any incoming message from V_1.

Fig. 26. $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$: Simulator for the case of corrupt V_1

This completes the simulation for the case of a corrupt V_1. We now describe the simulator for the case of a corrupt V_2. Simulator $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$ already has knowledge of δ^2_{xy}, B_1 and B_2.

1) $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the sender, for δ^2_{xy}. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Similar steps are followed for the case of B_1 and B_2.
2) If $\mathsf{flag} = 0$:
   – $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the receiver R. The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{mul}}$ on behalf of V_2, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{V}_2}$, $[\![\mathsf{y}]\!]_{\mathsf{V}_2}$ and $\mathsf{I}_{\mathsf{V}_2}$.
3) Else If $\mathsf{flag} = 1$:
   – $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_2} = (\perp,\perp,\perp)$, $[\![\mathsf{y}]\!]_{\mathsf{V}_2} = (\perp,\perp,\perp)$ and invokes the ideal functionality $\mathcal{F}_{\mathsf{mul}}$ on behalf of V_2 .
   – $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$ sends the final circuit output O to V_2 on behalf of the pair of honest parties and discards any incoming message from V_2.

Fig. 27. $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$: Simulator for the case of corrupt V_2

We describe the simulator for the case of a corrupt E_1. The case of a corrupt E_2 is similar to this case and hence can be worked out in a similar way. Note that, $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ has knowledge of A_1, B_1 and μ^2_{z}.

1) $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the sender, for each A_1 and B_1.If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ sets $\mathsf{flag} = 1$ and goes to step 3).
2) If $\mathsf{flag} = 0$:
   – $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1, for the case of μ^2_{z}, where $\mathsf{z} = \mathsf{xy}$.If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Else the simulator invokes $\mathcal{F}_{\mathsf{mul}}$, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{E}_1}$, $[\![\mathsf{y}]\!]_{\mathsf{E}_1}$ and $\mathsf{I}_{\mathsf{E}_1}$.
3) Else If $\mathsf{flag} = 1$:
   – $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{E}_1} = (\perp,\perp,\perp)$, $[\![\mathsf{y}]\!]_{\mathsf{E}_1} = (\perp,\perp,\perp)$ shares and invokes $\mathcal{F}_{\mathsf{mul}}$ on behalf of E_1.
   – $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ sends the final circuit output O to E_1 on behalf of the pair of honest parties and discards any incoming message from E_1.

Fig. 28. $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$: Simulator for the case of corrupt E_1

D.4 Security of Couple Sharing

In this section, we describe the ideal functionality followed by a detailed security proof for our Couple Sharing protocol and prove security in the standard model. The ideal functionality for the Couple Sharing phase appears in Figure 29.

Case 1: (S = E)

– $\mathcal{F}_{\mathsf{cSh}}$ receives x from parties E_1 and E_2 who wants to generate $[\![\cdot]\!]$-sharing of x. Other parties input $\perp$ to the functionality. Each party also send its internal randomness to $\mathcal{F}_{\mathsf{cSh}}$. Functionality $\mathcal{F}_{\mathsf{cSh}}$ sets $\mathsf{flag} = 1$, if the received copies of x mismatch.
– If $\mathsf{flag} = 1$:

– $\mathcal{F}_{\mathsf{cSh}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
– $\mathcal{F}_{\mathsf{cSh}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
– $\mathcal{F}_{\mathsf{cSh}}$ sends the final output O to all the parties.

– If $\mathsf{flag} = 0$:

– $\mathcal{F}_{\mathsf{cSh}}$ randomly samples $\mu^1_x \in \mathbb{Z}_{2^\ell}$ and sets $\sigma^1_x = 0, \sigma^2_x = 0$ and $\mu^2_x = x - \mu^1_x$.
– The output shares sent by $\mathcal{F}_{\mathsf{cSh}}$ are as follows:

V_1: $(0, 0, \mu^1_x)$, V_2: $(0, 0, \mu^2_x)$
E_1: $(0, \mu^1_x, \mu^2_x)$, E_2: $(0, \mu^1_x, \mu^2_x)$

Case 2: (S = V)

– $\mathcal{F}_{\mathsf{cSh}}$ receives x from parties V_1 and V_2 who wants to generate $[\![\cdot]\!]$-sharing of x. Other parties input $\perp$ to the functionality. Each party also send its internal randomness to $\mathcal{F}_{\mathsf{cSh}}$. Functionality $\mathcal{F}_{\mathsf{cSh}}$ sets $\mathsf{flag} = 1$, if the received copies of x mismatch.
– If $\mathsf{flag} = 1$:

– $\mathcal{F}_{\mathsf{cSh}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
– $\mathcal{F}_{\mathsf{cSh}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
– $\mathcal{F}_{\mathsf{cSh}}$ sends the final output O to all the parties.

