Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning

Sai Rahul Rachuri*, Ajith Suresh†
*International Institute of Information Technology Bangalore, India
Email: rahul.rachuri@icloud.com
†Indian Institute of Science, Bangalore India
Email: ajith@iisc.ac.in

***Abstract*—Machine learning has started to be deployed in fields such as healthcare and finance, which involves dealing with a lot of sensitive data. This propelled the need for and growth of privacy-preserving machine learning (PPML). We propose an actively secure four-party protocol (4PC), and a framework for PPML, showcasing its applications on four of the most widely-known machine learning algorithms – Linear Regression, Logistic Regression, Neural Networks, and Convolutional Neural Networks.**

Our 4PC protocol tolerating at most one malicious corruption is practically efficient as compared to Gordon et al. (ASIACRYPT 2018) as the 4th party in our protocol is not active in the online phase, except input sharing and output reconstruction stages. Concretely, we reduce the online communication as compared to them by 1 ring element. We use the protocol to build an efficient mixed-world framework (Trident) to switch between the Arithmetic, Boolean, and Garbled worlds. Our framework operates in the offline-online paradigm over rings and is instantiated in an outsourced setting for machine learning, where the data is secretly shared among the servers. Also, we propose conversions especially relevant to privacy-preserving machine learning. With the privilege of having an extra honest party, we outperform the current state-of-the-art ABY3 (for three parties), in terms of both rounds as well as communication complexity.

The highlights of our framework include using a minimal number of expensive circuits overall as compared to ABY3. This can be seen in our technique for truncation, which does not affect the online cost of multiplication and removes the need for any circuits in the offline phase. Our B2A conversion has an improvement of $7\times$ in rounds and $18\times$ in the communication complexity. In addition to these, all of the special conversions for machine learning, e.g. Secure Comparison, achieve constant round complexity.

The practicality of our framework is argued through improvements in the benchmarking of the aforementioned algorithms when compared with ABY3. All the protocols are implemented over a 64-bit ring in both LAN and WAN settings. Our improvements go up to $187\times$ for the training phase and $158\times$ for the prediction phase when observed over LAN and WAN.

I. Introduction

Machine learning is one of the fastest-growing research domains today. Applications for machine learning range from smarter keyboard predictions to better object detection in self-driving cars to avoid collisions. This is in part due to more data being made available with the rise of internet companies such as Google and Amazon, as well as due to the machine learning algorithms themselves getting more robust and accurate. In fact, machine learning algorithms have now started to beat humans at some complicated tasks such as classifying echocardiograms [1], and they are only getting better. Techniques such as deep learning and reinforcement learning are at the forefront making such breakthroughs possible.

The level of accuracy and robustness required is very high to operate in mission-critical fields such as healthcare, where the functioning of the model is vital to the working of the system. Accuracy and robustness are governed by two factors, one of them is the high amount of computing power demanded to train deep learning models. The other factor influencing the accuracy of the model is the variance in the dataset. Variance in datasets comes from collecting data from multiple diverse sources, which is typically infeasible for a single company to achieve.

Towards this, companies such as Microsoft (Azure), Amazon (AWS), Google (Google Cloud), etc. have entered into space by offering "Machine Learning as a Service (MLaaS)". MLaaS works in two different ways, depending on the end-user. The first scenario is companies offering their trained machine learning models that a customer can query to obtain the prediction result. The second scenario is when multiple customers/companies want to come together and train a common model using their datasets, but none of them wish to share the data in the clear. While promising, both models require the end-user to make compromises. In the case of an individual customer, privacy of his/her query is not maintained and in the case of companies, policies like the European Union General Data Protection Regulation (GDPR) or the EFF's call for information fiduciary rules for businesses have made it hard and often illegal for companies to share datasets with each other without prior consent of the customers, security, and other criteria met. Even with all these criteria met, data is proprietary information of a company which they would not want to share due to concerns such as competitive advantage.

Due to the huge interest in using machine learning, the field of privacy-preserving machine learning (PPML) has become a fast-growing area of research that addresses the aforementioned problems through techniques for privacy-preserving training and prediction. These techniques when deployed ensure that no information about the query or the datasets is leaked beyond what is permissible by the algorithm, which in some cases might be only the prediction output. Recently there have been a slew of works that have used the techniques of Secure Multiparty Computation (MPC) to perform efficient PPML, works such as [2]–[6] making huge contributions.

Secure multiparty computation is an area of extensive research that allows for n mutually distrusting parties to perform computations together on their private inputs, such that no coalition of t parties, controlled by an adversary, can learn any information beyond what is already known and permissible by the algorithm. While MPC has been shown to be practical [7]–[9], MPC for a small number of parties in the *honest majority* setting [10]–[15], [15]–[19] has become popular over the last few years due to applications such as financial data analysis [20], email spam filtering [21], distributed credential encryption [16], privacy-preserving statistical studies [22] that involve only a few parties. This is also evident from popular MPC frameworks such as Sharemind [23] and VIFF [24].

Our Setting: In this work we deal with the specific case of MPC with 4 parties (4PC), tolerating at most 1 malicious corruption. The state-of-the-art three-party (3PC) PPML frameworks in the honest majority setting such as ABY3 [5], SecureNN [6], and ASTRA [25] (prediction only) have fast and efficient protocols for the semi-honest case but are significantly slower when it comes to the malicious setting. This is primarily due to the underlying operations such as Dot Product, Secure Comparison, and Truncation being more expensive in the malicious setting. For instance, the Dot Product protocol of ABY3 incurs communication cost that is linearly dependent on the size of the underlying vector. Since these operations are performed many times, especially during the training phase, more efficient protocols for these operations are crucial in building a better PPML framework.

The motivation behind our 4PC setting is to investigate the performance improvement, both theoretical and practical, over the existing solutions in the 3PC setting, when given the privilege of an additional honest party. We show later in this work that having an extra honest party helps us achieve simpler and much more efficient protocols as compared to 3PC. For instance, operating in 4PC eliminates the need for expensive multiplication triples and allows us to perform a dot product at a cost that is independent of the size of the two vectors.

Our ML constructions are built on a new 4PC scheme, instead of the one proposed by Gordon et al. [26], primarily due to the following reasons:

1) Our protocol requires only three out of the four parties to be active during most of the online phase. On the contrary, [26] demands all the four parties to be active during the online phase. Thus, our protocol is more efficient in the setting where the computation is outsourced to a set of servers.

2) Using the new secret sharing scheme, our protocol shifts 25% (1 ring element) of the online communication to the offline phase, thus improving the online efficiency.

While our setting is more communication efficient, we assume the presence of an extra honest party which demands an additional 3 pairwise authentic channels when compared to that of 3PC. However, *monetary cost* [27] is an important parameter to look at since the servers need to be running for a long time for complex ML models. The time servers run for and the compute power of the servers dictate the cost of operation. As the fourth party in our framework does not have to be online throughout the online phase, we can shut the server down for most of the online phase. Aided by this fact, the total monetary cost, which would be the total cost of hiring 4 servers to run our framework for either the training or prediction phase of an algorithm, we come out ahead of ABY3, primarily because the total running time of the servers in our framework is much lower. More details about monetary cost are presented in Appendix E.

Offline-online paradigm: To improve efficiency, a class of MPC protocols operates in the *offline-online* paradigm [28]. Data-independent computations are carried out in the offline phase, doing so paves way for a fast and efficient online phase of the protocol. Moreover, since the computations performed in the offline phase are data-independent, not all the parties need to be active throughout this phase, placing less reliance on each party. This paradigm has proved its ability to improve the efficiency of protocols in both theoretical [28]–[33] and practical [4], [34]–[41] domains. It is especially useful in a scenario like MLaaS, where the same functions need to be performed many times and the function descriptions are known beforehand. Furthermore, we operate in the outsourced setting of MPC, which allows for an arbitrary number of parties to come together and perform their joint computation via a set of servers. Each server can be thought of as a representative for a subset of data owners, or as an independent party. The advantage of this setting is that it allows the framework to easily scale for a large number of parties and the security notions reduce to that of a standard 4PC between the servers.

Rings vs Fields: In the pursuit of practical efficiency, protocols in MPC that operate over rings are preferred to ones that work over finite fields. This is because of the way computations are carried out in standard 32/64-bit CPUs. Since these architectures have been around for a while, many algorithms are optimized for them. Moreover, operating over rings means that we do not have to override basic operations such as addition and multiplication, unlike with finite fields.

Although MPC techniques have been making a lot of progress towards being practically efficient, we cannot directly use the current best MPC protocols to perform PPML. This is mainly due to two reasons, which are:

1) MPC techniques operate in three different worlds – Arithmetic, Boolean, and Garbled. Each of these worlds is naturally better suited to carry out certain types of computations. For example, the Arithmetic domain (over a ring $\mathbb{Z}_{2^\ell}$) is more suited to perform addition whereas the Garbled world is more suited to perform division. Activation functions used in machine learning, such as Rectified Linear Unit (ReLU), have operations that alternate between multiplications and comparisons. Operating in only one of the worlds, as most of the current MPC techniques do, does not give us the maximum possible efficiency. The mixed protocol framework for MPC was first shown to be practical by TASTY [42], which combined Garbled circuits and homomorphic encryption. The idea was later applied to the ML domain by SecureML [2], ABY3 [5] etc., where protocols to switch between the three worlds were proposed. These mixed world frameworks have proven to be orders of magnitude more efficient than operating in a single world.

2) Since most of the computations and intermediate values in machine learning are decimal numbers, we embed them over a ring by allocating the least significant bits to the fractional part. But several multiplications performed may lead to an

overflow. A naive solution to avoid this is to use a large ring to accommodate a fixed number of multiplications, but the number of multiplications for machine learning varies based on the algorithm, making this infeasible. SecureML tackled this problem through truncation, which approximates the value by sacrificing the accuracy by an infinitesimal amount, performed after every multiplication. This technique, however, does not extend into the 3PC or 4PC setting, due to the attack described in ABY3, requiring us to come up with new techniques.

Frameworks such as SecureML and ABY3 have tackled both these issues in the honest majority setting by proposing ways to switch between the three worlds efficiently, as well as efficient ways to do truncation. ABY3 is a lot more efficient than SecureML, in large part due to the 3PC primitives it uses. But ABY3 cannot avoid some expensive operations such as evaluation of a Ripple Carry Adder (RCA) in its truncation and activation functions. Truncation and activation functions – ReLU and Sigmoid, need rounds proportional to the underlying ring size in ABY3. This gives a lot of scope for improvement in the efficiency, which we achieve through our 4PC framework.

A. Our Contribution

We propose an efficient framework for mixed world computations in the four-party honest majority setting with active security over the ring $\mathbb{Z}_{2^\ell}$. Our protocols are optimized for PPML and follow the offline-online paradigm. Our improvements come from having an additional honest party in the protocol. Our contributions can be summed up as follows:

1) *Efficient 4PC Protocol*: We propose an efficient four-party protocol with active security which proceeds through a masked evaluation inspired by Gordon et al. [26]. Our protocol requires 3 ring elements in the online phase per multiplication as opposed to 4 of [26], achieving a 25% improvement. This improvement is achieved by not compromising on the total cost (6 ring elements). Another significant advantage of our protocol is that the fourth party is not required for evaluation in the online phase. This is not the case with [26], where all the parties need to be online throughout the protocol execution. In addition to the stated contributions, our framework also achieves fairness without affecting the complexity of a multiplication gate.

Conversion	Ref.	Rounds	Communication
G2B	ABY3	1	κ
	This	1	3
G2A	ABY3	1	$2\ell\kappa$
	This	1	3ℓ
B2G	ABY3	1	2κ
	This	1	κ
A2G	ABY3	1	$2\ell\kappa$
	This	1	$\ell\kappa$
A2B	ABY3	$1+\log\ell$	$9\ell\log\ell+9\ell$
	This	$1+\log\ell$	$3\ell\log\ell+\ell$
B2A	ABY3	$1+\log\ell$	$9\ell\log\ell+9\ell$
	This	1	3ℓ

Table I: Online cost of share conversions of ABY3 [5] and **This** work. ℓ denotes the size of underlying ring in bits and κ denotes the computational security parameter.

2) *Fast Mixed World Computation*: We propose a framework – Trident, that is geared towards a high throughput online phase as compared to the existing alternatives. This throughput is achieved by making use of an additional honest party. Every one of the conversions we propose to switch between the worlds is more efficient in terms of online communication complexity as compared to ABY3, with our improvements ranging from $2\times$ to $2\kappa/3\times$, where κ denotes the computational security parameter. More concretely, if we aim for 128-bit computational security, our framework gives a maximum improvement of $\approx 85\times$. For instance, the technique we propose to perform bit composition (B2A) requires only 1 round, as opposed to $1+\log\ell$ rounds in ABY3, which translates to a $7\times$ gain for a 64-bit ring. The table below provides the concrete cost of our online phase in comparison to ABY3. The overall cost comparison can be found in Table IX.

3) *Efficient Truncation*: The highlight of the protocol we propose for truncation is that it can be combined with our multiplication protocol with no additional cost in the online phase. In contrast, the online cost for multiplication in ABY3 increases from 9 to 12 ring elements, which gives us a $4\times$ improvement in online communication. Moreover, we forgo the need for $(2\ell-2)$-round Ripple Carry Adders (RCA), as opposed to ABY3, in the offline phase resulting in an improvement of $63\times$ in rounds for a 64-bit ring.

Conversion	Ref.	Rounds	Communication
Multiplication with Truncation	ABY3	1	12ℓ
	This	1	3ℓ
Secure Comparison	ABY3	$\log\ell$	$18\ell\log\ell$
	This	3	$5\ell+2$
Bit2A $[\![b]\!]^{\mathbf{B}} \rightarrow [\![b]\!]$	ABY3	2	18ℓ
	This	1	3ℓ
BitInj $[\![b]\!]^{\mathbf{B}}[\![\mathrm{v}]\!] \rightarrow [\![b\mathrm{v}]\!]$	ABY3	3	27ℓ
	This	1	3ℓ
ReLU	ABY3	$3+\log\ell$	45ℓ
	This	4	$8\ell+2$
Sigmoid	ABY3	$4+\log\ell$	$81\ell+9$
	This	5	$16\ell+7$

Table II: Online cost of ML conversions of ABY3 [5] and **This** work. ℓ denotes the size of underlying ring in bits.

4) *Secure Comparison*: We propose an efficient instantiation of secure comparison, with constant round complexity. ABY3 in comparison uses an optimized Parallel Prefix Adder (PPA), which takes $\log\ell$ rounds in the online phase. This amounts to $2\times$ improvement in the online rounds for a 64-bit ring. We also improve the online communication complexity by $\approx 21\times$.

5) *ML Building Blocks*: The building blocks for ML highlighted in the Table II (more details in Table X), have improvements ranging from $2\times$ - $3\times$ in the round complexity and $5\times$ - $9\times$ in the communication complexity. For the activation functions ReLU and Sigmoid, our solution brings down the round complexity from $\mathrm{O}(\log\ell)$ (of ABY3) to a constant.

6) *Implementation*: We implement all the stated protocols and test them over LAN and WAN. We benchmark the training and prediction phases of the algorithms – Linear Regression, Logistic Regression, Neural Networks (NN), and Convolutional

Neural networks (CNN). In order to compare with ABY3, we implement their protocols as well and obtain the benchmarks in our environment. For the training phase of Linear Regression, we have improvements across different configurations in the range of $2\times$ to $251.84\times$. Similarly, for Logistic Regression, our improvements lie in the range of $2.71\times$ to $67.88\times$. The respective range of improvements for NN and CNN are $2.94\times$-$68.04\times$ and $3.19\times$-$45.64\times$. Table III gives the concrete gain of the aforementioned algorithms over the most widely used MNIST dataset [43], which has 784 features, implemented with a batch size of 128. Moreover, our framework is able to process 23 online iterations of NN in a second for a batch size of 128, over LAN. This is a huge improvement over ABY3, which can process only 2.5 iterations, that too in the semi-honest setting. Similarly, for CNN, we could process 10.46 iterations as opposed to 2 of ABY3.

Network	Linear Regression	Logistic Regression	NN	CNN
LAN	**81.08×**	**27.07×**	**68.08×**	**45.64×**
WAN	**2.17×**	**2.76×**	**2.97×**	**3.19×**

Table III: Gain in online throughput for ML Training over ABY3 [5] for $d = 784$ features and batch size of 128.

We also provide results for the prediction phase and give throughput (no. of predictions per second) comparison details for the aforementioned algorithms, using real-world datasets. The gain in online throughput for prediction ranges from $3\times$ to $145.18\times$ for Linear Regression and $3\times$ to $158.40\times$ for Logistic Regression over LAN and WAN combined. Similarly, the online throughput gain ranges from $335.44\times$ to $421.72\times$ for NN and $598.44\times$ to $759.65\times$ for CNN.

II. Preliminaries and Definitions

We consider a set of four parties $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$ that are connected by pair-wise private and authentic channels in a synchronous network. The function f to be evaluated is expressed as a circuit ckt, whose topology is publicly known 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. The term D denotes the multiplicative depth of the circuit, while $\mathsf{I}, \mathsf{O}, \mathsf{A}, \mathsf{M}$ denote the number of input wires, output wires, addition gates and multiplication gates respectively in ckt.

We use the notation w_v to denote a wire w with value v flowing through it. We use $\mathsf{g} = (\mathsf{w}_\mathsf{x}, \mathsf{w}_\mathsf{y}, \mathsf{w}_\mathsf{z}, \mathsf{op})$ to denote a gate in the ckt with left input wire w_x, right input wire w_y, output wire w_z and operation op, which is either addition $(+)$ or multiplication $(\times)$.

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

a) Shared Key Setup: In order to facilitate non-interactive communication, parties use functionality $\mathcal{F}_{\mathsf{setup}}$ that establishes pre-shared random keys for a pseudo-random function (PRF) among them. Similar setup for the three-party case can be found in [4], [5], [11], [12], [25].

In our protocols, we make use of a *collision-resistant* hash function, denoted by $\mathsf{H}()$, to save communication. We defer the formal details of key setup and hash function to Appendix A.

III. Our 4PC Protocol

In this section, we provide details for our 4PC protocol. We begin with the sharing semantics in Section III-A followed by explaining the relevant building blocks in Section III-B. We elaborate on the stages of our protocol in Section III-C. Lastly, in Section III-D, we show how to improve the security to achieve fairness.

A. Sharing Semantics

In this section, we explain three variants of secret sharing that are used in this work. The sharings work over both arithmetic ($\mathbb{Z}_{2^\ell}$) and boolean ($\mathbb{Z}_{2^1}$) rings.

a) $[\cdot]$-sharing: A value v is said to be $[\cdot]$-shared among parties P_1, P_2, P_3, if the parties P_1, P_2 and P_3 respectively hold the values $\mathsf{v}_1, \mathsf{v}_2$ and v_3 such that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2 + \mathsf{v}_3$. We use $[\cdot]_{P_i}$ to denote the $[\cdot]$-share of party P_i for $i \in \{1, 2, 3\}$.

b) $\langle\cdot\rangle$-sharing: A value v is said to be $\langle\cdot\rangle$-shared among parties P_1, P_2, P_3, if the parties P_1, P_2 and P_3 respectively holds values $(\mathsf{v}_2, \mathsf{v}_3), (\mathsf{v}_3, \mathsf{v}_1)$ and $(\mathsf{v}_1, \mathsf{v}_2)$ such that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2 + \mathsf{v}_3$. We denote $\langle\cdot\rangle$-shares of the parties as follows:

$$\langle \mathsf{v} \rangle_{P_1} = (\mathsf{v}_2, \mathsf{v}_3), \ \langle \mathsf{v} \rangle_{P_2} = (\mathsf{v}_3, \mathsf{v}_1), \ \langle \mathsf{v} \rangle_{P_3} = (\mathsf{v}_1, \mathsf{v}_2)$$

c) $[\![\cdot]\!]$-sharing: A value v is said to be $[\![\cdot]\!]$-shared among parties P_0, P_1, P_2, P_3, if

- there exist values $\lambda_\mathsf{v}, \mathsf{m}_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_\mathsf{v}$.
- parties P_1, P_2, P_3 know the value m_v in clear, while the value λ_v is $\langle\cdot\rangle$-shared among them.
- party P_0 knows $\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}$ and $\lambda_{\mathsf{v},3}$ in clear.

We denote the $[\![\cdot]\!]$-shares of the parties as follows:

$$[\![\mathsf{v}]\!]_{P_0} = (\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3}) \quad [\![\mathsf{v}]\!]_{P_1} = (\mathsf{m}_\mathsf{v}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3})$$
$$[\![\mathsf{v}]\!]_{P_2} = (\mathsf{m}_\mathsf{v}, \lambda_{\mathsf{v},3}, \lambda_{\mathsf{v},1}) \quad [\![\mathsf{v}]\!]_{P_3} = (\mathsf{m}_\mathsf{v}, \lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2})$$

We use $[\![\mathsf{v}]\!] = (\mathsf{m}_\mathsf{v}, \langle\lambda_\mathsf{v}\rangle)$ to denote the $[\![\cdot]\!]$-share of v.

d) Linearity of the secret sharing schemes: Given the $[\cdot]$-sharing of x, y and public constants c_1, c_2, parties can locally compute $[c_1\mathsf{x} + c_2\mathsf{y}] = c_1 [\mathsf{x}] + c_2 [\mathsf{y}]$.

$$\begin{aligned}[c_1\mathsf{x} + c_2\mathsf{y}] &= (c_1\mathsf{x}_1 + c_2\mathsf{y}_1, c_1\mathsf{x}_2 + c_2\mathsf{y}_2, c_1\mathsf{x}_3 + c_2\mathsf{y}_3) \\ &= c_1 [\mathsf{x}] + c_2 [\mathsf{y}]\end{aligned}$$

It is easy to see that the linearity trivially extends to $\langle\cdot\rangle$-sharing as well. That is $\langle c_1\mathsf{x} + c_2\mathsf{y}\rangle = c_1\langle\mathsf{x}\rangle + c_2\langle\mathsf{y}\rangle$. Similarly, given the $[\![\cdot]\!]$-sharing of x, y and public constants c_1, c_2, parties can locally compute $[\![c_1\mathsf{x} + c_2\mathsf{y}]\!]$. Note that the linearity property enables parties to *non-interactively* evaluate an addition gate as well as perform the multiplication of their shares with a public constant.