– If $\mathsf{flag} = 0$:

– $\mathcal{F}_{\mathsf{cSh}}$ randomly samples $\sigma^1_x \in \mathbb{Z}_{2^\ell}$ and sets $\mu^1_x = 0,\ \mu^2_x = 0$ and $\sigma^2_x = x - \sigma^1_x$.
– The output shares sent by $\mathcal{F}_{\mathsf{cSh}}$ are as follows:

V_1: $(\sigma^1_x, \sigma^2_x, 0)$, V_2: $(\sigma^1_x, \sigma^2_x, 0)$
E_1: $(\sigma^1_x, 0, 0)$, E_2: $(\sigma^2_x, 0, 0)$

Fig. 29. Functionality $\mathcal{F}_{\mathsf{cSh}}$: Ideal Functionality for Couple Sharing of x

We first begin by describing the simulator for the case of a corrupt V_1 for both the cases of $\mathbf{S} = \mathbf{E}$ and $\mathbf{V}$. In case of $\mathbf{S} = \mathbf{E}$, simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{cSh}}}$ emulates the $\mathcal{F}_{\mathsf{bic}}$

functionality on behalf of V_1 as the helper for μ_x^2. In case of $\mathbf{S} = \mathbf{V}$, simulator $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$ emulates the $\mathcal{F}_{\mathsf{bic}}$ functionality on behalf of V_1 as the sender for σ_x^2. The simulator then checks if any of the output leads to exchange of internal randomness among two pair of honest parties, in which case $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$ sets $x = \perp$, and invokes the $\mathcal{F}_{\mathsf{cSh}}$ functionality on behalf of V_1. The case of a corrupt V_2 is similar to this case and hence can be worked out in a similar way.

Case 1: $(\mathbf{S} = \mathbf{E})$

1) $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the helper, for μ_x^2.
2) The simulator invokes the ideal functionality $\mathcal{F}_{\mathsf{cSh}}$ on behalf of V_1, with input as x.

Case 2: $(\mathbf{S} = \mathbf{V})$

1) $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the sender, for σ_x^2. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$ sets $\mathsf{flag} = 1$ and goes to step 3).
2) If $\mathsf{flag} = 0$:
   – The simulator invokes the ideal functionality $\mathcal{F}_{\mathsf{cSh}}$ on behalf of V_1, with input as x.
3) Else If $\mathsf{flag} = 1$:
   – The simulator invokes the ideal functionality $\mathcal{F}_{\mathsf{cSh}}$ on behalf of V_1, with input as $\perp$.
   – $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$ sends the final circuit output O to V_1 on behalf of the pair of honest parties and discards any incoming message from V_1.

Fig. 30. $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$: Simulator for the case of corrupt V_1

We describe the simulator for the case of a corrupt E_1. The simulation steps for a corrupt E_2 is similar to this case and hence can be worked out in a similar way.

Case 1: $(\mathbf{S} = \mathbf{E})$

1) $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{E}_1}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the sender, for μ_x^2. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{V}_1}$ sets $\mathsf{flag} = 1$ and goes to step 3).
2) If $\mathsf{flag} = 0$:
   – The simulator invokes the ideal functionality $\mathcal{F}_{\mathsf{cSh}}$ on behalf of E_1, with input as x.
3) Else If $\mathsf{flag} = 1$:
   – The simulator invokes the ideal functionality $\mathcal{F}_{\mathsf{cSh}}$ on behalf of E_1, with input as $\perp$.
   – $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{E}_1}$ sends the final circuit output O to E_1 on behalf of the pair of honest parties and discards any incoming message from E_1.

Case 2: $(\mathbf{S} = \mathbf{V})$

1) $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{E}_1}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the helper, for σ_x^2.
2) The simulator invokes the ideal functionality $\mathcal{F}_{\mathsf{cSh}}$ on behalf of E_1, with input as x.

Fig. 31. $\mathcal{S}_{\Pi_{\mathsf{cSh}}}^{\mathsf{E}_1}$: Simulator for the case of corrupt E_1

D.5 Security of Dot Product

In this section, we describe the ideal functionality followed by a detailed security proof for our Dot Product Protocol and prove security in the standard model. The ideal functionality for the Dot product protocol appears in Figure 32.

Functionality $\mathcal{F}_{\mathsf{dp}}$ receives the inputs from the parties as follows:

– V_1: $[\![\vec{\mathbf{x}}]\!]_{\mathsf{V}_1}$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{V}_1}$ and internal randomness $\mathsf{I}_{\mathsf{V}_1}$.
– V_2: $[\![\vec{\mathbf{x}}]\!]_{\mathsf{V}_2}$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{V}_2}$ and internal randomness $\mathsf{I}_{\mathsf{V}_2}$.
– E_1: $[\![\vec{\mathbf{x}}]\!]_{\mathsf{E}_1}$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{E}_1}$ and internal randomness $\mathsf{I}_{\mathsf{E}_1}$.
– E_2: $[\![\vec{\mathbf{x}}]\!]_{\mathsf{E}_2}$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{E}_2}$ and internal randomness $\mathsf{I}_{\mathsf{E}_2}$.

On receiving the inputs $\mathcal{F}_{\mathsf{dp}}$ performs the following steps:

– If for any $\sigma_{x_i}^1 \in \sigma_{\vec{\mathbf{x}}}^1$, the copies of $\sigma_{x_i}^1$ received from $\mathsf{V}_1, \mathsf{V}_2$ and E_1 mismatch, $\mathcal{F}_{\mathsf{dp}}$ sets $\mathsf{flag} = 1$. $\mathcal{F}_{\mathsf{dp}}$ also performs similar checks for $\sigma_{\vec{\mathbf{x}}}^2, \mu_{\vec{\mathbf{x}}}^1$, $\mu_{\vec{\mathbf{x}}}^2$ and the shares of $[\![\vec{\mathbf{y}}]\!]$.
– If $\mathsf{flag} = 1$:
  – $\mathcal{F}_{\mathsf{dp}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
  – $\mathcal{F}_{\mathsf{dp}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
  – $\mathcal{F}_{\mathsf{dp}}$ sends the final output O to all the parties.
– Else If $\mathsf{flag} = 0$:
  – $\mathcal{F}_{\mathsf{mul}}$ computes $\forall i, x_i = \mu_{x_i}^1 + \mu_{x_i}^2 - \sigma_{x_i}^1 - \sigma_{x_i}^2$, $y_i = \mu_{y_i}^1 + \mu_{y_i}^2 - \sigma_{y_i}^1 - \sigma_{y_i}^2$ and set $z = \Sigma_{i=1}^d x_i y_i$.
  – $\mathcal{F}_{\mathsf{mul}}$ randomly samples σ_z^1, σ_z^2 and $\mu_z^1 \in \mathbb{Z}_{2^\ell}$ and set $\mu_z^2 = z + \sigma_z^1 + \sigma_z^2 - \mu_z^1$.
  – The output shares sent by $\mathcal{F}_{\mathsf{mul}}$ are as follows:
    V_1: $(\sigma_z^1, \sigma_z^2, \mu_z^1)$, V_2: $(\sigma_z^1, \sigma_z^2, \mu_z^2)$
    E_1: $(\sigma_z^1, \mu_z^1, \mu_z^2)$, E_2: $(\sigma_z^2, \mu_z^1, \mu_z^2)$

Fig. 32. $\mathcal{F}_{\mathsf{dp}}$: Ideal Functionality for dot product of two values x and y

We first begin by describing the simulator for the case of a corrupt V_1. Thus $\mathcal{S}_{\Pi_{\mathsf{dp}}}^{\mathsf{V}_1}$ emulates the $\mathcal{F}_{\mathsf{bic}}$ functionality on behalf of V_1 for each of δ_{xy}^2, A_1 and A_2. The simulator then checks if any of the output leads to exchange of internal randomness among two pair of honest parties, in which case $\mathcal{S}_{\Pi_{\mathsf{dp}}}^{\mathsf{V}_1}$ sets $[\![\vec{\mathbf{x}}]\!]_{\mathsf{V}_1} = (\perp, \perp, \perp)$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{V}_1} = (\perp, \perp, \perp)$ and invokes the $\mathcal{F}_{\mathsf{dp}}$ functionality on behalf of V_1.

1) $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{dp}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the sender, for δ^2_{xy}. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{dp}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Similar steps are followed for the case of A_1 and A_2.

2) If $\mathsf{flag} = 0$:
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{dp}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the helper T. The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{dp}}$ on behalf of V_1, with inputs as $[\![\vec{\mathbf{x}}]\!]_{\mathsf{V}_1}$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{V}_1}$ and $\mathsf{I}_{\mathsf{V}_1}$.

3) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{dp}}}$ sets $[\![\vec{\mathbf{x}}]\!]_{\mathsf{V}_1} = (\perp,\perp,\perp)$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{V}_1} = (\perp,\perp,\perp)$ shares and invokes the ideal functionality $\mathcal{F}_{\mathsf{dp}}$ on behalf of V_1 .
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{dp}}}$ sends the final circuit output O to V_1 on behalf of the pair of honest parties and discards any incoming message from V_1.

Fig. 33. $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{dp}}}$: Simulator for the case of corrupt V_1

This completes the simulation for the case of a corrupt V_1. We now describe the simulator for the case of a corrupt V_2.

1) $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{dp}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the sender, for δ^2_{xy}. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{dp}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Similar steps are followed for the case of B_1 and B_2.

2) If $\mathsf{flag} = 0$:
   - $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{dp}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the receiver R. The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{dp}}$ on behalf of V_2, with inputs as $[\![\vec{\mathbf{x}}]\!]_{\mathsf{V}_2}$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{V}_2}$ and $\mathsf{I}_{\mathsf{V}_2}$.

3) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{dp}}}$ sets $[\![\vec{\mathbf{x}}]\!]_{\mathsf{V}_2} = (\perp,\perp,\perp)$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{V}_2} = (\perp,\perp,\perp)$ shares and invokes the ideal functionality $\mathcal{F}_{\mathsf{dp}}$ on behalf of V_2 .
   - $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{dp}}}$ sends the final circuit output O to V_2 on behalf of the pair of honest parties and discards any incoming message from V_2.

Fig. 34. $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{dp}}}$: Simulator for the case of corrupt V_2

We describe the simulator for the case of a corrupt a corrupt E_1. The case of a corrupt E_2 is similar to this case and hence can be worked out in a similar way.

1) $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{dp}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the sender, for each A_1 and B_1.If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{dp}}}$ sets $\mathsf{flag} = 1$ and goes to step 3).

2) If $\mathsf{flag} = 0$:
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{dp}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1, for the case of μ^2_{z}, where $\mathsf{z} = \Sigma^d_{i=1} x_i y_i$. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{dp}}}$ sets $\mathsf{flag}' = 1$ and goes to step 3). Else the simulator invokes $\mathcal{F}_{\mathsf{dp}}$, with inputs as $[\![\vec{\mathbf{x}}]\!]_{\mathsf{E}_1}$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{E}_1}$ and $\mathsf{I}_{\mathsf{E}_1}$.

3) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{dp}}}$ sets $[\![\vec{\mathbf{x}}]\!]_{\mathsf{E}_1} = (\perp,\perp,\perp)$, $[\![\vec{\mathbf{y}}]\!]_{\mathsf{E}_1} = (\perp,\perp,\perp)$ shares and invokes $\mathcal{F}_{\mathsf{dp}}$ on behalf of E_1.
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{dp}}}$ sends the final circuit output O to E_1 on behalf of the pair of honest parties and discards any incoming message from E_1.

Fig. 35. $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{dp}}}$: Simulator for the case of corrupt E_1

D.6 Security of Truncation

In this section, we describe the ideal functionality followed by a detailed security proof for our Truncation protocol and prove security in the standard model. The ideal functionality for the Truncation protocol appears in Figure 36.

Functionality $\mathcal{F}_{\mathsf{mulTr}}$ receives the inputs from the parties as follows:

- V_1: $[\![x]\!]_{\mathsf{V}_1}$, $[\![y]\!]_{\mathsf{V}_1}$ and internal randomness $\mathsf{I}_{\mathsf{V}_1}$.
- V_2: $[\![x]\!]_{\mathsf{V}_2}$, $[\![y]\!]_{\mathsf{V}_2}$ and internal randomness $\mathsf{I}_{\mathsf{V}_2}$.
- E_1: $[\![x]\!]_{\mathsf{E}_1}$, $[\![y]\!]_{\mathsf{E}_1}$ and internal randomness $\mathsf{I}_{\mathsf{E}_1}$.
- E_2: $[\![x]\!]_{\mathsf{E}_2}$, $[\![y]\!]_{\mathsf{E}_2}$ and internal randomness $\mathsf{I}_{\mathsf{E}_2}$.

On receiving the inputs $\mathcal{F}_{\mathsf{mulTr}}$ performs the following steps:

– $\mathcal{F}_{\mathsf{mulTr}}$ computes $\delta_{\mathsf{xy}} = \sigma_{\mathsf{x}}\sigma_{\mathsf{y}}$ using the shares of V_1. Similarly, $\mathcal{F}_{\mathsf{mulTr}}$ computes another copy δ'_{xy} using the shares of V_2. If $\delta_{\mathsf{xy}} \neq \delta'_{\mathsf{xy}}$, $\mathcal{F}_{\mathsf{mulTr}}$ sets $\mathsf{flag} = 1$ else $\mathcal{F}_{\mathsf{mulTr}}$ samples $\delta^1_{\mathsf{xy}} \in \mathbb{Z}_{2^\ell}$ and sets $\delta^2_{\mathsf{xy}} = \delta_{\mathsf{xy}} - \delta^1_{\mathsf{xy}}$.

– $\mathcal{F}_{\mathsf{mulTr}}$ computes $\mathsf{A}_1 = -\mu^1_{\mathsf{x}}\sigma^1_{\mathsf{y}} - \mu^1_{\mathsf{y}}\sigma^1_{\mathsf{x}} + \delta^1_{\mathsf{xy}} + \sigma^1_z + \Delta_1$ using the shares of V_1. Similarly, $\mathcal{F}_{\mathsf{mulTr}}$ computes another copy A'_1 using the shares of E_1. If $\mathsf{A}_1 \neq \mathsf{A}'_1$, $\mathcal{F}_{\mathsf{mulTr}}$ sets $\mathsf{flag} = 1$. Similar steps are performed for the case of A_2, B_1 and B_2.

– If $\mathsf{flag} = 1$:
  – $\mathcal{F}_{\mathsf{mulTr}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
  – $\mathcal{F}_{\mathsf{mulTr}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
  – $\mathcal{F}_{\mathsf{mulTr}}$ sends the final output O to all the parties.

– Else If $\mathsf{flag} = 0$:
  – $\mathcal{F}_{\mathsf{mulTr}}$ computes $x = \mu^1_x + \mu^2_x - \sigma^1_x - \sigma^2_x$, $y = \mu^1_y + \mu^2_y - \sigma^1_y - \sigma^2_y$ and set $z = (xy)^{\mathsf{t}}$, where value xy is truncated by d bits.
  – $\mathcal{F}_{\mathsf{mulTr}}$ randomly samples σ^1_z, σ^2_z and $\mu^1_z \in \mathbb{Z}_{2^\ell}$ and set $\mu^2_z = z + \sigma^1_z + \sigma^2_z - \mu^1_z$.
  – The output shares sent by $\mathcal{F}_{\mathsf{mulTr}}$ are as follows:

V_1: $(\sigma^1_z, \sigma^2_z, \mu^1_z)$, V_2: $(\sigma^1_z, \sigma^2_z, \mu^2_z)$
E_1: $(\sigma^1_z, \mu^1_z, \mu^2_z)$, E_2: $(\sigma^2_z, \mu^1_z, \mu^2_z)$

Fig. 36. $\mathcal{F}_{\mathsf{mulTr}}$: Ideal Functionality for truncation of two values x and y

We first begin by describing the simulator for the case of a corrupt V_1. Note that, $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ already has the knowledge of $\mathsf{I}_{\mathsf{V}_1}$, δ^2_{xy}, A_1, A_2 and r^{t}. Note that only for the case of when V_1 acts as a sender in the Π_{bic} protocol, the output of Π_{bic} can lead to pair of honest parties exchanging their internal randomness with each other. Thus $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ emulates the $\mathcal{F}_{\mathsf{bic}}$ functionality on behalf of V_1 for each of δ^2_{xy}, σ^2_{rt} A_1 and A_2. The simulator then checks if any of the output leads to exchange of internal randomness among two pair of honest parties, in which case $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_1} = (\bot,\bot,\bot)$, $[\![\mathsf{y}]\!]_{\mathsf{V}_1} = (\bot,\bot,\bot)$ shares and invoke the $\mathcal{F}_{\mathsf{mulTr}}$ functionality on behalf of V_1.

1) $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the sender, for σ^2_{rt}. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Similar steps are followed for the case of δ^2_{xy}, A_1 and A_2.
2) If $\mathsf{flag} = 0$:
   – $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the helper T. The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{mulTr}}$ on behalf of V_1, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{V}_1}$, $[\![\mathsf{y}]\!]_{\mathsf{V}_1}$ and $\mathsf{I}_{\mathsf{V}_1}$.
3) Else If $\mathsf{flag} = 1$:
   – $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_1} = (\bot,\bot,\bot)$, $[\![\mathsf{y}]\!]_{\mathsf{V}_1} = (\bot,\bot,\bot)$ shares and invokes $\mathcal{F}_{\mathsf{mulTr}}$ on behalf of V_1 .
   – $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$ sends the final circuit output O to V_1 on behalf of the pair of honest parties and discards any incoming message from V_1.

Fig. 37. $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mulTr}}}$: Simulator for the case of corrupt V_1

This completes the simulation for the case of a corrupt V_1. We now describe the simulator for the case of a corrupt V_2.

1) $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mulTr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the sender, for σ^2_{rt}. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Similar steps are followed for the case of δ^2_{xy}, B_1 and B_2.
2) If $\mathsf{flag} = 0$:
   – $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mulTr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the receiver R. The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{mul}}$ on behalf of V_2, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{V}_2}$, $[\![\mathsf{y}]\!]_{\mathsf{V}_2}$ and $\mathsf{I}_{\mathsf{V}_2}$.
3) Else If $\mathsf{flag} = 1$:
   – $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mulTr}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_2} = (\bot,\bot,\bot)$, $[\![\mathsf{y}]\!]_{\mathsf{V}_2} = (\bot,\bot,\bot)$ shares and invokes $\mathcal{F}_{\mathsf{mulTr}}$ on behalf of V_2 .
   – $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mulTr}}}$ sends the final circuit output O to V_2 on behalf of the pair of honest parties and discards any incoming message from V_2.

Fig. 38. $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mulTr}}}$: Simulator for the case of corrupt V_2

We describe the simulator for the case of a corrupt a corrupt E_1. The case of a corrupt E_2 is similar to this case and hence can be worked out in a similar way.

1) $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mulTr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the sender, for each A_1 and B_1.If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mulTr}}}$ sets $\mathsf{flag} = 1$ and goes to step 3).
2) If $\mathsf{flag} = 0$:
   – $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mulTr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1, for the case of μ^2_{z}, where $\mathsf{z} = \mathsf{xy}$.If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mulTr}}}$ sets $\mathsf{flag}' = 1$ and goes to step 3). Else the simulator invokes $\mathcal{F}_{\mathsf{mulTr}}$, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{E}_1}$, $[\![\mathsf{y}]\!]_{\mathsf{E}_1}$ and $\mathsf{I}_{\mathsf{E}_1}$.
3) Else If $\mathsf{flag} = 1$:
   – $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mulTr}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{E}_1} = (\bot,\bot,\bot)$, $[\![\mathsf{y}]\!]_{\mathsf{E}_1} = (\bot,\bot,\bot)$ shares and invokes $\mathcal{F}_{\mathsf{mulTr}}$ on behalf of E_1.
   – $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mulTr}}}$ sends the final circuit output O to E_1 on behalf of the pair of honest parties and discards any incoming message from E_1.

Fig. 39. $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mulTr}}}$: Simulator for the case of corrupt E_1

D.7 Security of MSB Extraction

In this section, we describe the ideal functionality followed by a security proof for our MSB Extraction protocol and prove security in the standard model. The ideal functionality for the MSB Extraction protocol appears in Figure 40.

Functionality $\mathcal{F}_{\mathsf{bin}}$ receives the inputs from the parties as follows:

– V_1: $[\![x]\!]_{\mathsf{V}_1}$ and internal randomness $\mathsf{I}_{\mathsf{V}_1}$.
– V_2: $[\![x]\!]_{\mathsf{V}_2}$ and internal randomness $\mathsf{I}_{\mathsf{V}_2}$.
– E_1: $[\![x]\!]_{\mathsf{E}_1}$ and internal randomness $\mathsf{I}_{\mathsf{E}_1}$.
– E_2: $[\![x]\!]_{\mathsf{E}_2}$ and internal randomness $\mathsf{I}_{\mathsf{E}_2}$.