B. Building Blocks

a) Sharing Protocol: Protocol Π_{Sh} (Fig. 1) enables party P_i to generate $[\![\cdot]\!]$-share of value v. The offline phase is done using the pre-shared keys in such a way that P_i will get the entire mask λ. During the online phase, P_i computes $\mathsf{m_v}$ and sends to P_1, P_2, P_3 who exchange the hash values to check for consistency.

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

Offline:

- If $P_i = P_0$, parties in $\mathcal{P} \setminus \{P_j\}$ together sample $\lambda_{\mathsf{v},j}$ for $j \in \{1,2,3\}$.
- If $P_i = P_k$ for $k \in \{1,2,3\}$, parties in $\mathcal{P}$ together sample $\lambda_{\mathsf{v},k}$. In addition, parties in $\mathcal{P} \setminus \{P_j\}$ together sample $\lambda_{\mathsf{v},j}$ for $j \in \{1,2,3\} \setminus \{k\}$.

Online:

- P_i computes $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2, P_3.
- P_1, P_2, P_3 mutually exchange $\mathsf{H}(\mathsf{m_v})$ and `abort` if the received values are inconsistent.

Figure 1: $[\![\cdot]\!]$-sharing of a value v by party P_i.

Looking ahead, we also encounter scenarios where party P_0 has to generate $\langle\cdot\rangle$-sharing of a value v in the offline phase. We call the resultant protocol as Π_{aSh} and the formal details appear in Fig. 2.

Protocol $\Pi_{\mathsf{aSh}}(P_0, \mathsf{v})$

Offline:

- Parties in $\mathcal{P} \setminus \{P_1\}$ sample random $\mathsf{v}_1 \in \mathbb{Z}_{2^\ell}$, while parties in $\mathcal{P} \setminus \{P_2\}$ sample random v_2.
- P_0 computes $\mathsf{v}_3 = -(\mathsf{v} + \mathsf{v}_1 + \mathsf{v}_2)$ and sends it to both P_1 and P_2, who exchange $\mathsf{H}(\mathsf{v}_3)$ and `abort` if there is a mismatch.

Figure 2: $\langle\cdot\rangle$-sharing of a value v by party P_0.

b) Reconstruction Protocol: Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, \mathsf{v})$ (Fig. 3) enables parties in $\mathcal{P}$ to compute v, given its $[\![\cdot]\!]$-share. Towards this, each party receives the missing share from one other party and hash of the missing share from one of the other two parties. If the received shares are consistent, he/she will proceed with the reconstruction. Reconstruction towards a single party can be viewed as a special case of this protocol.

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

Online:

- P_1 receives $\lambda_{\mathsf{v},1}$ and $\mathsf{H}(\lambda_{\mathsf{v},1})$ from P_2 and P_0 respectively.
- P_2 receives $\lambda_{\mathsf{v},2}$ and $\mathsf{H}(\lambda_{\mathsf{v},2})$ from P_3 and P_0 respectively.
- P_3 receives $\lambda_{\mathsf{v},3}$ and $\mathsf{H}(\lambda_{\mathsf{v},3})$ from P_1 and P_0 respectively.
- P_0 receives $\mathsf{m_v}$ and $\mathsf{H}(\mathsf{m_v})$ from P_1 and P_2 respectively.

P_i for $i \in \{0,1,2,3\}$ `abort` if the received values are inconsistent. Else computes $\mathsf{v} = \mathsf{m_v} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2} - \lambda_{\mathsf{v},3}$.

Figure 3: Reconstruction of value v among parties in $\mathcal{P}$.

C. Stages of our 4PC protocol

Our protocol Π_{4PC} consists of three stages, namely – Input Sharing, Evaluation and Output Reconstruction. We elaborate on each of these stages below.

a) Input Sharing: For each wire $\mathsf{w_v}$ holding the value v, of which $P_i \in \mathcal{P}$ is the owner, he/she generates $[\![\cdot]\!]$-share of v by executing the $\Pi_{\mathsf{Sh}}(P_i, \mathsf{v})$ protocol.

b) Evaluation: In this stage, parties evaluate the circuit in a topological order, where the following invariant is maintained for every gate g: given the inputs of g in $[\![\cdot]\!]$-shared fashion, the output is generated in the $[\![\cdot]\!]$-shared fashion. For the case of an addition gate $\mathsf{g} = (\mathsf{w_x}, \mathsf{w_y}, \mathsf{w_z}, +)$, the linearity of our sharing scheme maintains this invariant.

For a multiplication gate $\mathsf{g} = (\mathsf{w_x}, \mathsf{w_y}, \mathsf{w_z}, \times)$, the protocol proceeds as follows: during the offline phase, parties P_1, P_2, P_3 locally compute $[\cdot]$-shares of $\gamma_{\mathsf{xy}} = \lambda_{\mathsf{x}}\lambda_{\mathsf{y}}$, followed by exchanging them to form a $\langle\cdot\rangle$-sharing of γ_{xy}. Before exchanging the shares of γ_{xy}, the parties randomize the shares by adding a share of 0 to the share of γ_{xy} to prevent leakage. In addition, P_0 helps the parties in verifying the correctness of shares received in the aforementioned step. During the online phase, the goal is to compute $\mathsf{m_z}$. Note that,

$$\begin{aligned}\mathsf{m_z} &= \mathsf{z} + \lambda_{\mathsf{z}} = \mathsf{xy} + \lambda_{\mathsf{z}} = (\mathsf{m_x} - \lambda_{\mathsf{x}})(\mathsf{m_y} - \lambda_{\mathsf{y}}) + \lambda_{\mathsf{z}} \\ &= \mathsf{m_x}\mathsf{m_y} - \lambda_{\mathsf{x}}\mathsf{m_y} - \lambda_{\mathsf{y}}\mathsf{m_x} + \lambda_{\mathsf{x}}\lambda_{\mathsf{y}} + \lambda_{\mathsf{z}}\end{aligned}$$

Parties P_1, P_2, P_3 locally compute $[\cdot]$-share of $\mathsf{m_z} - \mathsf{m_x}\mathsf{m_y}$ followed by an exchange to reconstruct $\mathsf{m_z} - \mathsf{m_x}\mathsf{m_y}$. By the nature of our secret-sharing scheme, every missing share can be computed by two parties. This facilitates the parties to verifiably reconstruct $\mathsf{m_z} - \mathsf{m_x}\mathsf{m_y}$ by having one party send the missing share and the other send a hash of the same.

Each of P_1, P_2, P_3 locally add $\mathsf{m_x}\mathsf{m_y}$ to the result to obtain $\mathsf{m_z}$. We call the resultant protocol Π_{Mult} (Fig. 4).

Protocol $\Pi_{\mathsf{Mult}}(\mathsf{w_x}, \mathsf{w_y}, \mathsf{w_z})$

Offline:

- Parties in $\mathcal{P} \setminus \{P_j\}$ together sample $\lambda_{\mathsf{z},j}$ for $j \in \{1,2,3\}$.
- Parties invoke protocol Π_{Zero} (Fig. 22) to generate A, B, Γ such that $A + B + \Gamma = 0$. Parties locally compute the following:
  - P_0, P_1 compute $\gamma_{\mathsf{xy},2} = \lambda_{\mathsf{x},2}\lambda_{\mathsf{y},2} + \lambda_{\mathsf{x},2}\lambda_{\mathsf{y},3} + \lambda_{\mathsf{x},3}\lambda_{\mathsf{y},2} + A$.
  - P_0, P_2 compute $\gamma_{\mathsf{xy},3} = \lambda_{\mathsf{x},3}\lambda_{\mathsf{y},3} + \lambda_{\mathsf{x},3}\lambda_{\mathsf{y},1} + \lambda_{\mathsf{x},1}\lambda_{\mathsf{y},3} + B$.
  - P_0, P_3 compute $\gamma_{\mathsf{xy},1} = \lambda_{\mathsf{x},1}\lambda_{\mathsf{y},1} + \lambda_{\mathsf{x},1}\lambda_{\mathsf{y},2} + \lambda_{\mathsf{x},2}\lambda_{\mathsf{y},1} + \Gamma$.
- Parties exchange the following:
  - P_1 receives $\gamma_{\mathsf{xy},3}$ and $\mathsf{H}(\gamma_{\mathsf{xy},3})$ from P_2 and P_0 respectively.
  - P_2 receives $\gamma_{\mathsf{xy},1}$ and $\mathsf{H}(\gamma_{\mathsf{xy},1})$ from P_3 and P_0 respectively.
  - P_3 receives $\gamma_{\mathsf{xy},2}$ and $\mathsf{H}(\gamma_{\mathsf{xy},2})$ from P_1 and P_0 respectively.
- P_i for $i \in \{1,2,3\}$ `abort` if received values are inconsistent.

Online: Let $\mathsf{m'_z} = \mathsf{m_z} - \mathsf{m_x}\mathsf{m_y}$.

- Parties locally compute the following:
  - P_1, P_3 compute $\mathsf{m'_{z},2} = -\lambda_{\mathsf{x},2}\mathsf{m_y} - \lambda_{\mathsf{y},2}\mathsf{m_x} + \gamma_{\mathsf{xy},2} + \lambda_{\mathsf{z},2}$.
  - P_2, P_1 compute $\mathsf{m'_{z},3} = -\lambda_{\mathsf{x},3}\mathsf{m_y} - \lambda_{\mathsf{y},3}\mathsf{m_x} + \gamma_{\mathsf{xy},3} + \lambda_{\mathsf{z},3}$.
  - P_3, P_2 compute $\mathsf{m'_{z},1} = -\lambda_{\mathsf{x},1}\mathsf{m_y} - \lambda_{\mathsf{y},1}\mathsf{m_x} + \gamma_{\mathsf{xy},1} + \lambda_{\mathsf{z},1}$.
- Parties exchange the following:
  - P_1 receives $\mathsf{m'_{z},1}$ and $\mathsf{H}(\mathsf{m'_{z},1})$ from P_2 and P_3 respectively.
  - P_2 receives $\mathsf{m'_{z},2}$ and $\mathsf{H}(\mathsf{m'_{z},2})$ from P_3 and P_1 respectively.

– P_3 receives $\mathsf{m}'_{\mathsf{z},3}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},3})$ from P_1 and P_2 respectively.
– P_i for $i \in \{1,2,3\}$ `abort` if the received values are inconsistent. Else, he / she computes $\mathsf{m}_\mathsf{z} = (\mathsf{m}'_{\mathsf{z},1} + \mathsf{m}'_{\mathsf{z},2} + \mathsf{m}'_{\mathsf{z},3}) + \mathsf{m}_\mathsf{x}\mathsf{m}_\mathsf{y} = \mathsf{m}'_\mathsf{z} + \mathsf{m}_\mathsf{x}\mathsf{m}_\mathsf{y}$.

Figure 4: Multiplication Protocol.

As a very important optimization, note that the exchange of hash values for every multiplication gate can be delayed until the output reconstruction stage. Moreover, all the corresponding values can be appended and hashed, resulting in an overall communication of only 3 ring elements.

c) Output Reconstruction: For each of the output wire w_y with value y, parties execute protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, [\![\mathsf{y}]\!])$ to reconstruct the output.

Correctness and Security: We prove the correctness of Π_{4PC} below and defer the security details to Appendix F.

Theorem III.1 (Correctness)**.** *Protocol* Π_{4PC} *is correct.*

Proof: We claim that for every wire in ckt, the parties hold a $[\![\cdot]\!]$-sharing of the wire value in Π_{4PC}. The correctness for the input and output wires follows from Π_{Sh} and Π_{Rec} respectively. The claim for addition gates follows from the linearity of $[\![\cdot]\!]$-sharing. For a multiplication gate $\mathsf{g} = (\mathsf{w}_\mathsf{x}, \mathsf{w}_\mathsf{y}, \mathsf{w}_\mathsf{z}, \times)$, when evaluated using Π_{Mult}, the parties receive $\mathsf{xy} + \lambda_\mathsf{z}$ in the online phase, resulting in obtaining $[\![\mathsf{z}]\!]$. The correctness of $\mathsf{xy} + \lambda_\mathsf{z}$ is ensured through the verified reconstruction as shown in Π_{Mult}. ■

Theorem III.2 (Communication Efficiency)**.** Π_{4PC} *requires one round with an amortized communication of* $3\mathsf{M}$ *ring elements during the offline phase. In the online phase,* Π_{4PC} *requires one round with an amortized communication of at most* $3\mathsf{I}$ *ring elements in the Input-sharing stage,* D *rounds with an amortized communication of* $3\mathsf{M}$ *ring elements for evaluation stage and one round with an amortized communication of* $3\mathsf{O}$ *elements for the output-reconstruction stage.*

D. Achieving Fairness

For fairness, we need to ensure that all the parties are *alive* in the protocol during the output reconstruction stage. On top of this, we also need to prevent the adversary from mounting a selective abort attack, where he can make some of the honest parties abort the protocol. To achieve this, parties P_1, P_2, P_3 set a bit b to `continue`, if the verification of the multiplication gates was successful, else set it to `abort`, and send it to P_0. P_0 then sends `abort` back to all the parties if one of the parties sends `abort`, thus ensuring aliveness. Remaining parties then exchange their reply from P_0 and follow the honest-majority in deciding whether to proceed or abort. Since there can be only 1 corruption, all the parties will now be on the same page, preventing a selective abort. If the parties decide to proceed, they exchange the missing shares. Using the fact that there is at most 1 corruption and the structure of our secret-sharing scheme, the most commonly received missing share will be consistent among the honest parties.

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

Online:

– P_1, P_2, P_3 set bit $\mathsf{b} =$ `abort` if the verification for multiplication fails. Else set $\mathsf{b} =$ `continue`.
– P_1, P_2, P_3 send b to P_0 who sends back `abort`, if he/she receives at least one `abort` bit. Else sends `continue` to P_1, P_2, P_3.
– P_1, P_2, P_3 mutually exchange the message received from P_0. Parties `abort` if the majority of the messages received are `abort`. Else they exchange the missing share as follows:
  – P_0 receives m_v from P_1, P_2 and $\mathsf{H}(\mathsf{m}_\mathsf{v})$ from P_3 respectively.
  – P_1 receives $\lambda_{\mathsf{v},1}$ from P_2, P_3 and $\mathsf{H}(\lambda_{\mathsf{v},1})$ from P_0 respectively.
  – P_2 receives $\lambda_{\mathsf{v},2}$ from P_3, P_1 and $\mathsf{H}(\lambda_{\mathsf{v},2})$ from P_0 respectively.
  – P_3 receives $\lambda_{\mathsf{v},3}$ from P_1, P_2 and $\mathsf{H}(\lambda_{\mathsf{v},3})$ from P_0 respectively.
– P_i for $i \in \{0,1,2,3\}$ chooses the missing share that forms the majority and computes $\mathsf{v} = \mathsf{m}_\mathsf{v} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2} - \lambda_{\mathsf{v},3}$.

Figure 5: Fair reconstruction of value v among parties in $\mathcal{P}$.

IV. Mixed Protocol Framework

In this section, we present our mixed protocol framework, Trident. Before we go into the details of it, we discuss another world of MPC, called The Garbled World. To evaluate circuits over a ring $\mathbb{Z}_{2^\ell}$, we operate in the arithmetic world and to evaluate boolean circuits ($\mathbb{Z}_{2^\ell}$) we use either the boolean world or the Garbled world, depending on the operation being performed. Superscripts {**A, B, G**} are used to indicate the respective worlds. If there is no superscript, the values are assumed to operate in the arithmetic world.

A. The Garbled World

For the Garbled world, we use the MRZ [16] scheme in the 4PC setting. In 4PC, parties P_1, P_2, P_3 act as the garblers and the party P_0 acts as the sole evaluator. As an optimisation, P_0 can share his inputs with only P_1, P_2 instead of all three parties. For cross-verification, P_1 sends the garbled circuit to P_0 while P_2 sends a hash of the it to P_0. We incorporate the recent optimisations including free XOR [**?**], [44], [45], half-gates [46], [47], fixed-key AES garbling [48]. As opposed to the dishonest majority setting, this scheme removes the need for expensive public key primitives (in terms of communication) such as Oblivious Transfers altogether. We present the protocols for a single bit, and each operation can be performed ℓ times in parallel to support ℓ-bit values.

a) Sharing Semantics: For a bit v, $[\![\mathsf{v}]\!]^{\mathbf{G}}$ is defined as $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_i} = \mathsf{K}^0_\mathsf{v} \in \{0,1\}^\kappa$ for $i \in \{1,2,3\}$ and $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_0} = \mathsf{K}^\mathsf{v}_\mathsf{v} = \mathsf{K}^0_\mathsf{v} \oplus \mathsf{vR}$, where κ is the computational security parameter. Here R is a global *offset* with the least significant bit as one, and is known only to P_1, P_2, P_3 (generated by shared randomness), and R is common across all the $[\![\cdot]\!]^{\mathbf{G}}$-sharing. It is easy to see that XOR of the least significant bit of $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_1}$ (resp. $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_2}, [\![\mathsf{v}]\!]^{\mathbf{G}}_{P_3}$) and $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_0}$ is v. For a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, we abuse the notation $[\![\mathsf{v}]\!]^{\mathbf{G}}$ to denote the set of $[\![\cdot]\!]^{\mathbf{G}}$-shares of each bit of v.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(P_i, \mathsf{v})$

Offline:

- P_1, P_2, P_3 samples a random $\mathsf{K}^0_{\mathsf{v}} \in \{0,1\}^\kappa$, computes $\mathsf{K}^1_{\mathsf{v}} = \mathsf{K}^0_{\mathsf{v}} \oplus \mathsf{R}$ and set $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_1} = [\![\mathsf{v}]\!]^{\mathbf{G}}_{P_2} = [\![\mathsf{v}]\!]^{\mathbf{G}}_{P_3} = \mathsf{K}^0_{\mathsf{v}}$.
- P_1, P_2, P_3 compute commitment of $\mathsf{K}^0_{\mathsf{v}}, \mathsf{K}^1_{\mathsf{v}}$. P_1, P_2 send the commitments to P_0 in a random permuted order, who `abort` if the received commitments mismatch.

Online:

- P_i sends $\mathsf{K}^0_{\mathsf{v}} \oplus \mathsf{vR}$ to P_0, who sets it as $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_0}$.
- P_i decommits the right key $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$ to P_0, who `abort` if the decommitment is incorrect.

Figure 6: $[\![\cdot]\!]^{\mathbf{G}}$-sharing of v by P_i for $i \in \{1,2,3\}$.

b) Input Sharing: Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(P_i, \mathsf{v})$ enables P_i to generate $[\![\cdot]\!]^{\mathbf{G}}$-sharing of value v. During the protocol, P_0 needs to ensure that it obtains the correct $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$. To tackle this, we make the garblers commit both keys to P_0, who can then verify the correctness by cross-checking the commitments received. The formal details for the case when P_i is one of the garblers appear in Fig. 6.

If $P_i = P_0$, then $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(P_i, \mathsf{v})$ proceeds as follows: P_0 samples random bit v_1, computes $\mathsf{v}_2 = \mathsf{v} \oplus \mathsf{v}_1$ and sends v_1 and v_2 to P_1 and P_2 respectively. Parties execute $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(P_1, \mathsf{v}_1)$ and $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(P_2, \mathsf{v}_2)$ to generate $[\![\mathsf{v}_1]\!]^{\mathbf{G}}$ and $[\![\mathsf{v}_2]\!]^{\mathbf{G}}$ respectively. Parties then locally compute $[\![\mathsf{v}]\!]^{\mathbf{G}} = [\![\mathsf{v}_1]\!]^{\mathbf{G}} \oplus [\![\mathsf{v}_2]\!]^{\mathbf{G}}$, using the XOR gate evaluation method via the free-XOR technique. Here, the commitments of the keys need not be permuted, as P_0 already knows the actual v_1 and v_2. As an optimization, the computation of $[\![\mathsf{v}_1]\!]^{\mathbf{G}}$ can be offloaded to the offline phase.

c) Reconstruction: If $P_i = P_0$, then P_1, P_2 send the least significant bit of their shares and P_i verifies if it received the same bit from both P_1 and P_2. If P_i is one of the garblers, then P_0 sends its share to P_i. Due to the *authenticity* of the underlying garbling scheme [49], a corrupt P_0 cannot send an incorrect share to P_i. If there are multiple reconstructions towards $P_i \in \{P_1, P_2, P_3\}$, P_0 can send the least significant bit of its shares along with a hash of all the corresponding shares.

d) Operations: Let $\mathsf{u}, \mathsf{v} \in \{0,1\}$ be $[\![\cdot]\!]^{\mathbf{G}}$-shared with P_1, P_2, P_3 holding the shares $(\mathsf{K}^0_{\mathsf{u}}, \mathsf{K}^0_{\mathsf{v}})$, and P_0 holding the shares $(\mathsf{K}^0_{\mathsf{u}} \oplus \mathsf{uR}, \mathsf{K}^0_{\mathsf{v}} \oplus \mathsf{vR})$. Let c denote the output.

– *XOR:* The parties locally compute $[\![\mathsf{c}]\!]^{\mathbf{G}} = [\![\mathsf{u}]\!]^{\mathbf{G}} \oplus [\![\mathsf{v}]\!]^{\mathbf{G}}$.
– *AND:* P_1, P_2, P_3 sample random $\mathsf{K}^0_{\mathsf{c}} \in \{0,1\}^\kappa$, compute $\mathsf{K}^1_{\mathsf{u}} = \mathsf{K}^0_{\mathsf{u}} \oplus \mathsf{R}$ and construct a garbled table for AND using the garbling scheme described in [46], [48]. P_1 sends the garbled table to P_0, while P_2 sends a hash of the table to P_0. P_0 evaluates the table[1] to obtain $[\![\mathsf{c}]\!]^{\mathbf{G}}_{P_0} = \mathsf{K}^{\mathsf{c}}_{\mathsf{c}}$. P_i for $i \in \{1,2,3\}$ sets $[\![\mathsf{c}]\!]^{\mathbf{G}}_{P_i} = \mathsf{K}^0_{\mathsf{c}}$.

B. Building Blocks

a) Verifiable Arithmetic/Boolean Sharing: Protocol Π_{vSh} (Fig. 7) allows two parties P_i, P_j to generate $[\![\cdot]\!]$-sharing of value v in a verifiable manner. On a high level, P_i executes Π_{Sh} on v, while P_j helps in verification by sending $\mathsf{H}(\mathsf{m}_{\mathsf{v}})$ to parties P_1, P_2, P_3.

[1]The garbled table can be send during the offline phase, while P_0 needs to evaluate the garbled circuit during the online phase.

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

Offline: Parties execute offline steps of $\Pi_{\mathsf{Sh}}(P_i, \mathsf{v})$.

Online:

- P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2, P_3.
- P_j computes $\mathsf{H}(\mathsf{m}_{\mathsf{v}})$ and sends to P_1, P_2, P_3, who `abort` if the received values are inconsistent.

Figure 7: Verifiable Arithmetic/Boolean sharing of a value v.

We observe that the parties can *non-interactively* generate $[\![\cdot]\!]$-sharing of a value v when all of the parties P_1, P_2, P_3 know v. Parties set $\lambda_{\mathsf{v},1} = \lambda_{\mathsf{v},2} = \lambda_{\mathsf{v},3} = 0$ and $\mathsf{m}_{\mathsf{v}} = \mathsf{v}$. We abuse the notation and use $\Pi_{\mathsf{vSh}}(P_1, P_2, P_3, \mathsf{v})$ to denote this protocol.

b) Verifiable Garbled Sharing: Protocol $\Pi^{\mathbf{G}}_{\mathsf{vSh}}$ (Fig. 8) is adapted from ABY3 [5] and allows two parties P_i, P_j to generate $[\![\cdot]\!]^{\mathbf{G}}$-sharing of value v in a verifiable manner. When P_i, P_j are both garblers, one of them can send the key while the other can send just the hash to check for inconsistency. If $P_0 = P_j$, the other parties (P_1, P_2) send commitments of the keys in order, to P_0. In addition, P_i sends the decommitment of the actual key to P_0.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{vSh}}(P_i, P_j, \mathsf{v})$

Offline: P_1, P_2, P_3 locally sample random $\mathsf{K}^0_{\mathsf{v}} \in \{0,1\}^\kappa$, compute $\mathsf{K}^1_{\mathsf{v}} = \mathsf{K}^0_{\mathsf{v}} \oplus \mathsf{R}$ and set $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_1} = [\![\mathsf{v}]\!]^{\mathbf{G}}_{P_2} = [\![\mathsf{v}]\!]^{\mathbf{G}}_{P_3} = \mathsf{K}^0_{\mathsf{v}}$.

Online:

- **If** $(P_i, P_j) = (P_1, P_0)$**:**
  - P_1, P_2 compute commitments $\mathsf{Com}(\mathsf{K}^0_{\mathsf{v}}), \mathsf{Com}(\mathsf{K}^1_{\mathsf{v}})$ and send it to P_0. In addition, P_1 sends decommitment of $\mathsf{Com}(\mathsf{K}^{\mathsf{v}}_{\mathsf{v}})$ to P_0.
  - P_0 `abort` if either the received commitments are inconsistent or the decommitment is incorrect. Else he/she sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_0} = \mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$.
- **If** $(P_i, P_j) = (P_k, P_0)$ **for** $k \in \{2,3\}$**:** The steps are similar as above.
- **If** $(P_i, P_j) = (P_1, P_2)$**:** P_1 and P_2 sends $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$ and $\mathsf{H}(\mathsf{K}^{\mathsf{v}}_{\mathsf{v}})$ respectively to P_0, who `abort` if the received values are inconsistent. Else he/she sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_{P_0} = \mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$.
- **If** $(P_i, P_j) = (P_1, P_3)$ **or** (P_2, P_3)**:** The steps are similar as above.

Figure 8: Verifiable Garbled sharing of a value v.

c) Dot Product: Given two vectors $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$, each of size d, Π_{DotP} (Fig. 9) computes the dot product $\mathsf{z} = \vec{\mathsf{x}} \odot \vec{\mathsf{y}}$. ABY3 [5] and ASTRA [25] have proposed efficient dot product protocols for the semi-honest and malicious settings. In the semi-honest case, their dot product cost is the same as a single multiplication, but for the malicious case it scales with the vector size. In comparison, the cost of our dot product is independent of the vector size. On a high level, instead of performing reconstruction for each multiplication $\mathsf{x}_j \cdot \mathsf{y}_j$ for $j \in \{1, \ldots, d\}$, parties locally add their shares corresponding to all the multiplications and perform a single exchange.

Protocol $\Pi_{\mathsf{DotP}}(\vec{\mathsf{x}}, \vec{\mathsf{y}})$

Let $\mathsf{z} = \vec{\mathsf{x}} \odot \vec{\mathsf{y}}$.

Offline:

- Parties in $\mathcal{P} \setminus \{P_j\}$ together sample $\lambda_{\mathsf{z},j}$ for $j \in \{1,2,3\}$.
- Parties invoke protocol Π_{Zero} (Fig. 22) to generate A, B, Γ such that $A+B+\Gamma = 0$. Parties locally compute the following:
  - P_0, P_1 compute $\gamma_{\mathsf{xy},2} = \sum_{j=1}^{d} \gamma_{\mathsf{x}_j\mathsf{y}_j,2} = \sum_{j=1}^{d} (\lambda_{\mathsf{x}_j,2}\lambda_{\mathsf{y}_j,2} + \lambda_{\mathsf{x}_j,2}\lambda_{\mathsf{y}_j,3} + \lambda_{\mathsf{x}_j,3}\lambda_{\mathsf{y}_j,2}) + A$.
  - P_0, P_2 compute $\gamma_{\mathsf{xy},3} = \sum_{j=1}^{d} \gamma_{\mathsf{x}_j\mathsf{y}_j,3} = \sum_{j=1}^{d} (\lambda_{\mathsf{x}_j,3}\lambda_{\mathsf{y}_j,3} + \lambda_{\mathsf{x}_j,3}\lambda_{\mathsf{y}_j,1} + \lambda_{\mathsf{x}_j,1}\lambda_{\mathsf{y}_j,3}) + B$.
  - P_0, P_3 compute $\gamma_{\mathsf{xy},1} = \sum_{j=1}^{d} \gamma_{\mathsf{x}_j\mathsf{y}_j,1} = \sum_{j=1}^{d} (\lambda_{\mathsf{x}_j,1}\lambda_{\mathsf{y}_j,1} + \lambda_{\mathsf{x}_j,1}\lambda_{\mathsf{y}_j,2} + \lambda_{\mathsf{x}_j,2}\lambda_{\mathsf{y}_j,1}) + \Gamma$.
- Parties exchange the following:
  - P_1 receives $\gamma_{\mathsf{xy},3}$ and $\mathsf{H}(\gamma_{\mathsf{xy},3})$ from P_2 and P_0 respectively.
  - P_2 receives $\gamma_{\mathsf{xy},1}$ and $\mathsf{H}(\gamma_{\mathsf{xy},1})$ from P_3 and P_0 respectively.
  - P_3 receives $\gamma_{\mathsf{xy},2}$ and $\mathsf{H}(\gamma_{\mathsf{xy},2})$ from P_1 and P_0 respectively.
- P_i for $i \in \{1,2,3\}$ `abort` if the received values are inconsistent.

Online: Let $\mathsf{m}'_{\mathsf{z}} = \mathsf{m}_{\mathsf{z}} - \sum_{j=1}^{d} \mathsf{m}_{\mathsf{x}_j}\mathsf{m}_{\mathsf{y}_j}$.

- Parties locally compute the following:
  - P_1, P_3: $\mathsf{m}'_{\mathsf{z},2} = \sum_{j=1}^{d} (-\lambda_{\mathsf{x}_j,2}\mathsf{m}_{\mathsf{y}_j} - \lambda_{\mathsf{y}_j,2}\mathsf{m}_{\mathsf{x}_j}) + \gamma_{\mathsf{xy},2} + \lambda_{\mathsf{z},2}$.
  - P_2, P_1: $\mathsf{m}'_{\mathsf{z},3} = \sum_{j=1}^{d} (-\lambda_{\mathsf{x}_j,3}\mathsf{m}_{\mathsf{y}_j} - \lambda_{\mathsf{y}_j,3}\mathsf{m}_{\mathsf{x}_j}) + \gamma_{\mathsf{xy},3} + \lambda_{\mathsf{z},3}$.
  - P_3, P_2: $\mathsf{m}'_{\mathsf{z},1} = \sum_{j=1}^{d} (-\lambda_{\mathsf{x}_j,1}\mathsf{m}_{\mathsf{y}_j} - \lambda_{\mathsf{y}_j,1}\mathsf{m}_{\mathsf{x}_j}) + \gamma_{\mathsf{xy},1} + \lambda_{\mathsf{z},1}$.
- Parties exchange the following:
  - P_1 receives $\mathsf{m}'_{\mathsf{z},1}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},1})$ from P_2 and P_3 respectively.
  - P_2 receives $\mathsf{m}'_{\mathsf{z},2}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},2})$ from P_3 and P_1 respectively.
  - P_3 receives $\mathsf{m}'_{\mathsf{z},3}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},3})$ from P_1 and P_2 respectively.
- P_i for $i \in \{1,2,3\}$ `abort` if the received values are inconsistent. Else, he / she computes $\mathsf{m}_{\mathsf{z}} = (\mathsf{m}'_{\mathsf{z},1} + \mathsf{m}'_{\mathsf{z},2} + \mathsf{m}'_{\mathsf{z},3}) + + \sum_{j=1}^{d} (\mathsf{m}_{\mathsf{x}_j}\mathsf{m}_{\mathsf{y}_j}) = \mathsf{m}'_{\mathsf{z}} + \sum_{j=1}^{d} (\mathsf{m}_{\mathsf{x}_j}\mathsf{m}_{\mathsf{y}_j})$.

Figure 9: Dot Product Protocol.

C. Sharing Conversions

We now discuss the inter-sharing conversions among **A**rithmetic, **B**oolean, and **G**arbled sharing.

a) Garbled to Boolean Sharing (G2B): To convert a garbled share into the boolean world, P_1, P_2 first generate the garbled and boolean shares of a random value (r) using their shared randomness in the offline phase. In addition, they communicate the garbled circuit which performs the XOR of two bits along with the decoding information (Note that the garbled circuit does not have to be communicated due to the free XOR technique). In the online phase, P_0 evaluates and obtains $\mathsf{v} \oplus \mathsf{r}$ and sends it to P_3 along with the hash of the corresponding key. Authenticity of the underlying garbling scheme ensures that a corrupt P_0 cannot send the wrong bit, as he will not be able to guess the right key for it.

Protocol Π_{G2B}

Offline:

- P_1, P_2 locally sample random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. Parties execute $\Pi^{\mathbf{G}}_{\mathsf{vSh}}(P_1, P_2, \mathsf{r})$ and $\Pi^{\mathbf{B}}_{\mathsf{vSh}}(P_1, P_2, \mathsf{r})$ to generate $[\![\mathsf{r}]\!]^{\mathbf{G}}$ and $[\![\mathsf{r}]\!]^{\mathbf{B}}$ respectively.
- P_1, P_2, P_3 garble a boolean adder circuit $\mathsf{Add}(\mathsf{x}, \mathsf{y})$ that computes $\mathsf{x} \oplus \mathsf{y}$. P_1 sends the garbled circuit along with the decoding information to P_0, while P_2 sends a combined hash to P_0.

Online:

- P_0 evaluates the circuit Add on v and r to obtain $\mathsf{v} \oplus \mathsf{r}$. P_0 sends $\mathsf{v} \oplus \mathsf{r}$ along with a hash of the actual key corresponding to $\mathsf{v} \oplus \mathsf{r}$ to P_3. P_3 `abort` if the received values are inconsistent.
- Else, parties execute $\Pi^{\mathbf{B}}_{\mathsf{vSh}}(P_3, P_0, \mathsf{v} \oplus \mathsf{r})$ to generate $[\![\mathsf{v} \oplus \mathsf{r}]\!]^{\mathbf{B}}$.
- Parties locally compute $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v} \oplus \mathsf{r}]\!]^{\mathbf{B}} \oplus [\![\mathsf{r}]\!]^{\mathbf{B}}$.

Figure 10: Garbled to Boolean Sharing.

b) Garbled to Arithmetic Sharing (G2A): This conversion proceeds in a similar way as Π_{G2B}. The major difference is that instead of the garbled circuit for XOR, the parties communicate a circuit for subtraction of two ℓ-bit values. Note that P_0 needs to communicate only a single hash combining all the keys corresponding to the ℓ-bits of $\mathsf{v} - \mathsf{r}$.

Protocol Π_{G2A}

Offline:

- P_1, P_2 locally sample random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. Parties execute $\Pi^{\mathbf{G}}_{\mathsf{vSh}}(P_1, P_2, \mathsf{r})$ and $\Pi^{\mathbf{A}}_{\mathsf{vSh}}(P_1, P_2, \mathsf{r})$ to generate $[\![\mathsf{r}]\!]^{\mathbf{G}}$ and $[\![\mathsf{r}]\!]^{\mathbf{A}}$ respectively.
- P_1, P_2, P_3 garble a subtractor circuit $\mathsf{Sub}(\mathsf{x}, \mathsf{y})$ that computes $\mathsf{x} - \mathsf{y}$. P_1 sends the garbled circuit along with the decoding information to P_0, while P_2 sends a combined hash to P_0, who `abort` if the received values are inconsistent.

Online:

- P_0 evaluates the circuit Sub on v and r to obtain $\mathsf{v} - \mathsf{r}$. P_0 sends $\mathsf{v} - \mathsf{r}$ along with a combined hash of all the actual keys corresponding to $\mathsf{v} - \mathsf{r}$ to P_3. P_3 `abort` if the received values are inconsistent.
- Else, parties execute $\Pi^{\mathbf{A}}_{\mathsf{vSh}}(P_3, P_0, \mathsf{v} - \mathsf{r})$ to generate $[\![\mathsf{v} - \mathsf{r}]\!]^{\mathbf{A}}$.
- Parties locally compute $[\![\mathsf{v}]\!]^{\mathbf{A}} = [\![\mathsf{v} - \mathsf{r}]\!]^{\mathbf{A}} + [\![\mathsf{r}]\!]^{\mathbf{A}}$.

Figure 11: Garbled to Arithmetic Sharing.

c) Boolean to Garbled Sharing (B2G): Since the bit $\mathsf{v} = (\mathsf{m}_{\mathsf{v}} \oplus \lambda_{\mathsf{v},1}) \oplus (\lambda_{\mathsf{v},2} \oplus \lambda_{\mathsf{v},3})$ in the boolean world, if we can get the garbled shares of $\mathsf{x} = (\mathsf{m}_{\mathsf{v}} \oplus \lambda_{\mathsf{v},1})$ and $\mathsf{y} = (\lambda_{\mathsf{v},2} \oplus \lambda_{\mathsf{v},3})$, parties can use the free XOR technique to compute the garbled shares of v locally. Each of x, y is possessed by two parties, enabling them to verifiably generate the garbled shares using the protocol $\Pi^{\mathbf{G}}_{\mathsf{vSh}}$.

Protocol Π_{B2G}

Offline: P_0, P_1 execute $\Pi^{\mathbf{G}}_{\mathsf{vSh}}(P_1, P_0, \mathsf{y})$ to generate $[\![\mathsf{y}]\!]^{\mathbf{G}}$ where $\mathsf{y} = \lambda_{\mathsf{v},2} \oplus \lambda_{\mathsf{v},3}$.

Online:

- P_2, P_3 execute $\Pi^{\mathbf{G}}_{\mathsf{vSh}}(P_2, P_3, \mathsf{x})$ to generate $[\![\mathsf{x}]\!]^{\mathbf{G}}$ where $\mathsf{x} = \mathsf{m}_{\mathsf{v}} \oplus \lambda_{\mathsf{v},1}$.
- Parties locally compute $[\![\mathsf{v}]\!]^{\mathbf{G}} = [\![\mathsf{x}]\!]^{\mathbf{G}} \oplus [\![\mathsf{y}]\!]^{\mathbf{G}}$.

Figure 12: Boolean to Garbled Sharing.

d) Arithmetic to Garbled Sharing (A2G): Similar to Π_{B2G}, $\mathsf{v} = (\mathsf{m_v} - \lambda_{\mathsf{v},1}) - (\lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3})$ and the parties can verifiably generate the garbled shares of $\mathsf{x} = (\mathsf{m_v} - \lambda_{\mathsf{v},1})$ and $\mathsf{y} = (\lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3})$ using $\Pi^{\mathbf{G}}_{\mathsf{vSh}}$. In the online phase, the parties evaluate a garbled subtractor circuit to obtain the shares of $\mathsf{v} = \mathsf{x} - \mathsf{y}$.

Protocol Π_{A2G}

Offline:

- P_0, P_1 execute $\Pi^{\mathbf{G}}_{\mathsf{vSh}}(P_1, P_0, \mathsf{y})$ to generate $[\![\mathsf{y}]\!]^{\mathbf{G}}$ where $\mathsf{y} = \lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3}$.
- P_1, P_2, P_3 garble a subtractor circuit $\mathsf{Sub}(\mathsf{x}, \mathsf{y})$ that computes $\mathsf{x} - \mathsf{y}$. P_1 sends the garbled circuit to P_0, while P_2 sends a hash of the same to P_0, who `abort` if the received values are inconsistent.

Online:

- P_2, P_3 execute $\Pi^{\mathbf{G}}_{\mathsf{vSh}}(P_2, P_3, \mathsf{x})$ to generate $[\![\mathsf{x}]\!]^{\mathbf{G}}$ where $\mathsf{x} = \mathsf{m_v} - \lambda_{\mathsf{v},1}$.
- Parties compute $[\![\mathsf{v}]\!]^{\mathbf{G}} = [\![\mathsf{x}]\!]^{\mathbf{G}} - [\![\mathsf{y}]\!]^{\mathbf{G}}$ by evaluating circuit Sub.

Figure 13: Arithmetic to Garbled Sharing.

e) Arithmetic to Boolean Sharing (A2B): This conversion proceeds similarly to Π_{A2G}, the only difference being parties now generate boolean shares of $\mathsf{x} = (\mathsf{m_v} - \lambda_{\mathsf{v},1})$ and $\mathsf{y} = (\lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3})$ and evaluate a boolean subtractor circuit instead to compute boolean shares of $\mathsf{v} = \mathsf{x} - \mathsf{y}$.

Protocol Π_{A2B}

Offline: P_0, P_1 execute $\Pi^{\mathbf{B}}_{\mathsf{vSh}}(P_1, P_0, \mathsf{y})$ to generate $[\![\mathsf{y}]\!]^{\mathbf{B}}$ where $\mathsf{y} = \lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3}$.

Online:

- P_2, P_3 execute $\Pi^{\mathbf{B}}_{\mathsf{vSh}}(P_2, P_3, \mathsf{x})$ to generate $[\![\mathsf{x}]\!]^{\mathbf{B}}$ where $\mathsf{x} = \mathsf{m_v} - \lambda_{\mathsf{v},1}$.
- Parties compute $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{x}]\!]^{\mathbf{B}} - [\![\mathsf{y}]\!]^{\mathbf{B}}$ by evaluating an ℓ-bit Boolean subtractor circuit Sub.

Figure 14: Arithmetic to Boolean Sharing.

f) Bit to Arithmetic Sharing (Bit2A): Let u and v denote the bits λ_{b} and $\mathsf{m_b}$ respectively over the ring $\mathbb{Z}_{2^\ell}$. Then,

$$\mathsf{b} = \mathsf{m_b} \oplus \lambda_{\mathsf{b}} = \mathsf{v} + \mathsf{u} - 2\mathsf{vu}$$

Party P_0 generates $\langle\cdot\rangle$-shares of u in the offline phase. To ensure the correctness of the shares, parties P_1, P_2, P_3 check whether the following equation holds – $(\lambda_{\mathsf{b}} \oplus \mathsf{r}_b)' = \mathsf{u} + \mathsf{r}'_b - 2\mathsf{u}\mathsf{r}'_b$ where the superscript $(')$ denotes the corresponding bits over ring $\mathbb{Z}_{2^\ell}$. After the verification, parties locally convert $\langle\mathsf{u}\rangle$ to $[\![\mathsf{u}]\!]$. In the online phase, parties multiply $[\![\mathsf{u}]\!], [\![\mathsf{v}]\!]$ to generate $[\![\mathsf{uv}]\!]$ followed by locally computing $[\![\mathsf{b}]\!] = [\![\mathsf{v}]\!] + [\![\mathsf{u}]\!] - 2[\![\mathsf{uv}]\!]$. Note that since λ_{v} is set to 0 while executing $\Pi_{\mathsf{Sh}}(P_1, P_2, P_3, \mathsf{v})$ protocol, γ_{uv}-sharing is not needed during multiplication.

Protocol Π_{Bit2A}

Offline: Let u and $\lambda_{\mathsf{u},i}$ denote the bits λ_{b} and $\lambda_{\mathsf{b},i}$ respectively over the ring $\mathbb{Z}_{2^\ell}$. Here $i \in \{1, 2, 3\}$.