On receiving the inputs $\mathcal{F}_{\mathsf{msb}}$ performs the following steps:

– $\mathcal{F}_{\mathsf{msb}}$ sets $\mathsf{flag} = 1$, if the copies σ^1_x received from $\mathsf{V}_1, \mathsf{V}_2$ and E_1 mismatch. $\mathcal{F}_{\mathsf{msb}}$ also performs similar checks for σ^2_x, μ^1_x and μ^2_x.
– If $\mathsf{flag} = 1$:
  – $\mathcal{F}_{\mathsf{msb}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
  – $\mathcal{F}_{\mathsf{msb}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
  – $\mathcal{F}_{\mathsf{msb}}$ sends the final output O to all the parties.
– Else If $\mathsf{flag} = 0$:
  – $\mathcal{F}_{\mathsf{msb}}$ computes $x = \mu^1_x + \mu^2_x - \sigma^1_x - \sigma^2_x$ and set $b = \mathsf{msb}(x)$.
  – $\mathcal{F}_{\mathsf{msb}}$ randomly samples σ^1_b, σ^2_b and $\mu^1_b \in \mathbb{Z}_{2^1}$ and set $\mu^2_b = b \oplus \sigma^1_b \oplus \sigma^2_b \oplus \mu^1_b$.
  – The output shares sent by $\mathcal{F}_{\mathsf{bin}}$ are as follows:
    V_1: $(\sigma^1_b, \sigma^2_b, \mu^1_b)$, V_2: $(\sigma^1_b, \sigma^2_b, \mu^2_b)$
    E_1: $(\sigma^1_b, \mu^1_b, \mu^2_b)$, E_2: $(\sigma^2_b, \mu^1_b, \mu^2_b)$

Fig. 40. $\mathcal{F}_{\mathsf{msb}}$: Ideal Functionality for extracting the MSB of value x

We give a description of the simulator for the case of a corrupt V_1. The case of a corrupt V_2, E_1 and E_2 is similar to this case and hence can be worked out in a similar way. Note that the PPA circuit primarily consists of AND gates and hence the simulator for the $\mathcal{F}_{\mathsf{msb}}$ is required to emulate the simulation steps corresponding to $\mathcal{F}_{\mathsf{cSh}}$ to prepare the inputs followed by $\mathcal{F}_{\mathsf{mul}}$ functionality, with respect to each AND gate in the circuit. For the case of corrupt V_1, simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{msb}}}$ first emulates $\mathcal{F}_{\mathsf{cSh}}$ functionality on behalf of V_1 to prepare the PPA circuit inputs followed by emulating $\mathcal{F}_{\mathsf{mul}}$ on behalf of V_1 for each AND gate in the PPA circuit with the appropriate inputs. Thus at any point if the adversary behaves maliciously the underlying either $\mathcal{F}_{\mathsf{cSh}}$ or $\mathcal{F}_{\mathsf{mul}}$ will take care of the misbehavior and give the final circuit output.

D.8 Security of Bit Conversion

In this section, we describe the ideal functionality followed by a detailed security proof for our Bit Conversion protocol and prove security in the standard model. The ideal functionality for the Bit Conversion protocol appears in Figure 41.

Functionality $\mathcal{F}_{\mathsf{bin}}$ receives the inputs from the parties as follows:

- V_1: $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_1}$ and internal randomness $\mathsf{I}_{\mathsf{V}_1}$.
- V_2: $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_2}$ and internal randomness $\mathsf{I}_{\mathsf{V}_2}$.
- E_1: $[\![b]\!]^{\mathbf{B}}_{\mathsf{E}_1}$ and internal randomness $\mathsf{I}_{\mathsf{E}_1}$.
- E_2: $[\![b]\!]^{\mathbf{B}}_{\mathsf{E}_2}$ and internal randomness $\mathsf{I}_{\mathsf{E}_2}$.

On receiving the inputs $\mathcal{F}_{\mathsf{btr}}$ performs the following steps:

– $\mathcal{F}_{\mathsf{btr}}$ sets $\mathsf{flag} = 1$, if the copies σ^1_b received from $\mathsf{V}_1, \mathsf{V}_2$ and E_1 mismatch. $\mathcal{F}_{\mathsf{btr}}$ also performs similar checks for σ^2_b, μ^1_b and μ^2_b.

– If $\mathsf{flag} = 1$:
  - $\mathcal{F}_{\mathsf{btr}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
  - $\mathcal{F}_{\mathsf{btr}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
  - $\mathcal{F}_{\mathsf{btr}}$ sends the final output O to all the parties.

– Else If $\mathsf{flag} = 0$:
  - $\mathcal{F}_{\mathsf{btr}}$ computes $b = \mu^1_b \oplus \mu^2_b \oplus \sigma^1_b \oplus \sigma^2_b$ and set $z = b$.
  - $\mathcal{F}_{\mathsf{bin}}$ randomly samples σ^1_z, σ^2_z and $\mu^1_z \in \mathbb{Z}_{2^\ell}$ and set $\mu^2_z = z + \sigma^1_z + \sigma^2_z - \mu^1_z$.
  - The output shares sent by $\mathcal{F}_{\mathsf{bin}}$ are as follows:
    V_1: $(\sigma^1_z, \sigma^2_z, \mu^1_z)$, V_2: $(\sigma^1_z, \sigma^2_z, \mu^2_z)$
    E_1: $(\sigma^1_z, \mu^1_z, \mu^2_z)$, E_2: $(\sigma^2_z, \mu^1_z, \mu^2_z)$

Fig. 41. $\mathcal{F}_{\mathsf{btr}}$: Ideal Functionality for conversion of bit b

We first begin by describing the simulator for the case of a corrupt V_1. The case of a corrupt V_2 is similar to this case and hence can be worked out in a similar way.