- P_0 executes $\Pi_{\mathsf{aSh}}(P_0, \mathsf{u})$ (Fig. 2) to generate $\langle\mathsf{u}\rangle$. Let the shares be $\langle\mathsf{u}\rangle_{P_1} = (\mathsf{u}_2, \mathsf{u}_3)$, $\langle\mathsf{u}\rangle_{P_2} = (\mathsf{u}_3, \mathsf{u}_1)$, and $\langle\mathsf{u}\rangle_{P_3} = (\mathsf{u}_1, \mathsf{u}_2)$.
- P_1, P_2, P_3 performs the following check:
  - P_1, P_2 sample a random ring element r and a random bit r_b. Let r'_b denotes the bit r_b over ring $\mathbb{Z}_{2^\ell}$.
  - P_1 computes $\mathsf{x}_1 = \lambda_{\mathsf{b},3} \oplus \mathsf{r}_b, \mathsf{y}_1 = (\mathsf{u}_2 + \mathsf{u}_3)(1 - 2\mathsf{r}'_b) + \mathsf{r}'_b + \mathsf{r}$ and sends $(\mathsf{x}_1, \mathsf{y}_1)$ to P_3.
  - P_2 computes $\mathsf{y}_2 = \mathsf{u}_1(1 - 2\mathsf{r}'_b) - \mathsf{r}$ and sends $\mathsf{H}(\mathsf{y}_2)$ to P_3.
  - P_3 computes $\mathsf{x} = \lambda_{\mathsf{b}} \oplus \mathsf{r}_b = \mathsf{x}_1 \oplus \lambda_{\mathsf{b},1} \oplus \lambda_{\mathsf{b},2}$ and `abort` if $\mathsf{H}(\mathsf{x}' - \mathsf{y}_1) \neq \mathsf{H}(\mathsf{y}_2)$. Here x' denotes the bit x over ring $\mathbb{Z}_{2^\ell}$.
- If the verification succeeds, P_1, P_2, P_3 converts $\langle\mathsf{u}\rangle$ to $[\![\mathsf{u}]\!]$ locally by setting $\mathsf{m_u} = 0$ and $\langle\lambda_{\mathsf{u}}\rangle = -\langle\mathsf{u}\rangle$.

Online: Let v denotes the bit $\mathsf{m_b}$ over ring $\mathbb{Z}_{2^\ell}$.

- Parties execute $\Pi_{\mathsf{vSh}}(P_1, P_2, P_3, \mathsf{v})$ to generate $[\![\mathsf{v}]\!]$.
- Parties execute Π_{Mult} on $[\![\mathsf{u}]\!]$ and $[\![\mathsf{v}]\!]$ to generate $[\![\mathsf{uv}]\!]$.
- Parties locally compute $[\![\mathsf{b}]\!] = [\![\mathsf{v}]\!] + [\![\mathsf{u}]\!] - 2[\![\mathsf{uv}]\!]$.

Figure 15: Bit to Arithmetic Sharing.

g) Boolean to Arithmetic Sharing (B2A): We use the fact that a value v can be expressed as $\sum_{i=0}^{\ell-1} 2^i \cdot \mathsf{v}_i$, where v_i denotes the ith bit of v over a ring $\mathbb{Z}_{2^\ell}$. Note that

$$\mathsf{v} = \sum_{i=0}^{\ell-1} 2^i \cdot \mathsf{v}_i = \sum_{i=0}^{\ell-1} 2^i \cdot (\mathsf{m}'_{\mathsf{v}_i} + \lambda'_{\mathsf{u}_i} - 2\mathsf{m}'_{\mathsf{v}_i} \cdot \lambda'_{\mathsf{u}_i})$$

where $\mathsf{m}'_{\mathsf{v}_i}$ and λ'_{u_i} denote the bits $\mathsf{m}_{\mathsf{v}_i}$ and λ_{u_i} respectively over the ring $\mathbb{Z}_{2^\ell}$.

The offline phase of Π_{B2A} proceeds similar to Π_{Bit2A}, where each bit λ_{v_i} for $i \in \{0, \ldots, \ell - 1\}$ is converted to $\langle\cdot\rangle$-share. During the online phase, parties locally compute $[\cdot]$-shares of v followed by generating $[\![\cdot]\!]$-shares of it by executing Π_{vSh} protocol. Parties then locally add their shares to obtain $[\![\mathsf{v}]\!]$.

Protocol Π_{B2A}

Offline: Let v_i denotes the ith bit of value v. Let p_i and $\lambda_{\mathsf{p}_i,j}$ denote the bits λ_{v_i} and $\lambda_{\mathsf{v}_i,j}$ respectively over the ring $\mathbb{Z}_{2^\ell}$, where $i \in \{0, \ldots, \ell - 1\}$ and $j \in \{1, 2, 3\}$.

- Parties execute offline steps of Π_{Bit2A} on each p_i for $i \in \{0, \ldots, \ell - 1\}$, to generate $\langle\mathsf{p}_i\rangle$. Let the shares be $\langle\mathsf{p}_i\rangle_{P_1} = (\mathsf{p}_{i,2}, \mathsf{p}_{i,3})$, $\langle\mathsf{p}_i\rangle_{P_2} = (\mathsf{p}_{i,3}, \mathsf{p}_{i,1})$, and $\langle\mathsf{p}_i\rangle_{P_3} = (\mathsf{p}_{i,1}, \mathsf{p}_{i,2})$.

Online: Let q_i for $i \in \{0, \ldots, \ell-1\}$ denotes bit $\mathsf{m}_{\mathsf{v}_i}$ over $\mathbb{Z}_{2^\ell}$.

- Parties compute the following:
  - P_1, P_3 compute $\mathsf{x} = \sum_{i=0}^{\ell-1} 2^i(\mathsf{q}_i + \mathsf{p}_{i,2} - 2\mathsf{q}_i \cdot \mathsf{p}_{i,2})$.
  - P_2, P_1 compute $\mathsf{y} = \sum_{i=0}^{\ell-1} 2^i(\mathsf{p}_{i,3} - 2\mathsf{q}_i \cdot \mathsf{p}_{i,3})$.
  - P_3, P_2 compute $\mathsf{z} = \sum_{i=0}^{\ell-1} 2^i(\mathsf{p}_{i,1} - 2\mathsf{q}_i \cdot \mathsf{p}_{i,1})$.
- Parties generate $[\![\mathsf{x}]\!], [\![\mathsf{y}]\!]$ and $[\![\mathsf{z}]\!]$ by executing $\Pi_{\mathsf{vSh}}(P_1, P_3, \mathsf{x})$, $\Pi_{\mathsf{vSh}}(P_2, P_1, \mathsf{y})$ and $\Pi_{\mathsf{vSh}}(P_3, P_2, \mathsf{z})$ respectively.
- Parties locally compute $[\![\mathsf{v}]\!] = [\![\mathsf{x}]\!] + [\![\mathsf{y}]\!] + [\![\mathsf{z}]\!]$.

Figure 16: Boolean to Arithmetic Sharing V2.

h) Bit Injection (BitInj): $[\![b]\!]^{\mathbf{B}}[\![\mathsf{v}]\!] \to [\![b\mathsf{v}]\!]$*:* Let y_1 and y_2 denote the values λ_{b} and $\lambda_{\mathsf{b}}\lambda_{\mathsf{v}}$ respectively over ring $\mathbb{Z}_{2^\ell}$. Similarly, let $\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2$ and x_3 denote the values $(\mathsf{m}_{\mathsf{b}}\mathsf{m}_{\mathsf{v}})$, $(\mathsf{m}_{\mathsf{b}})$, $(\mathsf{m}_{\mathsf{v}} - 2\mathsf{m}_{\mathsf{v}}\mathsf{m}_{\mathsf{b}})$ and $(2\mathsf{m}_{\mathsf{b}} - 1)$ respectively over ring $\mathbb{Z}_{2^\ell}$. Then,

$$\mathsf{b} \cdot \mathsf{v} = (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) = \mathsf{x}_0 - \mathsf{x}_1\mathsf{y}_1 + \mathsf{x}_2\mathsf{y}_2 + \mathsf{x}_3\mathsf{y}_3$$

Protocol Π_{BitInj}

Offline: Let y_1 and y_2 denote the values λ_{b} and $\lambda_{\mathsf{b}}\lambda_{\mathsf{v}}$ respectively over $\mathbb{Z}_{2^\ell}$.

- P_0 executes $\Pi_{\mathsf{aSh}}(P_0, \mathsf{y}_j)$ to generate $\langle \mathsf{y}_j \rangle$ for $j \in \{1, 2\}$. Let the shares be $\langle \mathsf{y}_j \rangle_{P_1} = (\mathsf{y}_{j,2}, \mathsf{y}_{j,3})$, $\langle \mathsf{y}_j \rangle_{P_2} = (\mathsf{y}_{j,3}, \mathsf{y}_{j,1})$, and $\langle \mathsf{y}_j \rangle_{P_3} = (\mathsf{y}_{j,1}, \mathsf{y}_{j,2})$.
- Parties verify the correctness of $\langle \mathsf{y}_1 \rangle$ using the steps similar to protocol Π_{Bit2A} (Fig. 15). To verify the correctness of $\langle \mathsf{y}_2 \rangle$, parties proceed as follows:
  - Parties execute Π_{Zero} (Fig. 22) to generate A, B, Γ such that $A + B + \Gamma = 0$.
  - P_1 computes $\mathsf{u}_2 = \lambda_{\mathsf{y}_1,2}\lambda_{\mathsf{v},2} + \lambda_{\mathsf{y}_1,2}\lambda_{\mathsf{v},3} + \lambda_{\mathsf{y}_1,3}\lambda_{\mathsf{v},2} + A$.
  - P_2 computes $\mathsf{u}_3 = \lambda_{\mathsf{y}_1,3}\lambda_{\mathsf{v},3} + \lambda_{\mathsf{y}_1,3}\lambda_{\mathsf{v},1} + \lambda_{\mathsf{y}_1,1}\lambda_{\mathsf{v},3} + B$.
  - P_3 computes $\mathsf{u}_1 = \lambda_{\mathsf{y}_1,1}\lambda_{\mathsf{v},1} + \lambda_{\mathsf{y}_1,1}\lambda_{\mathsf{v},2} + \lambda_{\mathsf{y}_1,2}\lambda_{\mathsf{v},1} + \Gamma$.
  - P_1 and P_2 send z_2 and $\mathsf{H}(-\mathsf{z}_3)$ respectively to P_3, where $\mathsf{z}_2 = \mathsf{u}_2 - \mathsf{y}_{2,2}$ and $\mathsf{z}_3 = \mathsf{u}_3 - \mathsf{y}_{2,3}$.
  - P_3 sets $\mathsf{z}_1 = \mathsf{u}_1 - \mathsf{y}_{2,1}$ and `abort` if $\mathsf{H}(\mathsf{z}_1 + \mathsf{z}_2) \neq \mathsf{H}(-\mathsf{z}_3)$.

Online: Let $\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2$ and x_3 denote the values $(\mathsf{m}_{\mathsf{b}}\mathsf{m}_{\mathsf{v}})$, $(\mathsf{m}_{\mathsf{b}})$, $(\mathsf{m}_{\mathsf{v}} - 2\mathsf{m}_{\mathsf{v}}\mathsf{m}_{\mathsf{b}})$ and $(2\mathsf{m}_{\mathsf{b}} - 1)$ respectively over ring $\mathbb{Z}_{2^\ell}$.

- Parties compute the following:
  - P_1, P_3 compute $\mathsf{c}_2 = \mathsf{x}_0 - \mathsf{x}_1\lambda_{\mathsf{v},2} + \mathsf{x}_2\mathsf{y}_{1,2} + \mathsf{x}_3\mathsf{y}_{2,2}$.
  - P_2, P_1 compute $\mathsf{c}_3 = -\mathsf{x}_1\lambda_{\mathsf{v},3} + \mathsf{x}_2\mathsf{y}_{1,3} + \mathsf{x}_3\mathsf{y}_{2,3}$.
  - P_3, P_2 compute $\mathsf{c}_1 = -\mathsf{x}_1\lambda_{\mathsf{v},1} + \mathsf{x}_2\mathsf{y}_{1,1} + \mathsf{x}_3\mathsf{y}_{2,1}$.
- Parties execute $\Pi_{\mathsf{vSh}}(P_1, P_3, \mathsf{c}_2)$, $\Pi_{\mathsf{vSh}}(P_2, P_1, \mathsf{c}_3)$ and $\Pi_{\mathsf{vSh}}(P_3, P_2, \mathsf{c}_1)$ to generate $[\![\mathsf{c}_2]\!]$, $[\![\mathsf{c}_3]\!]$ and $[\![\mathsf{c}_1]\!]$ respectively.
- Parties locally compute $[\![b\mathsf{v}]\!] = [\![\mathsf{c}_1]\!] + [\![\mathsf{c}_2]\!] + [\![\mathsf{c}_3]\!]$.

Figure 17: Bit Injection: $[\![\mathsf{b}]\!]^{\mathbf{B}}[\![\mathsf{v}]\!]^{\mathbf{A}} \to [\![\mathsf{bv}]\!]^{\mathbf{A}}$.

In the offline phase, P_0 generates $\langle\cdot\rangle$-shares of λ'_{b} and $\lambda_{\mathsf{b}}\lambda_{\mathsf{v}}$ where λ'_{b} denotes the bit λ_{b} over $\mathbb{Z}_{2^\ell}$. The check for $\langle \lambda'_{\mathsf{b}} \rangle$ is the same as the one for Π_{Bit2A}, to check $\langle \lambda_{\mathsf{b}}\lambda_{\mathsf{v}} \rangle$ parties proceed as mentioned in protocol Π_{BitInj} above. During the online phase, parties locally compute $[\cdot]$-shares of $\mathsf{b}\cdot\mathsf{v}$ followed by generating $[\![\cdot]\!]$-shares of it by executing Π_{vSh} protocol. Parties then locally add their shares to obtain $[\![\mathsf{b} \cdot \mathsf{v}]\!]$.

V. Privacy Preserving Machine Learning

Most of the intermediate values in machine learning algorithms involve operating over decimals. To represent decimal values, we use signed two's compliment over $\mathbb{Z}_{2^\ell}$ [2], [5], [25], where the most significant bit (msb) represents the sign and the last d bits represent the fractional part.

In order to perform privacy-preserving machine learning, we need efficient instantiations of three components – Share Truncation, Secure Comparison, and Non-linear Activation Functions. This section covers our protocols for performing the aforementioned components.

A. Share Truncation

We take inspiration for truncation from ABY3 [5], where they perform it on shares after evaluating a multiplication gate, preserving the underlying value with very high probability. Our approach improves upon ABY3 by not using any boolean circuits, thus improving the offline round complexity to constant.

Protocol $\Pi_{\mathsf{MultTr}}(\mathsf{w}_{\mathsf{x}}, \mathsf{w}_{\mathsf{y}}, \mathsf{w}_{\mathsf{z}})$

Offline:

- Parties execute the offline steps of protocol $\Pi_{\mathsf{Mult}}(\mathsf{w}_{\mathsf{x}}, \mathsf{w}_{\mathsf{y}}, \mathsf{w}_{\mathsf{z}})$ apart from λ_{z} not being generated.
- Parties locally sample the following random values:

$$\mathcal{P} \setminus \{P_2\} : \mathsf{r}_2, \qquad \mathcal{P} \setminus \{P_1\} : \mathsf{r}_1, \qquad \mathcal{P} \setminus \{P_3\} : \mathsf{r}_3$$

- P_0 locally compute $\mathsf{r} = \mathsf{r}_1 + \mathsf{r}_2 + \mathsf{r}_3$, locally truncates it to obtain r^{t} and executes $\Pi_{\mathsf{aSh}}(P_0, \mathsf{r}^{\mathsf{t}})$ to generate $\langle \mathsf{r}^{\mathsf{t}} \rangle$. Let the shares be $\langle \mathsf{r}^{\mathsf{t}} \rangle_{P_1} = (\mathsf{r}^{\mathsf{t}}_2, \mathsf{r}^{\mathsf{t}}_3)$, $\langle \mathsf{r}^{\mathsf{t}} \rangle_{P_2} = (\mathsf{r}^{\mathsf{t}}_3, \mathsf{r}^{\mathsf{t}}_1)$, and $\langle \mathsf{r}^{\mathsf{t}} \rangle_{P_3} = (\mathsf{r}^{\mathsf{t}}_1, \mathsf{r}^{\mathsf{t}}_2)$.
- Let r_{d} and $\mathsf{r}_{\mathsf{d},i}$ denote the last d bits of r and r_i respectively for $i \in \{1, 2, 3\}$. Parties verify the correctness of $\langle \mathsf{r}^{\mathsf{t}} \rangle$ as follows:
  - P_1 samples a random element c and computes $\mathsf{m}_1 = \mathsf{r}_2 - 2^{\mathsf{d}}\mathsf{r}^{\mathsf{t}}_2 - \mathsf{r}_{\mathsf{d},2} + \mathsf{c}$. P_1 sends $(\mathsf{m}_1, \mathsf{H}(\mathsf{c}))$ to P_2.
  - P_2 computes $\mathsf{m}_2 = (\mathsf{r}_1 + \mathsf{r}_3) - 2^{\mathsf{d}}(\mathsf{r}^{\mathsf{t}}_1 + \mathsf{r}^{\mathsf{t}}_3) - (\mathsf{r}_{\mathsf{d},1} + \mathsf{r}_{\mathsf{d},3})$ and `abort` if $\mathsf{H}(\mathsf{m}_1 + \mathsf{m}_2) \neq \mathsf{H}(\mathsf{c})$.
- Parties locally convert $\langle \mathsf{r}^{\mathsf{t}} \rangle$ to $[\![\mathsf{r}^{\mathsf{t}}]\!]$ by setting $\mathsf{m}_{\mathsf{r}^{\mathsf{t}}} = 0$ and $\langle \lambda_{\mathsf{r}^{\mathsf{t}}} \rangle = \langle \mathsf{r}^{\mathsf{t}} \rangle$.

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

- Parties locally compute the following:
  - P_1, P_3 compute $[\mathsf{z}']_2 = -\lambda_{\mathsf{x},2}\mathsf{m}_{\mathsf{y}} - \lambda_{\mathsf{y},2}\mathsf{m}_{\mathsf{x}} + \gamma_{\mathsf{xy},2} - \mathsf{r}_2$.
  - P_2, P_1 compute $[\mathsf{z}']_3 = -\lambda_{\mathsf{x},3}\mathsf{m}_{\mathsf{y}} - \lambda_{\mathsf{y},3}\mathsf{m}_{\mathsf{x}} + \gamma_{\mathsf{xy},3} - \mathsf{r}_3$.
  - P_3, P_2 compute $[\mathsf{z}']_1 = -\lambda_{\mathsf{x},1}\mathsf{m}_{\mathsf{y}} - \lambda_{\mathsf{y},1}\mathsf{m}_{\mathsf{x}} + \gamma_{\mathsf{xy},1} - \mathsf{r}_1$.
- Parties exchange the following:
  - P_1 receives $[\mathsf{z}']_1$ and $\mathsf{H}([\mathsf{z}']_1)$ from P_2 and P_3 respectively.
  - P_2 receives $[\mathsf{z}']_2$ and $\mathsf{H}([\mathsf{z}']_2)$ from P_3 and P_1 respectively.
  - P_3 receives $[\mathsf{z}']_3$ and $\mathsf{H}([\mathsf{z}']_3)$ from P_1 and P_2 respectively.
- P_i for $i \in \{1, 2, 3\}$ `abort` if the received values are inconsistent. Else, he computes $(\mathsf{z} - \mathsf{r}) = [\mathsf{z}']_1 + [\mathsf{z}']_2 + [\mathsf{z}']_3 + \mathsf{m}_{\mathsf{x}}\mathsf{m}_{\mathsf{y}}$.
- P_1, P_2, P_3 locally truncates $(\mathsf{z} - \mathsf{r})$ to obtain $(\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, followed by executing $\Pi_{\mathsf{vSh}}(P_1, P_2, P_3, (\mathsf{z} - \mathsf{r})^{\mathsf{t}})$ to generate $[\![(\mathsf{z} - \mathsf{r})^{\mathsf{t}}]\!]$.
- Parties locally compute $[\![\mathsf{z}^{\mathsf{t}}]\!] = [\![(\mathsf{z} - \mathsf{r})^{\mathsf{t}}]\!] + [\![\mathsf{r}^{\mathsf{t}}]\!]$.

Figure 18: Multiplication with Truncation.

We start by generating a random $(\mathsf{r}, \mathsf{r}^{\mathsf{t}})$ in the offline phase, where r^{t} is the truncated value of r. The truncated value of z can be obtained by first opening and truncating $\mathsf{z} - r$, and then adding it to r^{t}. Parties non-interactively generate $\langle \mathsf{r} \rangle$, such that P_0 obtains r. This is followed by P_0 generating $[\![\mathsf{r}^{\mathsf{t}}]\!]$, but since we cannot rely on P_0, parties P_1, P_2, P_3 perform a check to ensure the correctness of the share. On a high level, parties check the relation $\mathsf{r} = 2^{\mathsf{d}}\mathsf{r}^{\mathsf{t}} + \mathsf{r}_{\mathsf{d}}$, where r_{d} denotes the last d bits of r The formal details of the check are provided in the protocol above and the correctness appears in Lemma D.1.

B. Secure Comparison

The secure comparison technique allows parties to check whether $\mathsf{x} < \mathsf{y}$, given arithmetic shares of x, y. In fixed point

arithmetic, a simple way to achieve this is by computing $\mathsf{x}-\mathsf{y}$, and checking its sign, stored in the msb position. This protocol, inspired from ASTRA [25], is called Bit Extraction (Π_{BitExt}), since it extracts a bit from the given arithmetic shares and outputs the boolean shares of the bit.

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

Offline:

- P_1, P_2 sample random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and set $\mathsf{x} = \mathsf{msb}(\mathsf{r})$.
- Parties execute $\Pi_{\mathsf{vSh}}(P_1, P_2, \mathsf{r})$ and $\Pi^{\mathbf{B}}_{\mathsf{vSh}}(P_1, P_2, \mathsf{x})$ to generate $[\![\mathsf{r}]\!]$ and $[\![\mathsf{x}]\!]^{\mathbf{B}}$ respectively.

Online:

- Parties execute Π_{Mult} on $[\![\mathsf{r}]\!]$ and $[\![\mathsf{v}]\!]$ to generate $[\![\mathsf{rv}]\!]$, followed by reconstructing rv towards P_0 and P_3.
- P_3, P_0 set $\mathsf{y} = \mathsf{msb}(\mathsf{rv})$ followed by executing $\Pi^{\mathbf{B}}_{\mathsf{vSh}}(P_3, P_0, \mathsf{y})$ to generate $[\![\mathsf{y}]\!]^{\mathbf{B}}$.
- Parties locally compute $[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}} = [\![\mathsf{x}]\!]^{\mathbf{B}} \oplus [\![\mathsf{y}]\!]^{\mathbf{B}}$.

Figure 19: Extraction of MSB bit of value v.

C. Activation Functions

a) ReLU: The ReLU function is defined as $\mathsf{relu}(\mathsf{v}) = \mathsf{max}(0, \mathsf{v})$. This can be viewed as $\mathsf{relu}(\mathsf{v}) = (1 \oplus \mathsf{b})\mathsf{v}$ where bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. In order to generate $[\![\mathsf{relu}(\mathsf{v})]\!]$, parties first execute Π_{BitExt} on v to obtain $[\![\mathsf{b}]\!]^{\mathbf{B}}$ and locally compute $[\![1 \oplus \mathsf{b}]\!]^{\mathbf{B}}$. This is followed by executing Π_{BitInj} on $[\![1 \oplus \mathsf{b}]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$. The derivative of relu, denoted by $\mathsf{drelu}(\mathsf{v}) = (1 \oplus \mathsf{b})$.

b) Sigmoid: In our protocols, we use the approximation of sigmoid function [2], [5], [25], defined as:

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

This can be viewed as, $\mathsf{sig}(\mathsf{v}) = (1 \oplus \mathsf{b}_1)\mathsf{b}_2(\mathsf{v}+1/2) + (1 \oplus \mathsf{b}_2)$, where $\mathsf{b}_1 = 1$ if $\mathsf{v} + 1/2 < 0$ and $\mathsf{b}_2 = 1$ if $\mathsf{v} - 1/2 < 0$. The protocol is similar to that of relu apart from an additional bit extraction, bit multiplication and a bit injection is required.

VI. Implementation and Benchmarking

The improvements of our framework over the current state-of-the-art (ABY3) are showcased through our implementation, comparing the two. The training and prediction phases of Linear Regression, Logistic Regression, Neural Networks (NN), and Convolutional Neural Networks (CNN) are used for benchmarking. While we compare our construction with the malicious ABY3 in this section, the comparison of ours with the semi-honest version of ABY3 and with the 4PC protocol of Gordon et al. [26] are deferred to Appendix E.

a) Environment Details: We provide results for both LAN (1Gbps bandwidth) and WAN (40Mbps bandwidth) settings. In the LAN setting, we have machines with 3.6 GHz Intel Core i7-7700 CPU and 32 GB RAM. In the WAN setting, we use Google Cloud Platform[2] with machines located in West Europe (P_0), East Australia (P_1), South Asia (P_2) and South East Asia (P_3). We use n1-standard-8 instances, where machines are equipped with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors supporting hyper-threading, with 8 vCPUs, and 30 GB RAM. We measured the average round-trip time (rtt) for communicating 1 KB of data between every pair of parties. Over the LAN setting, the rtt turned out to be $0.296ms$. In the WAN setting, the rtt values were

P_0-P_1	P_0-P_2	P_0-P_3	P_1-P_2	P_1-P_3	P_2-P_3
$274.83ms$	$174.13ms$	$219.45ms$	$152.3ms$	$60.19ms$	$92.63ms$

We implement our protocols using the ENCRYPTO library [50] in C++17 over a 64 bit ring. Since the codes for ABY3 and MRZ [16] are not publicly available, we implement their protocols in our environment. The hash function is instantiated using SHA-256. We use multi-threading, wherever possible, to facilitate efficient computation and communication among the parties. To even out the results, each experiment is run 20 times and the average values are reported.

b) Datasets: To benchmark the training phase of machine learning algorithms, it is common practice to use synthetic datasets so that we have freedom to choose the parameters of the datasets. However, testing the accuracy of a trained model must be carried out with a real dataset, which is the MNIST [43] in our case. It contains 28×28 pixel images of handwritten numbers and 784 features. For benchmarking of the prediction phase, we use the following real-world datasets:

Dataset	#features	#samples
Candy (CD) Power Ranking [51]	13	85
Boston (BT) Housing Prices [52]	14	506
Weather (WR) Conditions in World War Two [53]	31	≈119000
CalCOFI (CI) - Oceanographic Data [54]	74	≈876000
Epileptic (EP) Seizures [55]	179	≈11500
Food **Recipes (RE)** [56]	680	≈20000
MNIST [43]	784	70000

We choose Boston, Weather and CalCOFI for linear regression, since they are best suited for it, while Candy, Epileptic and Recipes were chosen for logistic regression. For NN and CNN, we used the MNIST dataset.

A. Secure Training

The training phase in most of the machine learning algorithms consists of two stages– i) forward propagation, where the model computes the output, and ii) backward propagation, where the model parameters are adjusted according to the computed output and the actual output. We define one *iteration* in the training phase as one forward propagation followed by a backward propagation.

This section covers the improvements in the training phase of our protocol as compared to ABY3. We report the performance in terms of the number of iterations over varying feature (d) and batch sizes (B), where $d \in \{10, 100, 1000\}$ and $B \in \{128, 256, 512\}$. In LAN, we use *iterations per second* (#it/sec) as the metric, but since rtt is much higher for WAN, we instead use *iterations per minute* (#it/min).

a) **Linear Regression:** For linear regression, one iteration can be viewed as updating the weight vector $\vec{\mathbf{w}}$ using the

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

Gradient Descent algorithm (GD). The update function for $\vec{\mathbf{w}}$ is given by

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

where α denotes the learning rate and $\mathbf{X}_i$ denotes a subset of batch size B, randomly selected from the entire dataset in the ith iteration. Here the forward propagation consists of computing $\mathbf{X}_i \circ \vec{\mathbf{w}}$, while the weight vector is updated in the backward propagation. The update function consists of a series of matrix multiplications, which in turn can be achieved using dot product protocols. The operations of subtraction as well as multiplication by a public constant can be performed locally. We observe that the aforementioned update function can be computed entirely in the arithmetic domain and can be viewed in form of $[\![\cdot]\!]$-shares as

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

Network	#features	Ref.	Batch Size B		
			128	256	512
LAN #it/sec	10	ABY3	287.36	246.92	186.22
		This	1639.35	1204.82	1162.8
	100	ABY3	110.38	62.08	29.27
		This	1587.31	1176.48	1136.37
	1000	ABY3	13.51	6.88	3.43
		This	1095.3	883.4	861.33
WAN #it/min	10	ABY3	97.57	97.57	97.57
		This	195.14	195.14	195.14
	100	ABY3	97.57	97.11	95.08
		This	195.14	195.14	195.14
	1000	ABY3	89.95	80.10	68.94
		This	195.14	195.14	195.14

Table IV: Comparison of ABY3 (Malicious) and **This** for Linear Regression (higher = better).

Table IV provides concrete values for Linear Regression. Our improvement over LAN ranges from $4.88\times$ to $251.84\times$ and $2\times$ to $2.83\times$ over WAN. The gain comes due to two factors: One being the amount we save through our feature-independent communication of the dot product protocol (3 ring elements as opposed to $9d$). The other factor is our efficient truncation protocol, which reduces the online communication from 12 elements to 3 elements – by 75%. The reason for the discrepancy in gains in LAN and WAN is because in LAN, the rtt is in the order of microseconds, and scales with the communication size. In contrast, the rtt in WAN is in the order of milliseconds and does not scale with communication up to a threshold, within which all our protocols operate.

b) **Logistic Regression:** The iteration for the case of logistic regression is similar to that of linear regression, apart from an activation function being applied on $\mathbf{X}_i \circ \vec{\mathbf{w}}$ in the forward propagation. We instantiate the activation function using sigmoid (Section V-C). The update function for $\vec{\mathbf{w}}$ is given by

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

One iteration of logistic regression incurs an additional cost for computing $\mathsf{sig}(\mathbf{X}_j \circ \vec{\mathbf{w}})$ as compared with that for linear regression.

Network	#features	Ref.	Batch Size B		
			128	256	512
LAN #it/sec	10	ABY3	56.95	42.02	30.35
		This	338.99	257.01	226.61
	100	ABY3	43.34	27.89	16.2
		This	336.71	255.69	225.64
	1000	ABY3	11.36	6.06	3.13
		This	307.41	238.44	212.23
WAN #it/min	10	ABY3	20.54	20.54	20.52
		This	55.76	55.76	55.76
	100	ABY3	20.54	20.52	20.41
		This	55.76	55.76	55.76
	1000	ABY3	20.18	19.64	18.87
		This	55.76	55.76	55.76

Table V: Comparison of ABY3 (Malicious) and **This** for Logistic Regression (higher = better).

Table V provides concrete values for Logistic Regression. Logistic Regression can be thought of as an execution of Linear Regression followed by a Sigmoid function on the output, due to which our improvements for Linear Regression carry over to Logistic. Our improvement ranges from $5.95\times$ to $67.88\times$ over LAN and $2.71\times$ to $2.96\times$ over WAN. Our efficient Sigmoid protocol takes this result further by improving upon the round and communication complexity. The round complexity is brought down to constant from $4 + \log \ell$ to 5. Instantiated over ring $\mathbb{Z}_{2^{64}}$, this amounts to an improvement of 50%. The communication is also improved by $\approx 80\%$ (from 81 elements to roughly 16).

c) **Neural Networks:** For the case of NN, we follow steps similar to that of ABY3, where each node across all the layers, except the last layer, uses ReLU (relu) as the activation function. At the output layer, we use the MPC friendly variant of the softmax activation function, $\mathsf{smx}(\mathsf{u_i}) = \frac{\mathsf{relu}(\mathsf{u_i})}{\sum_{\mathsf{j}=1}^{\mathsf{nf}} \mathsf{relu}(\mathsf{u_j})}$, proposed by SecureML [2]. In order to perform the division, we switch from arithmetic to garbled world and then use a division garbled circuit.

The network is trained using the Gradient Descent, where the forward propagation comprises of computing activation matrices for all the layers in the network. Here, the activation matrix for all the layers except the output, is defined as $\mathbf{A}_i = \mathsf{relu}(\mathbf{U}_i)$, where $\mathbf{U}_i = \mathbf{A}_{i-1} \circ \mathbf{W}_i$. $\mathbf{A}_0$ is initialized to $\mathbf{X}_j$, where $\mathbf{X}_j$ is a subset of batch size B, randomly selected from the entire dataset for the j^{th} iteration. The activation matrix for the output layer is defined as $\mathbf{A}_m = \mathsf{smx}(\mathbf{U}_\mathsf{m})$.

During the backward propagation, error matrices are computed first. The error matrix for the output layer is defined as $\mathbf{E}_m = (\mathbf{A}_m - \mathbf{T})$, while for the remaining layers it is defined as $\mathbf{E}_i = (\mathbf{E}_{i+1} \circ \mathbf{W}_i^T) \otimes \mathsf{drelu}(\mathbf{U}_i)$. Here the operation $\otimes$ denotes element wise multiplication and drelu denotes the derivative of ReLU. This is followed by updating the weights as $\mathbf{W}_i = \mathbf{W}_i - \frac{\alpha}{B}\mathbf{A}_{i-1}^T \circ \mathbf{E}_i$.

We consider a NN with two hidden layers, each having 128 nodes followed by an output layer of 10 nodes. After each layer, the activation function ReLU is applied. NN training

Network	Ref.	LAN (#it/sec)			WAN (#it/min)		
		B-128	B-256	B-512	B-128	B-256	B-512
NN	ABY3	0.37	0.25	0.15	4.69	4.28	3.87
	This	23.00	13.55	7.70	13.94	13.94	13.79
CNN	ABY3	0.23	0.18	0.13	4.34	4.05	3.71
	This	10.46	5.63	2.99	13.86	13.67	13.16

Table VI: Comparison of ABY3 (Malicious) and **This** for NN and CNN (higher = better).

involves one forward pass followed by one back-propagation. In LAN, the number of iterations is maximum with a batch of 128, at 22.99 #it/sec, and comes down to 7.70 with a batch size of 512. Similarly, over WAN it is maximum at 13.94 and comes down to 13.79. As expected, the #it/sec has not decreased with increase in features due to our dot product protocol being feature-independent in terms of communication. ABY3 on the other hand, has reported 2.5 #it/sec with a batch size of 128, in the computationally lighter semi-honest setting. Table VI above provides more details.

We also considered a CNN discussed in [4] with 2 hidden layers, consisting of 100 and 10 nodes. Similar to ABY3, we *overestimate* the running time by replacing the convolutional kernel with a fully connected layer. In LAN, the number of iterations is maximum with a batch of 128, at 10.46 #it/sec, and comes down to 2.99 with a batch size of 512. Similarly, over WAN it is maximum at 13.86 and comes down to 13.16.

B. Secure Prediction

For Secure Prediction, we use online latency of the protocol as a metric to compare both works. The units are milliseconds in LAN and seconds in WAN. We use the MNIST dataset, which has 784 features, with a batch size of 1 and 100 for benchmarking. Our truncation protocol causes a bit-error at the least significant bit position, which is the same as that of ABY3 and SecureML [2] due to similarity in the techniques. We refer the readers to SecureML for a detailed analysis of the bit-error. The accuracy of the prediction itself however, ranges from 93% for linear regression to 98.3% for CNN.

Network	Batch Size	Ref.	Linear Regression	Logistic Regression	NN	CNN
LAN (ms)	1	ABY3	2.08	6.25	73.09	371.1
		This	0.25	1.75	4.51	5.4
	100	ABY3	37.80	49.68	1284.95	2010.06
		This	0.30	2.55	17.17	39.63
WAN (s)	1	ABY3	0.47	2.77	6.02	6.25
		This	0.16	0.93	2.31	2.31
	100	ABY3	0.49	2.79	7.04	7.5
		This	0.16	0.93	2.31	2.32

Table VII: Online Runtime of ABY3 (Malicious) and **This** for Secure Prediction of Linear, Logistic, NN, and CNN models for $d = 784$. (lower = better).

For Linear Regression, our improvement ranges from 3× to 126×, considering LAN and WAN together. For Logistic Regression, our improvement ranges from 3× to 19.48×, considering LAN and WAN together. In NN, we achieve an improvement ranging from 3.05× to 74.85×. Similarly for CNN, the improvement ranges from 2.71× to 68.82×.

Though not stated explicitly, our offline cost for linear regression is orders of magnitude more efficient as compared to ABY3. This improvement carries over for logistic regression, NN, and CNN networks as well. A large part of this improvement comes from the difference in the approaches to truncation. ABY3's approach entails using Ripple Carry Adder (RCA) circuits, which consume 128 rounds. ABY3 has pointed out that this was the reason SecureML performed better in total time for a single prediction. Our approach on the other hand, does not use any such circuit, resulting in an improvement of $\approx 15\times$ in communication and $64\times$ in rounds.

Throughput Comparison: We use a different metric to better illustrate the impact of our efficiency in the case of secure prediction, which is online throughput. Online throughput over LAN is the number of predictions that can be made in a second, and over WAN it is the number of predictions per minute. We have a total of 32 threads over 4 CPU cores, wherein each thread can perform 100 queries simultaneously without reduction in performance. Table VIII provides the online throughput comparison of ABY3 and ours for secure prediction over real-world datasets in a LAN setting. The gains for linear regression range from 26.16× to 145.18× and from 5.69× to 158.40× for logistic regression. Similarly, we observed gains of 335.44× and 598.44× for NN and CNN respectively.

Ref.	Linear Regression			Logistic Regression			NN	CNN
	BT	**WR**	**CI**	**CD**	**EP**	**RE**	**MNIST**	
ABY3	4.08	1.74	0.73	2.20	0.29	0.08	0.46	0.06
This	106.67	106.67	106.67	12.55	12.55	12.55	153.39	37.43

Table VIII: Online Throughput Comparison of ABY3 (Malicious) and **This** for Secure Prediction over LAN. (higher = better)

In WAN, even though our protocols are more communication efficient as compared to ABY3, we could not fully capitalize on it especially for Linear Regression and Logistic Regression. This is due to the limitation of being able to run only 32 CPU threads in parallel, which amounts to a lot of bandwidth not being utilized. This gap can be closed by introducing more CPU threads into our infrastructure. However, since we could not do this, in order to showcase the efficiency better, we limit the bandwidth and compute the gain in online throughput. As evident from the plot in Fig. 20, the gain increases as we limit the bandwidth. We observe that our protocols for Linear Regression and Logistic Regression achieve maximum bandwidth utilization at around 1.5 Mbps. On the other hand, NN and CNN utilize the entire bandwidth, even at 40 Mbps. So decreasing the bandwidth does have an effect on the throughput for NN and CNN.

VII. Conclusion

In this work, we presented an efficient privacy-preserving machine learning framework for the four-party setting, tolerating at most one malicious corruption. The theoretical improvements over the state-of-the-art 3PC framework of ABY3 were backed up by an extensive benchmarking. The improvements show that the availability of an additional honest party can improve the performance of ML protocols while at the same time, decreasing the total monetary cost of hiring the servers.

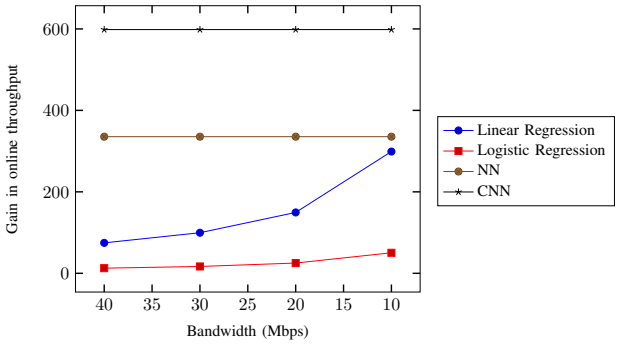

Figure 20: Throughput Gain in Low-end Networks.

An interesting open problem is extending this framework to the n party setting. Improving the security to guaranteed output delivery with a minimal trade-off in the concrete performance is another challenging problem. Another direction would be to try to integrate the protocols proposed in this paper into a compiler such as HyCC [57], which already has support for the two-party framework of ABY.

REFERENCES

[1] A. Madani, R. Arnaout, M. Mofrad, and R. Arnaout, "Fast and accurate classification of echocardiograms using deep learning," *CoRR*, 2017.

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

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

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

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

[6] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: Efficient and private neural network training," *IACR Cryptology ePrint Archive*, vol. 2018.

[7] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *ASIACRYPT*, 2009.

[8] C. Orlandi, "Is multiparty computation any good in practice?" in *IEEE ICASSP*, 2011.

[9] D. W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright, "From keys to databases - real-world applications of secure multi-party computation," *Comput. J.*, 2018.

[10] 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.

[11] 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.

[12] 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.

[13] Y. Lindell and A. Nof, "A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority," in *ACM CCS*, 2017.

[14] 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.

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

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

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

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

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

[20] D. Bogdanov, R. Talviste, and J. Willemson, "Deploying Secure Multi-Party Computation for Financial Data Analysis," in *FC*, 2012.

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

[22] D. Bogdanov, L. Kamm, B. Kubo, R. Rebane, V. Sokk, and R. Talviste, "Students and taxes: a privacy-preserving social study using secure computation," *IACR Cryptology ePrint Archive*, 2015.

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

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

[25] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High-throughput 3PC over Rings with Application to Secure Prediction," in *IACR Cryptology ePrint Archive*, 2019.

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

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

[28] D. Beaver, "Efficient Multiparty Protocols Using Circuit Randomization," in *CRYPTO*, 1991.

[29] D. Beaver, "Precomputing Oblivious Transfer," in *CRYPTO*, 1995.

[30] Z. Beerliová-Trubíniová and M. Hirt, "Efficient Multi-party Computation with Dispute Control," in *TCC*, 2006.

[31] Z. Beerliová-Trubíniová and M. Hirt, "Perfectly-Secure MPC with Linear Communication Complexity," in *TCC*, 2008.

[32] E. Ben-Sasson, S. Fehr, and R. Ostrovsky, "Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority," in *CRYPTO*, 2012.

[33] A. Choudhury and A. Patra, "An Efficient Framework for Unconditionally Secure Multiparty Computation," *IEEE Trans. Information Theory*, 2017.

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

[35] 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.

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

[37] M. Keller, E. Orsini, and P. Scholl, "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer," in *ACM CCS*, 2016.

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

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

[40] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPDZ2k: Efficient MPC mod 2ˆk for Dishonest Majority," *CRYPTO*, 2018.

[41] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *EUROCRYPT*, 2018.

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

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

[44] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[45] V. Kolesnikov, P. Mohassel, and M. Rosulek, "Flexor: Flexible garbling for XOR gates that beats free-xor," in *CRYPTO*, 2014.

[46] S. Zahur, M. Rosulek, and D. Evans, "Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates," in *EUROCRYPT*, 2015.

[47] S. Gueron, Y. Lindell, A. Nof, and B. Pinkas, "Fast Garbling of Circuits Under Standard Assumptions," in *ACM CCS*, 2015.