1) $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{btr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the helper for $\mu^2_{\mu_{b'}}$ and acting as the sender for $\sigma^2_{\sigma_{b'}}$. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{btr}}}$ sets $\mathsf{flag} = 1$ and goes to step 4).
2) Simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{msb}}}$ then simulates the steps of $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{mult}}}$ (Fig 26) on behalf of V_1 for the product $\mu_{b'}\sigma_{b'}$.
3) If $\mathsf{flag} = 0$:
   - The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{btr}}$ on behalf of V_1, with inputs as $[\![\mathsf{b}]\!]^{\mathbf{B}}_{\mathsf{V}_1}$ and $\mathsf{I}_{\mathsf{V}_1}$.
4) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{btr}}}$ sets $[\![\mathsf{b}]\!]^{\mathbf{B}}_{\mathsf{V}_1} = (\perp, \perp, \perp)$ and invokes $\mathcal{F}_{\mathsf{btr}}$ on behalf of V_1 .
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{btr}}}$ sends the final circuit output O to V_1 on behalf of the pair of honest parties and discards any incoming message from V_1.

Fig. 42. $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{btr}}}$: Simulator for the case of corrupt V_1

We now describe the simulator for the case of a corrupt a corrupt E_1. The case of a corrupt E_2 is similar to this case and hence can be worked out in a similar way.

1) $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{btr}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the helper for $\sigma^2_{\sigma_{b'}}$ and acting as the sender for $\mu^2_{\mu_{b'}}$. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{btr}}}$ sets $\mathsf{flag} = 1$ and goes to step 4).
2) $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{msb}}}$ then simulates the steps of $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$ (Fig 28) on behalf of E_1 for the product $\mu_{b'}\sigma_{b'}$.
3) If $\mathsf{flag} = 0$:
   - The simulator also invokes $\mathcal{F}_{\mathsf{btr}}$ on behalf of E_1, with inputs as $[\![\mathsf{b}]\!]^{\mathbf{B}}_{\mathsf{E}_1}$ and $\mathsf{I}_{\mathsf{E}_1}$.
4) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{btr}}}$ sets $[\![\mathsf{b}]\!]^{\mathbf{B}}_{\mathsf{E}_1} = (\perp, \perp, \perp)$ and invokes $\mathcal{F}_{\mathsf{btr}}$ on behalf of E_1 .
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{btr}}}$ sends the final circuit output O to E_1 on behalf of the pair of honest parties and discards any incoming message from E_1.

Fig. 43. $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{btr}}}$: Simulator for the case of corrupt E_1

D.9 Security of Bit Insertion

In this section, we describe the ideal functionality followed by a detailed security proof for our Bit Insertion protocol and prove security in the standard model. The ideal functionality for the Bit Insertion protocol appears in Figure 44.

Functionality $\mathcal{F}_{\mathsf{bin}}$ receives the inputs from the parties as follows:

- V_1: $[\![x]\!]_{\mathsf{V}_1}$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_1}$ and internal randomness $\mathsf{I}_{\mathsf{V}_1}$.
- V_2: $[\![x]\!]_{\mathsf{V}_2}$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_2}$ and internal randomness $\mathsf{I}_{\mathsf{V}_2}$.
- E_1: $[\![x]\!]_{\mathsf{E}_1}$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{E}_1}$ and internal randomness $\mathsf{I}_{\mathsf{E}_1}$.
- E_2: $[\![x]\!]_{\mathsf{E}_2}$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{E}_2}$ and internal randomness $\mathsf{I}_{\mathsf{E}_2}$.

On receiving the inputs $\mathcal{F}_{\mathsf{bin}}$ performs the following steps:

- $\mathcal{F}_{\mathsf{bin}}$ sets $\mathsf{flag} = 1$, if the copies σ^1_b received from $\mathsf{V}_1, \mathsf{V}_2$ and E_1 mismatch. $\mathcal{F}_{\mathsf{bin}}$ also performs similar checks for σ^2_b, μ^1_b and μ^2_b.
- A similar check is performed by $\mathcal{F}_{\mathsf{bin}}$ for the shares of $[\![x]\!]$.
- If $\mathsf{flag} = 1$:
  - $\mathcal{F}_{\mathsf{bin}}$ uses the internal randomness of the parties, computes all the inputs $i_1, \ldots, i_n$ of the circuit in clear.
  - $\mathcal{F}_{\mathsf{bin}}$ computes $O = f(i_1, \ldots, i_n)$ locally, where O denotes the output of the entire circuit evaluation.
  - $\mathcal{F}_{\mathsf{bin}}$ sends the final output O to all the parties.
- Else If $\mathsf{flag} = 0$:
  - $\mathcal{F}_{\mathsf{bin}}$ computes $x = \mu^1_x + \mu^2_x - \sigma^1_x - \sigma^2_x$, $b = \mu^1_b \oplus \mu^2_b \oplus \sigma^1_b \oplus \sigma^2_b$ and set $z = bx$, where $z = x$ if $b = 1$ else $z = 0$.
  - $\mathcal{F}_{\mathsf{bin}}$ randomly samples σ^1_z, σ^2_z and $\mu^1_z \in \mathbb{Z}_{2^\ell}$ and set $\mu^2_z = z + \sigma^1_z + \sigma^2_z - \mu^1_z$.
  - The output shares sent by $\mathcal{F}_{\mathsf{bin}}$ are as follows:
    V_1: $(\sigma^1_z, \sigma^2_z, \mu^1_z)$, V_2: $(\sigma^1_z, \sigma^2_z, \mu^2_z)$
    E_1: $(\sigma^1_z, \mu^1_z, \mu^2_z)$, E_2: $(\sigma^2_z, \mu^1_z, \mu^2_z)$

Fig. 44. $\mathcal{F}_{\mathsf{bin}}$: Ideal Functionality for bit insertion of bit b into value x

We first begin by describing the simulator for the case of a corrupt V_1. Note that, $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ already has the knowledge of $\mathsf{I}_{\mathsf{V}_1}$, $\sigma^2_{b'}$, δ^2_{xy}, A_1 and A_2. $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ emulates the $\mathcal{F}_{\mathsf{bic}}$ functionality on behalf of V_1 for each of $\sigma^2_{b'}$, δ^2_{xy}, A_1 and A_2. The simulator then checks if any of the output leads to exchange of internal randomness among two pair of honest parties, in which case $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ prepares incorrect $[\![\mathsf{x}]\!]_{\mathsf{V}_1}$ and $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_1}$ shares and invoke the $\mathcal{F}_{\mathsf{bin}}$ functionality on behalf of V_1.

1) $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the sender, for $\sigma^2_{b'}$. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ is set to 1, simulator $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ sets $\mathsf{flag} = 1$ and goes to step 4). Similar steps are followed for the case of δ^2_{xy}, A_1 and A_2.
2) $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ also emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the helper, for $\mu^2_{b'}$.
3) If $\mathsf{flag} = 0$:
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_1 acting as the helper T. The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{bin}}$ on behalf of V_1, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{V}_1}$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_1}$ and $\mathsf{I}_{\mathsf{V}_1}$.
4) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_1} = (\perp, \perp, \perp)$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_1} = (\perp, \perp, \perp)$ and invokes $\mathcal{F}_{\mathsf{bin}}$ on behalf of V_1 .
   - $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$ sends the final circuit output O to V_1 on behalf of the pair of honest parties and discards any incoming message from V_1.

Fig. 45. $\mathcal{S}^{\mathsf{V}_1}_{\Pi_{\mathsf{bin}}}$: Simulator for the case of corrupt V_1

This completes the simulation for the case of a corrupt V_1. We now describe the simulator for the case of a corrupt V_2.

1) $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{bin}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the sender, for $\sigma^2_{b'}$. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ is set to 1, simulator $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{bin}}}$ sets $\mathsf{flag} = 1$ and goes to step 4). Similar steps are followed for the case of δ^2_{xy}, B_1 and B_2.
2) $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{bin}}}$ additionally emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the receiver, for $\mu^2_{b'}$.
3) If $\mathsf{flag} = 0$:
   - $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{bin}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of V_2 acting as the receiver R. The simulator also invokes the ideal functionality $\mathcal{F}_{\mathsf{bin}}$ on behalf of V_2, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{V}_2}$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_2}$ and $\mathsf{I}_{\mathsf{V}_2}$.
4) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{bin}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{V}_2} = (\perp, \perp, \perp)$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{V}_2} = (\perp, \perp, \perp)$ and invokes $\mathcal{F}_{\mathsf{bin}}$ on behalf of V_2 .
   - $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{bin}}}$ sends the final circuit output O to V_2 on behalf of the pair of honest parties and discards any incoming message from V_2.

Fig. 46. $\mathcal{S}^{\mathsf{V}_2}_{\Pi_{\mathsf{mult}}}$: Simulator for the case of corrupt V_2

We describe the simulator for the case of a corrupt a corrupt E_1. The case of a corrupt E_2 is similar to this case and hence can be worked out in a similar way.

1) $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{bin}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the sender, for $\mu^2_{b'}$. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ is set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{bin}}}$ sets $\mathsf{flag} = 1$ and goes to step 3). Similar steps are followed for the case of A_1 and B_1. Additionally, $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{bin}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1 acting as the helper, for $\sigma^2_{b'}$.
2) If $\mathsf{flag} = 0$:
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{bin}}}$ emulates $\mathcal{F}_{\mathsf{bic}}$ on behalf of E_1, for the case of μ^2_{z}, where $\mathsf{z} = b'\mathsf{x}$. If the internal flag variable of $\mathcal{F}_{\mathsf{bic}}$ set to 1, simulator $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{bin}}}$ sets $\mathsf{flag}' = 1$ and goes to step 3). Else the simulator invokes $\mathcal{F}_{\mathsf{bin}}$, with inputs as $[\![\mathsf{x}]\!]_{\mathsf{E}_1}$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{E}_1}$ and $\mathsf{I}_{\mathsf{E}_1}$.
3) Else If $\mathsf{flag} = 1$:
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{bin}}}$ sets $[\![\mathsf{x}]\!]_{\mathsf{E}_1} = (\perp, \perp, \perp)$, $[\![b]\!]^{\mathbf{B}}_{\mathsf{E}_1} = (\perp, \perp, \perp)$ and invokes $\mathcal{F}_{\mathsf{bin}}$ on behalf of E_1.
   - $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{bin}}}$ sends the final circuit output O to E_1 on behalf of the pair of honest parties and discards any incoming message from E_1.

Fig. 47. $\mathcal{S}^{\mathsf{E}_1}_{\Pi_{\mathsf{mult}}}$: Simulator for the case of corrupt E_1