[48] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient Garbling from a Fixed-Key Blockcipher," in *IEEE S&P*, 2013.

[49] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of Garbled Circuits," in *ACM CCS*, 2012.

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

[51] W. Hickey, "The ultimate halloween candy power ranking," 2017. [Online]. Available: https://www.kaggle.com/fivethirtyeight/the-ultimate-halloween-candy-power-ranking/

[52] D. Harrison and D. L. Rubinfeld, "Hedonic housing prices and the demand for clean air," *Journal of Environmental Economics and Management*, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0095069678900062

[53] NOAA, "Weather conditions in world war two," 2017. [Online]. Available: https://www.kaggle.com/smid80/weatherww2/data

[54] S. Dane, "CalCOFI - over 60 years of oceanographic data," 2017. [Online]. Available: https://www.kaggle.com/sohier/calcofi

[55] R. G. Andrzejak, K. Lehnertz, F. Mormann, C. Rieke, P. David, and C. Elger, "Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state," *Physical review. E, Statistical, nonlinear, and soft matter physics*, 2002.

[56] H. Darwood, "Epicurious - recipes with rating and nutrition," 2017. [Online]. Available: https://www.kaggle.com/hugodarwood/epirecipes/

[57] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "Hycc: Compilation of hybrid protocols for practical secure computation," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018.

[58] V. A. Abril, P. Maene, N. Mertens, and N. P. Smart, "Bristol Fashion MPC Circuits," https://homes.esat.kuleuven.be/~nsmart/MPC/, 2019.

APPENDIX A
BUILDING BLOCKS

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

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

- One key shared between every pair of parties - k_{ij} for (P_i, P_j) where $i, j \in \{0,1,2,3\}$.
- One key shared between every group of three parties - k_{ijk} for (P_i, P_j, P_k) where $i, j, k \in \{0,1,2,3\}$.
- One key shared amongst all - $\mathsf{k}_{\mathcal{P}}$.

We present the ideal world functionality $\mathcal{F}_{\mathsf{setup}}$ below.

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

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys k_{ij} and k_{ijk} for $i, j, k \in \{0,1,2,3\}$ and $\mathsf{k}_{\mathcal{P}}$. Let y_i denote the keys corresponding to party P_i. Then

- $\mathsf{y}_i = (k_{01}, k_{02}, k_{03}, k_{012}, k_{013}, k_{023}$ and $\mathsf{k}_{\mathcal{P}})$ when $P_i = P_0$.
- $\mathsf{y}_i = (k_{01}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}$ and $\mathsf{k}_{\mathcal{P}})$ when $P_i = P_1$.
- $\mathsf{y}_i = (k_{02}, k_{12}, k_{23}, k_{012}, k_{023}, k_{123}$ and $\mathsf{k}_{\mathcal{P}})$ when $P_i = P_2$.
- $\mathsf{y}_i = (k_{03}, k_{13}, k_{23}, k_{013}, k_{023}, k_{123}$ and $\mathsf{k}_{\mathcal{P}})$ when $P_i = P_3$.

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

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

Figure 21: Functionality for Shared Key Setup

c) Generating Zero Share, Π_{Zero}: Protocol Π_{Zero} (Fig. 22) enables parties in $\mathcal{P}$ to generate a $\langle\cdot\rangle$-sharing of zero among P_1, P_2, P_3. In detail, parties P_1, P_2 and P_3 obtain values A, B and Γ respectively such that $A + B + \Gamma = 0$. In addition, P_0 obtains all the values A, B and Γ. The protocol is adapted to the 4PC setting from the 3PC protocol of [10], and we use $\mathcal{F}_{\mathsf{Zero}}$ to denote the ideal-world functionality for the same. We omit the proof for Π_{Zero} and refer readers to [10] since the protocols are almost similar.

Protocol Π_{Zero}

- Parties use the $\mathcal{F}_{\mathsf{setup}}$ functionality to establish the following set of keys among them:

$$\mathcal{P} \setminus \{P_3\} : \mathsf{k}_2, \qquad \mathcal{P} \setminus \{P_1\} : \mathsf{k}_3, \qquad \mathcal{P} \setminus \{P_2\} : \mathsf{k}_1$$

- Using the above set of keys and the PRF F, parties compute the following:
  - $P_0, P_1 : A = F(\mathsf{k}_2) - F(\mathsf{k}_1)$.
  - $P_0, P_2 : B = F(\mathsf{k}_3) - F(\mathsf{k}_2)$.
  - $P_0, P_3 : \Gamma = F(\mathsf{k}_1) - F(\mathsf{k}_3)$.

Figure 22: Generating $[\cdot]$-sharing of zero

APPENDIX B
ANALYSIS OF OUR 4PC PROTOCOL

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

Proof: During the offline phase, parties sample the λ-shares non-interactively using the shared key setup. During the online phase, P_i sends m-value to P_1, P_2, P_3 resulting in 1 round and a communication of at most 3ℓ bits (for the case when $P_i = P_0$). Parties P_1, P_2, P_3 then mutually exchange the hash of m-value received from P_0. Parties can combine m-values for several instances into a single hash and hence the cost gets amortized over multiple instances. ■

Lemma B.2 (Communication)**.** *Protocol Π_{aSh} (Fig. 2) requires 1 round and an amortized communication of 2ℓ bits in the offline phase.*

Proof: Protocol Π_{aSh} is performed entirely in the offline phase. During the protocol, P_0 computes and sends v_3 to both P_1 and P_2 resulting in 1 round and a communication

of 2ℓ bits. Parties P_1, P_2 mutually exchange hash of the value received from P_0 to ensure consistency. Similar to Π_{Sh}, hash for multiple instances can be combined and hence this cost gets amortized over multiple instances. ■

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

Proof: During the protocol, each party receives his/her missing share from another party, resulting in 1 round and a communication of 4ℓ bits. In addition, each party receives a hash of the missing share from another party for verification. The hash for multiple instances can be combined to a single hash and thus this cost gets amortized over multiple instances. ■

Lemma B.4 (Communication)**.** *Protocol* Π_{Mult} *(Fig. 4) requires* 1 *round and an amortized communication of* 3ℓ *bits in the offline phase and requires* 1 *round and an amortized communication of* 3ℓ *bits in the online phase.*

Proof: During the offline phase, each P_1, P_2, P_3 receives one share of γ_{xy} from another party resulting in 1 round and communication of 3ℓ bits. Also, each party receives a hash value from P_0 in the same round. The values corresponding to all the multiplication gates can be combined into a single hash resulting in an overall communication of just three hash values, which gets amortized. During the online phase, each of P_1, P_2, P_3 receives one $[\cdot]$-share of $\mathsf{m_z} - \mathsf{m_x}\mathsf{m_y}$ from another party resulting in 1 round and a communication of 3ℓ bits. Also, each party receives a hash value of the same from the third party. The values corresponding to all the multiplication gates can be combined into a single hash resulting in amortization of this cost. ■

Theorem B.5. Π_{4PC} *requires one round with an amortized communication of* $3\mathsf{M}$ *ring elements during the offline phase. In the online phase,* Π_{4PC} *requires one round with an amortized communication of at most* $3\mathsf{I}$ *ring elements in the Input-sharing stage,* D *rounds with an amortized communication of* $3\mathsf{M}$ *ring elements for evaluation stage and one round with an amortized communication of* $3\mathsf{O}$ *elements for the output-reconstruction stage.*

Proof: The proof follows from Lemmas B.1, B.4 and B.3. ■

Lemma B.6 (Communication)**.** *Protocol* Π_{fRec} *(Fig. 5) requires* 4 *rounds and an amortized communication of* 8ℓ *bits in the online phase.*

Proof: In the first round, P_0 receives a single bit from parties P_1, P_2, P_3. In the next round, P_0 sends back a single bit to P_1, P_2, P_3. In the third round, P_1, P_2, P_3 mutually exchanges the bit received from P_0. In the last round, each party receives the missing share from two other parties resulting in the communication of 8ℓ bits. In parallel, each party receives a hash of the missing share from the third party. Note that the first three rounds can be performed only once, for all the output wires together, amortizing the corresponding cost. In the last round, parties can compute a single hash corresponding to all the output wires resulting in getting this cost amortized over all the instances. ■

APPENDIX C
ANALYSIS OF CONVERSIONS

A. Building Blocks

Lemma C.1 (Communication)**.** *Protocol* Π_{vSh} *(Fig. 7) is non-interactive in the offline phase and requires* 1 *round and an amortized communication of at most* 2ℓ *bits in the online phase.*

Proof: The cost for the offline phase follows from Lemma B.1. In the online phase, parties P_i, P_j sends the masked value to P_1, P_2, P_3. For the case when $P_0 \in \{P_i, P_j\}$, this requires 1 round and a communication of 2ℓ bits. In other cases, communication of just ℓ bits is required. Note that the values corresponding to multiple instances of Π_{vSh} can be combined into a single hash, resulting in amortization of this cost. ■

Lemma C.2 (Communication)**.** *Protocol* $\Pi^{\mathbf{G}}_{\mathsf{vSh}}$ *(Fig. 8) is non-interactive in the offline phase, while it requires* 1 *round and an amortised communication of* κ *bits in the online phase.*

Proof: For the case when P_0 along with one of the garblers P_1, P_2, P_3 owns v, the protocol requires two commitments and one decommitment. ABY3 [5] has shown that the number of commitments can be limited to $2s$, when the number of values to be shared is larger than the statistical security parameter s. Consequently, the *amortized* cost per instance of $\Pi^{\mathbf{G}}_{\mathsf{vSh}}$ becomes κ bits (where the garbler sends the key $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$ to P_0). For the case when two of the garblers need to share multiple values, one garbler can combine all the actual keys to a single hash and send it to P_0, resulting in an amortized cost of κ bits. ■

Lemma C.3 (Communication)**.** *Protocol* Π_{DotP} *(Fig. 9) requires* 1 *round and an amortized communication of* 3ℓ *bits in the offline phase and requires* 1 *round and an amortized communication of* 3ℓ *bits in the online phase.*

Proof: The proof follows from Lemma B.4 since the protocol is similar to Π_{Mult}. ■

B. Sharing Conversions

We use $|\mathsf{Add}|$ and $|\mathsf{Sub}|$ to denote the size of garbled circuits corresponding to two ℓ-bit input adder and subtractor circuit respectively. We abuse the notation $|\mathsf{Decode}|$ to denote the size of decoding information for the corresponding garbled circuit. The lemmas in this section provide the amortized cost and omit the cost of hash values.

Lemma C.4 (Communication)**.** *Protocol* Π_{G2B} *(Fig. 10) requires* 1 *round and a communication of* $\kappa + 1 + |\mathsf{Decode}|$ *bits in the offline phase, while it requires* 1 *round and a communication of* 3 *bits in the online phase.*

Proof: During the offline phase, parties execute one instance of $\Pi^{\mathbf{G}}_{\mathsf{vSh}}$ and $\Pi^{\mathbf{B}}_{\mathsf{vSh}}$ resulting in 1 round and a communication of $\kappa + 1$ bits (Lemmas C.1 and C.2). Also, the decoding information for performing an XOR in the garbled world needs to be communicated to P_0. During the online phase, P_0 communicates a single bit to P_3. Also, P_0 performs the boolean sharing of the same bit resulting in a total

communication of 3 bits. The verification by P_3 can be pushed to the next round as long as the values are not revealed in the next round. Thus parties can proceed with the evaluation after the first round. ■

Lemma C.5 (Communication)**.** *Protocol* Π_{G2A} *(Fig. 11) requires* 1 *round and a communication of* $\ell\kappa + \ell + |\mathsf{Sub}| + |\mathsf{Decode}|$ *bits in the offline phase, while it requires* 1 *round and a communication of* 3ℓ *bits in the online phase.*

Proof: The protocol is similar to that of Π_{G2B} (Lemma C.4) with the main difference being a garbled subtractor circuit Sub is used for evaluating the final output. ■

Lemma C.6 (Communication)**.** *Protocol* Π_{B2G} *(Fig. 12) requires* 1 *round and a communication of* κ *bits in the offline phase, while it requires* 1 *round and a communication of* κ *bits in the online phase.*

Proof: The protocol Π_{B2G} involves the execution of an instance of protocol $\Pi^{\mathbf{G}}_{\mathsf{vSh}}$ in both the offline and online phases. The cost follows directly from Lemma C.2. ■

Lemma C.7 (Communication)**.** *Protocol* Π_{A2G} *(Fig. 13) requires* $-$ *rounds and a communication of* $\ell\kappa + |\mathsf{Sub}|$ *bits in the offline phase, while it requires* 1 *round and a communication of* $\ell\kappa$ *bits in the online phase.*

Proof: The protocol can be viewed as ℓ instances of Π_{B2G} (Lemma C.6), where each bit of v is converted to its garbled sharing. Moreover, a garbled subtractor circuit Sub is used to evaluate the output. ■

Lemma C.8 (Communication)**.** *Protocol* Π_{A2B} *(Fig. 14) requires* 1 *round and a communication of* $3\ell \log \ell + 2\ell$ *bits in the offline phase, while it requires* $1 + \log \ell$ *rounds and a communication of* $3\ell \log \ell + \ell$ *bits in the online phase.*

Proof: The protocol proceeds similarly to that of Π_{A2G} apart from the garbled world being replaced with the boolean world. Parties execute a single instance of $\Pi^{\mathbf{B}}_{\mathsf{vSh}}$ in both the offline and online phases. This results in one round and a communication of 2ℓ bits in the offline phase, while it results in one round and communication of ℓ bits in the online phases (Lemma C.1). Moreover, parties evaluate a boolean subtractor circuit Sub (Parallel Prefix Adder version mentioned in ABY3 [5]) of $\log \ell$ multiplicative depth and contain $\ell \log \ell$ AND gates. This results in an additional communication of $3\ell \log \ell$ bits in the offline and online phases (Lemma B.4). Moreover, the online rounds increases from 1 to $1 + \log \ell$. ■

Lemma C.9 (Communication)**.** *Protocol* Π_{Bit2A} *(Fig. 15) requires* 2 *rounds and a communication of* $3\ell + 1$ *bits in the offline phase, while it requires* 1 *round and a communication of* 3ℓ *bits in the online phase.*

Proof: During the offline phase, P_0 executes Π_{aSh} on u resulting in one round and a communication of 2ℓ bits (Lemma B.2). This is followed by parties P_1, P_2, P_3 performing the check to ensure the correctness of sharing done by P_0. During the check, P_1 sends one ring element and a bit to P_3, while P_2 sends a hash value. This results in an additional round and an amortized communication of $\ell + 1$ bits.

During the online phase, parties non-interactively generate the $[\![\cdot]\!]$-shares of v. This is followed by one arithmetic multiplication, resulting in one round and communication of 3ℓ bits in the online phase (Lemma B.4). Here, note that the offline phase for the multiplication is not required, since λ_{v} is set to 0 while executing the protocol $\Pi_{\mathsf{Sh}}(P_1, P_2, P_3, \mathsf{v})$, resulting in γ_{uv} to be set to 0. ■

Lemma C.10 (Communication)**.** *Protocol* Π_{B2A} *(Fig. 16) requires* 2 *rounds and a communication of* $3\ell^2 + \ell$ *bits in the offline phase, while it requires* 1 *round and a communication of* 3ℓ *bits in the online phase.*

Proof: The offline cost follows from that of protocol Π_{Bit2A} since the offline phase can be viewed as ℓ instances of that of Π_{Bit2A} (Lemma C.9). During the online phase, every pair from P_1, P_2, P_3 executes an instance of Π_{vSh} resulting in one round and communication of 3ℓ bits (Lemma C.1). ■

Lemma C.11 (Communication)**.** *Protocol* Π_{BitInj} *(Fig. 17) requires* 2 *rounds and a communication of* $6\ell + 1$ *bits in the offline phase, while it requires* 1 *round and a communication of* 3ℓ *bits in the online phase.*

Proof: The offline phase of Π_{BitInj} consists of generating $\langle\cdot\rangle$-shares of λ_{b} and $\lambda_{\mathsf{b}}\lambda_{\mathsf{v}}$. The case for λ_{b} is same as that of Π_{Bit2A} resulting in two rounds and a communication of $3\ell + 1$ bits. In parallel, P_0 execute Π_{aSh} on $\lambda_{\mathsf{b}}\lambda_{\mathsf{v}}$ resulting in an additional communication of 2ℓ bits. In order to verify the correctness of this sharing, P_1, P_2, P_3 performs a check. During the check, P_1 sends one ring element to P_3, while P_2 sends a hash value, resulting in an additional amortised communication of ℓ bits. Thus the offline phase requires two rounds and an amortised communication of $6\ell + 1$ bits.

During the online phase, every pair from P_1, P_2, P_3 executes an instance of Π_{vSh} resulting in one round and communication of 3ℓ bits (Lemma C.1). ■

Cost Comparison: Table IX provides a comparison of our sharing conversions with ABY3 [5]. Here, G1 denotes a two input garbled subtractor circuit, while $\mathsf{G2} = \mathsf{Gar}(2, \mathsf{Sub}, \ell)$ denotes two input garbled subtractor circuit along with its decoding information. Similarly, $\mathsf{G3} = \mathsf{Gar}(3, \mathsf{Sub}, \ell)$ and $\mathsf{G4} = \mathsf{Gar}(3, \mathsf{Adder}, \ell)$ denote three input garbled circuit for subtraction and addition respectively. Here ℓ denotes the ring size in bits.

APPENDIX D
ANALYSIS OF ML PROTOCOLS

Lemma D.1 (Correctness)**.** *In the offline phase of protocol* Π_{MultTr} *(Fig. 18), if a corrupt* P_0 *generates incorrect* $[\![\mathsf{r}^{\mathsf{t}}]\!]$ *sharing, then the honest evaluators* P_1, P_2, P_3 `abort`.

Proof: To see the correctness, it suffices to show that $\mathsf{m}_1 + \mathsf{m}_2 = \mathsf{c}$ where $\mathsf{m}_1 = \mathsf{r}_2 - 2^{\mathsf{d}}\mathsf{r}^{\mathsf{t}}_2 - \mathsf{r}_{\mathsf{d},2} + \mathsf{c}$ and $\mathsf{m}_2 = (\mathsf{r}_1 + \mathsf{r}_3) - 2^{\mathsf{d}}(\mathsf{r}^{\mathsf{t}}_1 + \mathsf{r}^{\mathsf{t}}_3) - (\mathsf{r}_{\mathsf{d},1} + \mathsf{r}_{\mathsf{d},3})$. Note that $\mathsf{r} = 2^{\mathsf{d}}\mathsf{r}^{\mathsf{t}} + \mathsf{r}_{\mathsf{d}}$ where r^{t} denoted the truncated value of r and r_{d} denoted the last d bits

Conv.	Work	Offline		Online	
		R.	Comm.	R.	Comm.
G2B	ABY3	1	κ	1	κ
	This	1	$\kappa + 1 + \lvert\mathsf{Decode}\rvert$	1	3
G2A	ABY3	1	$\lvert\mathsf{G3}\rvert + \ell\kappa$	1	$2\ell\kappa$
	This	1	$\ell\kappa + \ell + \lvert\mathsf{G2}\rvert$	1	3ℓ
B2G	ABY3	0	0	1	2κ
	This	1	κ	1	κ
A2G	ABY3	1	$\lvert\mathsf{G4}\rvert$	1	$2\ell\kappa$
	This	1	$\ell\kappa + \lvert\mathsf{G1}\rvert$	1	$\ell\kappa$
A2B	ABY3	3	$12\ell \log \ell + 12\ell$	$1 + \log \ell$	$9\ell \log \ell + 9\ell$
	This	1	$3\ell \log \ell + 2\ell$	$1 + \log \ell$	$3\ell \log \ell + \ell$
Bit2A	ABY3	1	24ℓ	2	18ℓ
	This	2	$3\ell + 1$	1	3ℓ
B2A	ABY3	3	$12\ell \log \ell + 12\ell$	$1 + \log \ell$	$9\ell \log \ell + 9\ell$
	This	2	$3\ell^2 + \ell$	1	3ℓ
BitInj	ABY3	1	36ℓ	3	27ℓ
	This	2	$6\ell + 1$	1	3ℓ

Table IX: Sharing conversions of ABY3 and Ours.

of r. Then,

$$\begin{aligned}\mathsf{m}_1 + \mathsf{m}_2 &= \left(\mathsf{r}_2 - 2^{\mathsf{d}}\mathsf{r}_2^{\mathsf{t}} - \mathsf{r}_{\mathsf{d},2} + \mathsf{c}\right) \\ &\quad + \left((\mathsf{r}_1 + \mathsf{r}_3) - 2^{\mathsf{d}}(\mathsf{r}_1^{\mathsf{t}} + \mathsf{r}_3^{\mathsf{t}}) - (\mathsf{r}_{\mathsf{d},1} + \mathsf{r}_{\mathsf{d},3})\right) \\ &= (\mathsf{r}_1 + \mathsf{r}_2 + \mathsf{r}_3) - 2^{\mathsf{d}}(\mathsf{r}_1^{\mathsf{t}} + \mathsf{r}_2^{\mathsf{t}} + \mathsf{r}_3^{\mathsf{t}}) \\ &\quad - (\mathsf{r}_{\mathsf{d},1} + \mathsf{r}_{\mathsf{d},2} + \mathsf{r}_{\mathsf{d},3}) + \mathsf{c} \\ &= (\mathsf{r}) - (2^{\mathsf{d}}\mathsf{r}^{\mathsf{t}} + \mathsf{r}_{\mathsf{d}}) + \mathsf{c} = 0 + \mathsf{c} = \mathsf{c}\end{aligned}$$

■

Lemma D.2 (Communication)**.** *Protocol* Π_{MultTr} *(Fig. 18) requires* 2 *rounds and a communication of* 6ℓ *bits in the offline phase, while it requires* 1 *round and a communication of* 3ℓ *bits in the online phase.*

Proof: During the offline phase, the offline phase of Π_{Mult} is executed, resulting in one round and communication of 3ℓ bits. In parallel, P_0 executes Π_{aSh} on r^{t} resulting in an additional communication of 2ℓ bits (Lemma B.2). To verify the correctness of this sharing, parties P_1, P_2, P_3 performs a check, where P_1 sends one ring element and hash value to P_2. This results in an additional amortized communication of ℓ bits. Thus the offline phase requires two rounds and an amortized communication of 6ℓ bits. The cost for online phase follows directly from Π_{Mult} (Lemma B.4) ■

Lemma D.3 (Communication)**.** *Protocol* Π_{BitExt} *(Fig. 19) requires* 1 *round and a communication of* $4\ell + 1$ *bits in the offline phase, while it requires* 3 *rounds and a communication of* $5\ell + 2$ *bits in the online phase.*

Proof: During the offline phase, parties execute one instance each of Π_{vSh} and $\Pi^{\mathbf{B}}_{\mathsf{vSh}}$ resulting in one round and a communication of $\ell + 1$ bits (Lemma C.1). Also, the offline phase for multiplication is performed resulting in an additional communication of 3ℓ bits.

During the online phase, parties first execute an arithmetic multiplication, resulting in one round and communication of 3ℓ bits (Lemma B.4). The value rv is reconstructed towards both P_0 and P_3, resulting in an additional round and an amortized communication of 2ℓ bits. This is followed by the last round, where parties execute one instance of $\Pi^{\mathbf{B}}_{\mathsf{vSh}}$ resulting in a communication of 2 bits. Thus the online phase requires three rounds and an amortized communication of $5\ell + 2$ bits. ■

Lemma D.4 (Communication)**.** *Protocol ReLU (*Π_{relu}*) requires* 3 *rounds and a communication of* $8\ell + 2$ *bits in the offline phase, while it requires* 4 *rounds and a communication of* $8\ell+2$ *bits in the online phase.*

Proof: The cost follows directly from Lemmas D.3 and C.11. ■

Lemma D.5 (Communication)**.** *Protocol Sigmoid (*Π_{sig}*) requires* 3 *rounds and a communication of* $15\ell + 7$ *bits in the offline phase, while it requires* 5 *rounds and a communication of* $16\ell + 7$ *bits in the online phase.*

Proof: The cost follows directly from Lemmas D.3, B.4, C.9 and C.11. ■

Cost Comparison: Table X provides a comparison of our work with ABY3 [5], in terms of ML protocols and special conversions.

Conv.	Work	Offline		Online	
		R.	Comm.	R.	Comm.
Π_{MultTr}	ABY3	$2\ell - 2$	$96\ell - 42d - 84$	1	12ℓ
	This	2	6ℓ	1	3ℓ
Π_{BitExt}	ABY3	1	$24\ell \log \ell$	$\log \ell$	$18\ell \log \ell$
	This	1	$4\ell + 1$	3	$5\ell + 2$
ReLU	ABY3	3	60ℓ	$3 + \log \ell$	45ℓ
	This	3	$8\ell + 2$	4	$8\ell + 2$
Sigmoid	ABY3	3	$108\ell + 12$	$4 + \log \ell$	$81\ell + 9$
	This	3	$15\ell + 7$	5	$16\ell + 7$

Table X: ML conversions of ABY3 and Ours. d denotes the number of features

APPENDIX E
BENCHMARKING

A. Motivating 4PC

In this section, we justify the reason for operating in the 4PC setting over the 3PC setting through the lens of benchmarking and practical efficiency. We begin with the comparison of our 4PC secret sharing scheme with that of Gordon et al. [26], which shows the reason we built a new 4PC. In Section E-A2 we compare the efficiency of our framework with ABY3. Since machine learning models take a long time to train, monetary cost becomes an important factor to consider when building PPML protocols, as shown in [27].

Monetary cost is computed by calculating the total running time for each of the servers and the cost of hiring them, which is based upon the computing power of the server. Total running time is the total time taken for the evaluation phase, which excludes the input sharing and reconstruction phase. So, if we have a protocol that communicates very less but the total running time for the servers is high due to the computation, the monetary cost will also be high. We compare the total running

times of our protocol against the others to show that we have a better balance in terms of computation and communication, making our monetary cost lower than the rest.

Ref.	P_0	P_1	P_2	P_3	Total
Gordon	7.84	3.13	7.34	3.21	21.52
This	0.00	6.19	6.19	3.81	16.19

Table XI: Total Online Runtime (in seconds) of Gordon et al. and **This** for evaluation on an AES-128 circuit (lower = better) over WAN.

1) Comparison with the protocol of Gordon et al. [26]: In Table XI, we compare the total online runtime of our 4PC protocol with that of Gordon et al. [26] for evaluating an AES-128 [58] circuit. As evident from the table, P_0 does not have to be active during the online phase, except for the sharing and the reconstruction phase. This means we can shut down the server that corresponds to P_0 for the entirety of the online evaluation phase, saving a lot in terms of monetary cost. The preprocessing phase for both protocols requires just 1 round of interaction.

2) Motivation of 4PC for ML: Here, we argue that even though we operate in the 4PC setting, meaning we have 4 servers active instead of 3 as in ABY3, our total monetary cost is still lower than that of ABY3. Table XII shows that our total runtime for both training and prediction phases in the malicious case is lower for all the algorithms considered. For the training phase, we use a batch size of 128. The number of features for both training and prediction is 784.

Phase	Ref.	Linear Regression	Logistic Regression	NN	CNN
Training (s)	ABY3	2.01	8.92	38.41	41.45
	This	0.92	3.76	13.07	13.19
Prediction (s)	ABY3	1.45	8.36	21.12	22.48
	This	0.44	2.74	6.90	6.93

Table XII: Total Online Runtime (in seconds) of ABY3 (Malicious) and **This** for Training and Prediction of Linear, Logistic, NN, and CNN models for $d = 784$ (lower = better) over a WAN setting.

B. Comparison with the ML framework of ABY3 [5] in the semi-honest setting

We compare the performance of our protocol with the semi-honest version of ABY3, giving them an advantage in terms of the threat model. We use ABY3S to denote the ABY3 protocol in the semi-honest setting. Our performance is the same for linear regression but as the protocols get more complex, the difference in performance increases in our favour. This is because in linear regression though we have the additional overhead of a hash, the cost is amortized, making our cost the same as ABY3S. For the other three algorithms, however, we outperform ABY3S due to our efficient protocols for bit extraction, bit injection, and Bit2A. Although we are worse than ABY3 in the offline phase due to our preprocessing for multiplication, our online phase is a lot more efficient for both training and prediction phases.

Algorithm	Ref.	LAN (#it/sec)	WAN (#it/min)
Linear Regression	ABY3S	1098.90	195.13
	This	1098.90	195.13
Logistic Regression	ABY3S	90.29	35.48
	This	307.41	55.75
Neural Networks	ABY3S	1.01	8.13
	This	23.00	13.94
CNN	ABY3S	0.37	7.13
	This	10.46	13.86

Table XIII: Comparison of Online Phase of ABY3 (Semi-Honest) and **This** for ML Training (higher = better)

In Table XIII, we compare the online phase of both protocols for ML training, in terms of the number of iterations per second they can process. In Table XIV and Table XV we compare the online phase for prediction through two benchmarking units, one being the runtime and the other being the throughput, which is the number of queries we can process per second.

Network	Ref.	Linear Regression	Logistic Regression	NN	CNN
LAN (ms)	ABY3S	0.30	9.14	480.81	1185.70
	This	0.30	2.55	17.17	39.63
WAN (s)	ABY3S	0.16	1.54	4.07	4.47
	This	0.16	0.93	2.31	2.31

Table XIV: Online Runtime of ABY3 (Semi-honest) and **This** for Secure Prediction of Linear, Logistic, NN, and CNN models for $d = 784$ (lower = better)

Algorithm	Ref.	LAN (queries/sec)	WAN (queries/min)
Linear Regression	ABY3S	106666.67	12488.62
	This	106666.67	12488.62
Logistic Regression	ABY3S	3512.62	1248.86
	This	12549.02	2081.46
Neural Networks	ABY3S	66.41	211.18
	This	153.39	368.13
CNN	ABY3S	21.47	51.54
	This	37.43	89.84

Table XV: Comparison of Online TP of ABY3 (Semi-Honest) and **This** for Secure Prediction (higher = better)

APPENDIX F
SECURITY OF OUR CONSTRUCTIONS

In this section, we prove the security of our Π_{4PC} protocol in the $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{Zero}}\}$-hybrid model, using the real world-ideal world paradigm. The ideal world functionality that realises Π_{4PC} is given in Fig. 23.

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

$\mathcal{F}_{\mathsf{4PC}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$ and is parameterized by a 4-ary function f, represented by a publicly known arithmetic circuit ckt over $\mathbb{Z}_{2^\ell}$.

Input: Upon receiving the input $\mathsf{x}, \ldots, \mathsf{x}_\mathsf{I}$ from the respective parties in $\mathcal{P}$, do the following: if $(\mathsf{Input}, *)$ message was received from P_i corresponding to x_j, then ignore. Otherwise record $\mathsf{x}'_j = \mathsf{x}_j$ internally. If $\mathsf{x}'_j \neq \mathbb{Z}_{2^\ell}$, consider $\mathsf{x}'_j = \mathtt{abort}$.

Output to adversary: If there exists $j \in \{1, \ldots, \mathsf{I}\}$ such that $\mathsf{x}'_j = \mathtt{abort}$, send $(\mathsf{Output}, \perp)$ to all the parties. Else, send $(\mathsf{Output}, (\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{O}))$ to the adversary $\mathcal{S}$, where $(\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{O}) = f(\mathsf{x}'_1, \ldots, \mathsf{x}'_\mathsf{I})$.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, (\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{O}))$.

Figure 23: Ideal world 4PC functionality

We begin with the case when P_0 is corrupted.

Theorem F.1. *In $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{Zero}}\}$-hybrid model, Π_{4PC} securely realizes the functionality $\mathcal{F}_{\mathsf{4PC}}$ against a static, malicious adversary $\mathcal{A}$, who corrupts P_0.*

Proof: Let $\mathcal{A}$ be a real-world adversary corrupting the distributor P_0 during the protocol Π_{4PC}. We present an ideal-world adversary (simulator) $\mathcal{S}_{\mathsf{4PC}}$ for $\mathcal{A}$ in Fig. 24 that simulates messages for corrupt P_0. The only communication to P_0 is during the output-reconstruction stage in the online phase. $\mathcal{S}_{\mathsf{4PC}}$ can easily simulate these messages, with the knowledge of function output and the masks corresponding to the circuit-output wires.

Simulator $\mathcal{S}_{\mathsf{4PC}}$

The simulator plays the role of the honest parties P_1, P_2, P_3 and simulates each step of Π_{4PC} to corrupt P_0 as follows and finally outputs $\mathcal{A}$'s output. The simulator initializes a boolean variable $\mathsf{flag} = 0$, which indicates whether an honest party aborts during the protocol.

Offline Phase: $\mathcal{S}_{\mathsf{4PC}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the keys $(k_{01}, k_{02}, k_{03}, k_{012}, k_{013}, k_{023}$ and $\mathsf{k}_\mathcal{P})$ to P_0. By emulating $\mathcal{F}_{\mathsf{setup}}$, it learns the λ-masks for all the wires in ckt.

- *Sharing Circuit-input Values*: Here the simulator has to do nothing, as the offline phase involves no communication.
- *Gate Evaluation*: No simulation is needed for the offline phase of an addition gate. For a multiplication gate g, the simulator emulates $\mathcal{F}_{\mathsf{Zero}}$ and gives A, B, Γ to P_0. It then receives hash of $\gamma_{\mathsf{xy},1}, \gamma_{\mathsf{xy},2}$ and $\gamma_{\mathsf{xy},3}$ from P_0 on behalf of P_2, P_3 and P_1 respectively. $\mathcal{S}_{\mathsf{4PC}}$ computes $\gamma_{\mathsf{xy},i}$ for $i \in \{1,2,3\}$ and sets $\mathsf{flag} = 1$, if any of the hash values received is inconsistent with the values computed.
- *Output Reconstruction*: Here the simulator has to do nothing, as the offline phase involves no communication.

If $\mathsf{flag} = 1$, $\mathcal{S}_{\mathsf{4PC}}$ invokes $\mathcal{F}_{\mathsf{4PC}}$ with input $\perp$ on behalf of P_0.

Online Phase:

- *Sharing Circuit-input Values*: For every input v_j of P_0, $\mathcal{S}_{\mathsf{4PC}}$ receives $\mathsf{m}_{\mathsf{v}_j}$ of behalf of P_1, P_2, P_3. $\mathcal{S}_{\mathsf{4PC}}$ sets $\mathsf{flag} = 1$ if the received values mismatch. Else, it computes the input v_j using $\mathsf{m}_{\mathsf{v}_j}$ and the λ-masks obtained in the offline phase.
- *Gate Evaluation*: No simulation is needed for the online phase of both addition and multiplication gates.
- *Obtaining function outputs*: If $\mathsf{flag} = 1$, $\mathcal{S}_{\mathsf{4PC}}$ invokes $\mathcal{F}_{\mathsf{4PC}}$ with input $\perp$ on behalf of P_0. Else it sends inputs $\{\mathsf{v}_j\}$ extracted on behalf of P_0 to $\mathcal{F}_{\mathsf{4PC}}$ and receives the function outputs $\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{O}$.
- *Output Reconstruction*: For each output y_j for $j \in 1, \ldots, \mathsf{O}$, $\mathcal{S}_{\mathsf{4PC}}$ computes $\mathsf{m}_{\mathsf{y}_j} = \mathsf{y}_j + \lambda_{\mathsf{y}_j}$, and sends $\mathsf{m}_{\mathsf{y}_j}$ and $\mathsf{H}(\mathsf{m}_{\mathsf{y}_j})$ to P_0 on behalf of P_1 and P_2 respectively. In parallel, he receives $\mathsf{H}(\lambda_{\mathsf{y}_j,i})$ for $i \in \{1,2,3\}$ from P_0, on behalf of P_1, P_2, P_3. $\mathcal{S}_{\mathsf{4PC}}$ initializes the set I to $\varnothing$. P_i for $i \in \{1,2,3\}$, is added to I, if hash of $\lambda_{\mathsf{y}_j,i}$ mismatches with the corresponding hash received from P_0. $\mathcal{S}_{\mathsf{4PC}}$ then sends I to $\mathcal{F}_{\mathsf{4PC}}$ and terminates.

Figure 24: Simulator for the case of corrupt P_0

The proof now simply follows from the fact that simulated view and real-world view of the adversary are computationally indistinguishable. ■

We next consider the case, when the adversary corrupts one of the evaluators, say P_1. The cases of corrupt P_2 and P_3 are handled symmetrically.

Theorem F.2. *In the $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{Zero}}\}$-hybrid model, Π_{4PC} securely realizes the functionality $\mathcal{F}_{\mathsf{4PC}}$ against a static, malicious adversary $\mathcal{A}$, who corrupts P_1.*

Proof: Let $\mathcal{A}$ be a real-world adversary corrupting the evaluator P_1 during the protocol Π_{4PC}. We now present the steps of the ideal-world adversary (simulator) $\mathcal{S}_{\mathsf{4PC}}$ for $\mathcal{A}$ for this case in Fig. 25.

Simulator $\mathcal{S}_{\mathsf{4PC}}$

The simulator plays the role of the honest parties P_0, P_2, P_3 and simulates each step of Π_{4PC} to corrupt P_1 as follows and finally outputs $\mathcal{A}$'s output. The simulator initializes a boolean variable $\mathsf{flag} = 0$, which indicates whether an honest party aborts during the protocol.

Offline Phase: $\mathcal{S}_{\mathsf{4PC}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the keys $(k_{01}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}$ and $\mathsf{k}_\mathcal{P})$ to P_1. By emulating $\mathcal{F}_{\mathsf{setup}}$, it learns the λ-masks for all the wires in ckt.

- *Sharing Circuit-input Values*: Here the simulator has to do nothing, as the offline phase involves no communication.
- *Gate Evaluation*: No simulation is needed for the offline phase of an addition gate. For a multiplication gate $\mathsf{g} = (\mathsf{w}_\mathsf{x}, \mathsf{w}_\mathsf{y}, \mathsf{w}_\mathsf{z}, \times)$, the simulator emulates $\mathcal{F}_{\mathsf{Zero}}$ and gives A to P_1. $\mathcal{S}_{\mathsf{4PC}}$ computes $\gamma_{\mathsf{xy},i}$ for $i \in \{1,2,3\}$ and sends $\gamma_{\mathsf{xy},3}$ and $\mathsf{H}(\gamma_{\mathsf{xy},3})$ to P_1 on behalf of P_2 and P_0 respectively. It receives $\gamma_{\mathsf{xy},2}$ from P_1 on behalf of P_3 and sets $\mathsf{flag} = 1$, if the received value is inconsistent.
- *Output Reconstruction*: Here the simulator has to do nothing, as the offline phase involves no communication.

If $\mathsf{flag} = 1$, $\mathcal{S}_{\mathsf{4PC}}$ invokes $\mathcal{F}_{\mathsf{4PC}}$ with input $\perp$ on behalf of P_0.

Online Phase:

- *Sharing Circuit-input Values*: For every input v_j of P_1, $\mathcal{S}_{\mathsf{4PC}}$ receives $\mathsf{m}_{\mathsf{v}_j}$ of behalf of P_2 and P_3. $\mathcal{S}_{\mathsf{4PC}}$ sets $\mathsf{flag} = 1$ if the received values mismatch. Else, it computes the input v_j using $\mathsf{m}_{\mathsf{v}_j}$ and the λ-masks obtained in the offline phase. For every input v_k of P_i for $i \in \{0,2,3\}$, $\mathcal{S}_{\mathsf{4PC}}$ sets $\mathsf{v}_k = 0$ and

sends $\mathsf{m}_{\mathsf{v}_k} = 0 + \lambda_{\mathsf{v}_k}$ to P_1 on behalf of P_i. $\mathcal{S}_{\mathsf{4PC}}$ performs the exchange of hash of $\mathsf{m}_{\mathsf{v}_k}$ honestly.

– *Gate Evaluation*: No simulation is needed for the online phase of addition gates. For a multiplication gate $\mathsf{g} = (\mathsf{w}_\mathsf{x}, \mathsf{w}_\mathsf{y}, \mathsf{w}_\mathsf{z}, \times)$, $\mathcal{S}_{\mathsf{4PC}}$ computes and sends $[\mathsf{m}'_\mathsf{z}]_1$ and $\mathsf{H}([\mathsf{m}'_\mathsf{z}]_1)$ to P_1 on behalf of P_2 and P_3 respectively. It receives $\mathsf{H}([\mathsf{m}'_\mathsf{z}]_2)$ and $[\mathsf{m}'_\mathsf{z}]_3$ from P_1 on behalf of P_2 and P_3 respectively.$\mathcal{S}_{\mathsf{4PC}}$ sets $\mathsf{flag} = 1$, if any of the received values are inconsistent.

– *Obtaining function outputs*: If $\mathsf{flag} = 1$, $\mathcal{S}_{\mathsf{4PC}}$ invokes $\mathcal{F}_{\mathsf{4PC}}$ with input $\perp$ on behalf of P_1. Else it sends inputs $\{\mathsf{v}_j\}$ extracted on behalf of P_1 to $\mathcal{F}_{\mathsf{4PC}}$ and receives the function outputs $\mathsf{y}_1, \ldots, \mathsf{y}_\mathsf{O}$.

– *Output Reconstruction*: For each output y_j for $j \in 1, \ldots, \mathsf{O}$, $\mathcal{S}_{\mathsf{4PC}}$ computes $\lambda_{\mathsf{y}_j} = \mathsf{m}_{\mathsf{y}_j} - \mathsf{y}_j$, and sends λ_{y_j} and $\mathsf{H}(\lambda_{\mathsf{y}_j})$ to P_1 on behalf of P_2 and P_0 respectively. In parallel, it receives $\lambda_{\mathsf{y}_j,3}$ and $\mathsf{H}(\mathsf{m}_{\mathsf{y}_j})$ from P_1 on behalf of P_3 and P_0 respectively. P_0 is added to I, if hash of $\mathsf{m}_{\mathsf{y}_j}$ mismatches with the corresponding hash received from P_1. Similarly, P_3 is added to I, if $\lambda_{\mathsf{y}_j,3}$ mismatches with the corresponding value received from P_1. $\mathcal{S}_{\mathsf{4PC}}$ then sends I to $\mathcal{F}_{\mathsf{4PC}}$ and terminates.

Figure 25: Simulator for the case of corrupt P_1

■

A. Security Proof in Detail

This section covers the security proofs for most of our constructions. The proofs for the rest can be easily derived. The proofs use the real-world/ideal-world paradigm in which $\mathcal{A}$ is the adversary in the real-world and $\mathcal{S}$ is the simulator for the ideal-world, which acts as the honest parties in the protocol and simulates messages received by $\mathcal{A}$. The simulator maintains a flag which is set to 0 at the start of the protocol. If an honest party `abort`, the flag is set to 1. Simulator for a particular protocol is represented as $\mathcal{S}$ with the protocol name as the subscript.

The simulation for a circuit ckt proceeds as follows: We start with the input sharing phase, and $\mathcal{S}$ sets the input of the honest parties to 0. The simulator can extract the input of the $\mathcal{A}$ from the sharing protocol Π_{Sh}, details of which are provided in the simulation for Π_{Sh}. Doing so gives the $\mathcal{S}$ all the inputs for the entire circuit, which means it can compute all the intermediate values and the output of the circuit. As we will see later, $\mathcal{S}$ will use this information to simulate each component of a circuit ckt.

For each of the constructions, we provide simulation proof for the case of corrupt P_0 and P_1. The cases of corrupt P_2 and P_3 follows similar to that of P_1.

a) Sharing Protocol: The ideal functionality realising protocol Π_{Sh} is presented in Fig. 26.

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

$\mathcal{F}_{\mathsf{Sh}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Sh}}$ receives the input v from party P_i while it receives $\perp$ from the other parties. If $\mathsf{v} = \perp$, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Randomly select $\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3}$ from $\mathbb{Z}_{2^\ell}$ and set $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_{\mathsf{v},1} + \lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3}$. The output shares are set as:

$$[\![\mathsf{v}]\!]_{P_0} = (\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3}) \qquad [\![\mathsf{v}]\!]_{P_1} = (\mathsf{m}_\mathsf{v}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3})$$

$$[\![\mathsf{v}]\!]_{P_2} = (\mathsf{m}_\mathsf{v}, \lambda_{\mathsf{v},3}, \lambda_{\mathsf{v},1}) \qquad [\![\mathsf{v}]\!]_{P_3} = (\mathsf{m}_\mathsf{v}, \lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2})$$

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, [\![\mathsf{v}]\!]_\mathcal{S})$ to the adversary $\mathcal{S}$, where $[\![\mathsf{v}]\!]_\mathcal{S}$ denotes the share of v corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest parties P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, [\![\mathsf{v}]\!]_i)$, where $[\![\mathsf{v}]\!]_i$ denotes the share of v corresponding to the honest party P_i.

Figure 26: Functionality for protocol Π_{Sh}

The simulator for the case of corrupt P_0 appears in Fig. 27.

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

Offline Phase: $\mathcal{S}_{\mathsf{Sh}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the keys $(k_{01}, k_{02}, k_{03}, k_{012}, k_{013}, k_{023}$ and $\mathsf{k}_\mathcal{P})$ to $\mathcal{A}$. By emulating $\mathcal{F}_{\mathsf{setup}}$, it learns the λ-values corresponding to input v.

– If $P_i = P_0$, $\mathcal{S}_{\mathsf{Sh}}$ computes $\lambda_{\mathsf{v},j}$ for $j \in 1, 2, 3$ on behalf of each P_j using the shared key.

– If $P_i = P_k$ for $k \in \{1, 2, 3\}$, $\mathcal{S}_{\mathsf{Sh}}$ computes $\lambda_{\mathsf{v},k}$ using the key $\mathsf{k}_\mathcal{P}$. In addition, the λ_v-shares corresponding to the honest parties are computed honestly by $\mathcal{S}_{\mathsf{Sh}}$.

Online Phase:

– If $P_i = P_0$, $\mathcal{S}_{\mathsf{Sh}}$ receives m_v of behalf of P_1, P_2, P_3. $\mathcal{S}_{\mathsf{Sh}}$ sets $\mathsf{flag} = 1$ if the received values mismatch. Else, it computes the input $\mathsf{v} = \mathsf{m}_\mathsf{v} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2} - \lambda_{\mathsf{v},3}$.

– If $P_i \neq P_0$, $\mathcal{S}_{\mathsf{Sh}}$ sets $\mathsf{v} = 0$ by assigning $\mathsf{m}_\mathsf{v} = \lambda_{\mathsf{v},1} + \lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3}$.

If $\mathsf{flag} = 0$ and $P_i = P_0$, $\mathcal{S}_{\mathsf{Sh}}$ invokes $\mathcal{F}_{\mathsf{Sh}}$ with input v on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{Sh}}$ with input $\perp$ on behalf of P_0.

Figure 27: Simulator $\mathcal{S}_{\mathsf{Sh}}$ for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 28.

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

Offline Phase: $\mathcal{S}_{\mathsf{Sh}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the keys $(k_{01}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}$ and $\mathsf{k}_\mathcal{P})$ to $\mathcal{A}$. By emulating $\mathcal{F}_{\mathsf{setup}}$, it learns the λ-values corresponding to input v.

– If $P_i = P_1$, $\mathcal{S}_{\mathsf{Sh}}$ computes $\lambda_{\mathsf{v},1}$ using the key $\mathsf{k}_\mathcal{P}$.

– If $P_i = P_k$ for $k \in \{0, 2, 3\}$, the λ_v-shares corresponding to the honest parties are computed honestly by $\mathcal{S}_{\mathsf{Sh}}$.

Online Phase:

– If $P_i = P_1$, $\mathcal{S}_{\mathsf{Sh}}$ receives m_v of behalf of P_2, P_3. $\mathcal{S}_{\mathsf{Sh}}$ sets $\mathsf{flag} = 1$ if the received values mismatch. Else, it computes the input $\mathsf{v} = \mathsf{m}_\mathsf{v} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2} - \lambda_{\mathsf{v},3}$.

– If $P_i \neq P_1$, $\mathcal{S}_{\mathsf{Sh}}$ sets $\mathsf{v} = 0$ by assigning $\mathsf{m}_\mathsf{v} = \lambda_{\mathsf{v},1} + \lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3}$. $\mathcal{S}_{\mathsf{Sh}}$ sends m_v to P_1 on behalf of the sender

If $\mathsf{flag} = 0$ and $P_i = P_1$, $\mathcal{S}_{\mathsf{Sh}}$ invokes $\mathcal{F}_{\mathsf{Sh}}$ with input v on behalf

of P_1. Else it invokes $\mathcal{F}_{\mathsf{Sh}}$ with input $\perp$ on behalf of P_1.

Figure 28: Simulator $\mathcal{S}_{\mathsf{Sh}}$ for the case of corrupt P_1

b) Verifiable Arithmetic/Boolean Sharing: The ideal functionality realising protocol Π_{vSh} is presented in Fig. 29.

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

$\mathcal{F}_{\mathsf{vSh}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Sh}}$ receives the input v from parties P_i, P_j while it receives $\perp$ from the other parties. If the received values mismatch, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Randomly select $\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3}$ from $\mathbb{Z}_{2^\ell}$ and set $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v},1} + \lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3}$. The output shares are set as:

$$[\![\mathsf{v}]\!]_{P_0} = (\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3}) \quad [\![\mathsf{v}]\!]_{P_1} = (\mathsf{m_v}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3})$$

$$[\![\mathsf{v}]\!]_{P_2} = (\mathsf{m_v}, \lambda_{\mathsf{v},3}, \lambda_{\mathsf{v},1}) \quad [\![\mathsf{v}]\!]_{P_3} = (\mathsf{m_v}, \lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2})$$

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, [\![\mathsf{v}]\!]_{\mathcal{S}})$ to the adversary $\mathcal{S}$, where $[\![\mathsf{v}]\!]_{\mathcal{S}}$ denotes the share of v corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest parties P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, [\![\mathsf{v}]\!]_i)$, where $[\![\mathsf{v}]\!]_i$ denotes the share of v corresponding to the honest party P_i.

Figure 29: Functionality for protocol Π_{vSh}

The simulator for the case of corrupt P_0 appears in Fig. 30.

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

Offline Phase: The simulation for the offline phase is similar to that of $\Pi_{\mathsf{Sh}}(P_i, \mathsf{v})$ for the case of corrupt P_0 (Fig. 27).

Online Phase: $\mathcal{S}_{\mathsf{vSh}}$ proceeds similar to that of $\Pi_{\mathsf{Sh}}(P_i, \mathsf{v})$ for the case of corrupt P_0 (Fig. 27). In addition, if $P_j = P_0$, it receives $\mathsf{H}(\mathsf{m_v})$ from $\mathcal{A}$ on behalf of P_1, P_2, and P_3. $\mathcal{S}_{\mathsf{vSh}}$ sets $\mathsf{flag} = 1$ if the received values mismatch.

If $\mathsf{flag} = 0$ and $P_j = P_0$, $\mathcal{S}_{\mathsf{vSh}}$ invokes $\mathcal{F}_{\mathsf{vSh}}$ with input v on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{vSh}}$ with input $\perp$ on behalf of P_0.

Figure 30: Simulator $\mathcal{S}_{\mathsf{vSh}}$ for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 31.

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

Offline Phase: The simulation for the offline phase is similar to that of $\Pi_{\mathsf{Sh}}(P_i, \mathsf{v})$ for the case of corrupt P_1 (Fig. 28).

Online Phase: $\mathcal{S}_{\mathsf{vSh}}$ proceeds similar to that of $\Pi_{\mathsf{Sh}}(P_i, \mathsf{v})$ for the case of corrupt P_1 (Fig. 28). In addition, if $P_j = P_1$, it receives $\mathsf{H}(\mathsf{m_v})$ from $\mathcal{A}$ on behalf of P_2, and P_3. $\mathcal{S}_{\mathsf{vSh}}$ sets $\mathsf{flag} = 1$ if the received values mismatch.

If $\mathsf{flag} = 0$ and $P_i = P_1$, $\mathcal{S}_{\mathsf{vSh}}$ invokes $\mathcal{F}_{\mathsf{vSh}}$ with input v on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{vSh}}$ with input $\perp$ on behalf of P_1.

Figure 31: Simulator $\mathcal{S}_{\mathsf{vSh}}$ for the case of corrupt P_1

c) Reconstruction Protocol: The ideal functionality realising protocol Π_{Rec} is presented in Fig. 32.

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

$\mathcal{F}_{\mathsf{Rec}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Rec}}$ receives the $[\![\cdot]\!]$-shares of value v from party P_i for $i \in \{0, 1, 2, 3\}$. The shares are

$$[\![\mathsf{v}]\!]_{P_0} = (\lambda'_{\mathsf{v},1}, \lambda'_{\mathsf{v},2}, \lambda'_{\mathsf{v},3}) \quad [\![\mathsf{v}]\!]_{P_1} = (\mathsf{m}'_{\mathsf{v}}, \lambda''_{\mathsf{v},2}, \lambda''_{\mathsf{v},3})$$

$$[\![\mathsf{v}]\!]_{P_2} = (\mathsf{m}''_{\mathsf{v}}, \lambda'''_{\mathsf{v},3}, \lambda''_{\mathsf{v},1}) \quad [\![\mathsf{v}]\!]_{P_3} = (\mathsf{m}'''_{\mathsf{v}}, \lambda'''_{\mathsf{v},1}, \lambda'''_{\mathsf{v},2})$$

$\mathcal{F}_{\mathsf{Rec}}$ sends $\perp$ to every party if either of the following condition is met: i) $\lambda'_{\mathsf{v},1} \neq \lambda''_{\mathsf{v},1} \neq \lambda'''_{\mathsf{v},1}$, ii) $\lambda'_{\mathsf{v},2} \neq \lambda''_{\mathsf{v},2} \neq \lambda'''_{\mathsf{v},2}$, iii) $\lambda'_{\mathsf{v},3} \neq \lambda''_{\mathsf{v},3} \neq \lambda'''_{\mathsf{v},3}$ or iv) $\mathsf{m}'_{\mathsf{v}} \neq \mathsf{m}''_{\mathsf{v}} \neq \mathsf{m}'''_{\mathsf{v}}$. Else it proceeds with the computation.

Computation of output: Set $\mathsf{v} = \mathsf{m_v} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2} - \lambda_{\mathsf{v},3}$.

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, \mathsf{v})$ to the adversary $\mathcal{S}$.

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

Figure 32: Functionality for protocol Π_{Rec}

The simulator for the case of corrupt P_0 appears in Fig. 33. As mentioned at the beginning of this section, $\mathcal{S}$ knows all of the intermediate values and the output of the ckt. $\mathcal{S}$ uses this information to simulate the Π_{Rec} protocol.

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

Online Phase:

- $\mathcal{S}_{\mathsf{Rec}}$ computes $\mathsf{m_v} = \mathsf{v} + \lambda_{\mathsf{v},1} + \lambda_{\mathsf{v},2} + \lambda_{\mathsf{v},3}$. It then sends $\mathsf{m_v}$ and $\mathsf{H}(\mathsf{m_v})$ to $\mathcal{A}$ on behalf of P_1 and P_2 respectively.
- $\mathcal{S}_{\mathsf{Rec}}$ receives $\mathsf{H}(\lambda'_{\mathsf{v},1}), \mathsf{H}(\lambda'_{\mathsf{v},2})$ and $\mathsf{H}(\lambda'_{\mathsf{v},1})$ from $\mathcal{A}$ on behalf of P_1, P_2 and P_3 respectively. $\mathcal{S}_{\mathsf{Rec}}$ sets $\mathsf{flag} = 1$ if any of the received hash values is inconsistent.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{Rec}}$ invokes $\mathcal{F}_{\mathsf{Rec}}$ with input $(\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3})$ on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{Rec}}$ with input $\perp$ on behalf of P_0.

Figure 33: Simulator $\mathcal{S}_{\mathsf{Rec}}$ for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 34.

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

Online Phase:

- $\mathcal{S}_{\mathsf{Rec}}$ sends $\lambda_{\mathsf{v},1}$ and $\mathsf{H}(\lambda_{\mathsf{v},1})$ to $\mathcal{A}$ on behalf of P_2 and P_0 respectively.
- $\mathcal{S}_{\mathsf{Rec}}$ receives $\lambda'_{\mathsf{v},3}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{v}})$ from $\mathcal{A}$ on behalf of P_3 and P_0 respectively. $\mathcal{S}_{\mathsf{Rec}}$ sets $\mathsf{flag} = 1$ if any of the received values is inconsistent.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{Rec}}$ invokes $\mathcal{F}_{\mathsf{Rec}}$ with input $(\mathsf{m_v}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3})$ on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{Rec}}$ with input $\perp$ on behalf of P_1.

Figure 34: Simulator $\mathcal{S}_{\mathsf{Rec}}$ for the case of corrupt P_1

d) Multiplication: The ideal functionality realising protocol Π_{Mult} is presented in Fig. 35.

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

$\mathcal{F}_{\mathsf{Mult}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Mult}}$ receives $[\![\cdot]\!]$-shares of values x and y from the parties as input. If $\mathcal{F}_{\mathsf{Mult}}$ receives $\perp$ from $\mathcal{S}$, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Compute $\mathsf{x} = \mathsf{m_x} - \lambda_{\mathsf{x},1} - \lambda_{\mathsf{x},2} - \lambda_{\mathsf{x},3}$, $\mathsf{y} = \mathsf{m_y} - \lambda_{\mathsf{y},1} - \lambda_{\mathsf{y},2} - \lambda_{\mathsf{y},3}$ and set $\mathsf{z} = \mathsf{xy}$. Randomly select $\lambda_{\mathsf{z},1}, \lambda_{\mathsf{z},2}, \lambda_{\mathsf{z},3}$ from $\mathbb{Z}_{2^\ell}$ and set $\mathsf{m_z} = \mathsf{z} + \lambda_{\mathsf{z},1} + \lambda_{\mathsf{z},2} + \lambda_{\mathsf{z},3}$. The output shares are set as:

$$[\![\mathsf{z}]\!]_{P_0} = (\lambda_{\mathsf{z},1}, \lambda_{\mathsf{z},2}, \lambda_{\mathsf{z},3}) \quad [\![\mathsf{z}]\!]_{P_1} = (\mathsf{m_z}, \lambda_{\mathsf{z},2}, \lambda_{\mathsf{z},3})$$

$$[\![\mathsf{z}]\!]_{P_2} = (\mathsf{m_z}, \lambda_{\mathsf{z},3}, \lambda_{\mathsf{z},1}) \quad [\![\mathsf{z}]\!]_{P_3} = (\mathsf{m_z}, \lambda_{\mathsf{z},1}, \lambda_{\mathsf{z},2})$$

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, [\![\mathsf{z}]\!]_{\mathcal{S}})$ to the adversary $\mathcal{S}$, where $[\![\mathsf{z}]\!]_{\mathcal{S}}$ denotes the share of z corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, [\![\mathsf{z}]\!]_i)$, where $[\![\mathsf{z}]\!]_i$ denotes the share of z corresponding to the honest party P_i.

Figure 35: Functionality for protocol Π_{Mult}

The simulator for the case of corrupt P_0 appears in Fig. 36.

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

Offline Phase:

- $\mathcal{S}_{\mathsf{Mult}}$ samples $\lambda_{\mathsf{z},j}$ for $j \in \{1,2,3\}$ honestly using the shared keys obtained.
- $\mathcal{S}_{\mathsf{Mult}}$ emulates $\mathcal{F}_{\mathsf{Zero}}$ functionality and gives the values A, B, and Γ to $\mathcal{A}$.
- $\mathcal{S}_{\mathsf{Mult}}$ receives $\mathsf{H}(\gamma_{\mathsf{xy},1})$, $\mathsf{H}(\gamma_{\mathsf{xy},2})$ and $\mathsf{H}(\gamma_{\mathsf{xy},3})$ from $\mathcal{A}$ on behalf of P_2, P_3 and P_1 respectively. $\mathcal{S}_{\mathsf{Mult}}$ sets $\mathsf{flag} = 1$ if any of the received values is inconsistent.

Online Phase: There is nothing to simulate as P_0 has no role during the online phase.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{Mult}}$ invokes $\mathcal{F}_{\mathsf{Mult}}$ with input $([\![\mathsf{x}]\!]_{P_0}, [\![\mathsf{y}]\!]_{P_0})$ on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{Mult}}$ with input $\perp$ on behalf of P_0.

Figure 36: Simulator for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 37.

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

Offline Phase:

- $\mathcal{S}_{\mathsf{Mult}}$ samples $\lambda_{\mathsf{z},j}$ for $j \in \{2,3\}$ honestly using the shared keys obtained.
- $\mathcal{S}_{\mathsf{Mult}}$ emulates $\mathcal{F}_{\mathsf{Zero}}$ functionality and gives the value A to $\mathcal{A}$.
- $\mathcal{S}_{\mathsf{Mult}}$ sends $\gamma_{\mathsf{xy},3}$ and $\mathsf{H}(\gamma_{\mathsf{xy},3})$ to $\mathcal{A}$ on behalf of P_2 and P_0 respectively. It receives $\gamma_{\mathsf{xy},2}$ from $\mathcal{A}$ on behalf of P_3 and sets $\mathsf{flag} = 1$ if the received value is inconsistent.

Online Phase:

- $\mathcal{S}_{\mathsf{Mult}}$ simulates the computation of m'_{z} shares honestly on behalf of P_2 and P_3.
- $\mathcal{S}_{\mathsf{Mult}}$ sends $\mathsf{m}'_{\mathsf{z},1}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},1})$ to $\mathcal{A}$ on behalf of P_2 and P_3 respectively. It receives $\mathsf{m}'_{\mathsf{z},3}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},2})$ from $\mathcal{A}$ on behalf of P_3 and P_2 respectively. $\mathcal{S}_{\mathsf{Mult}}$ sets $\mathsf{flag} = 1$ if any of the received values is inconsistent.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{Mult}}$ invokes $\mathcal{F}_{\mathsf{Mult}}$ with input $([\![\mathsf{x}]\!]_{P_1}, [\![\mathsf{y}]\!]_{P_1})$ on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{Mult}}$ with input $\perp$ on behalf of P_1.

Figure 37: Simulator for the case of corrupt P_1

e) Dot Product: The ideal functionality realising protocol Π_{DotP} is presented in Fig. 38.

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

$\mathcal{F}_{\mathsf{DotP}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{DotP}}$ receives $[\![\cdot]\!]$-shares of vectors $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$ from the parties as input. Here $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$ are d-length vectors. If $\mathcal{F}_{\mathsf{DotP}}$ receives $\perp$ from $\mathcal{S}$, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Compute $\mathsf{x}_i = \mathsf{m}_{\mathsf{x}_i} - \lambda_{\mathsf{x}_i,1} - \lambda_{\mathsf{x}_i,2} - \lambda_{\mathsf{x}_i,3}$, $\mathsf{y}_i = \mathsf{m}_{\mathsf{y}_i} - \lambda_{\mathsf{y}_i,1} - \lambda_{\mathsf{y}_i,2} - \lambda_{\mathsf{y}_i,3}$ for $i \in [d]$ and set $\mathsf{z} = \sum_{i=1}^{d} \mathsf{x}_i\mathsf{y}_i$. Randomly select $\lambda_{\mathsf{z},1}, \lambda_{\mathsf{z},2}, \lambda_{\mathsf{z},3}$ from $\mathbb{Z}_{2^\ell}$ and set $\mathsf{m_z} = \mathsf{z} + \lambda_{\mathsf{z},1} + \lambda_{\mathsf{z},2} + \lambda_{\mathsf{z},3}$. The output shares are set as:

$$[\![\mathsf{z}]\!]_{P_0} = (\lambda_{\mathsf{z},1}, \lambda_{\mathsf{z},2}, \lambda_{\mathsf{z},3}) \quad [\![\mathsf{z}]\!]_{P_1} = (\mathsf{m_z}, \lambda_{\mathsf{z},2}, \lambda_{\mathsf{z},3})$$

$$[\![\mathsf{z}]\!]_{P_2} = (\mathsf{m_z}, \lambda_{\mathsf{z},3}, \lambda_{\mathsf{z},1}) \quad [\![\mathsf{z}]\!]_{P_3} = (\mathsf{m_z}, \lambda_{\mathsf{z},1}, \lambda_{\mathsf{z},2})$$

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, [\![\mathsf{z}]\!]_{\mathcal{S}})$ to the adversary $\mathcal{S}$, where $[\![\mathsf{z}]\!]_{\mathcal{S}}$ denotes the share of z corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, [\![\mathsf{z}]\!]_i)$, where $[\![\mathsf{z}]\!]_i$ denotes the share of z corresponding to the honest party P_i.

Figure 38: Functionality for protocol Π_{Sh}

The simulator for the case of corrupt P_0 appears in Fig. 39.

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

Offline Phase:

- $\mathcal{S}_{\mathsf{DotP}}$ samples $\lambda_{\mathsf{z},j}$ for $j \in \{1,2,3\}$ honestly using the shared keys obtained.
- $\mathcal{S}_{\mathsf{DotP}}$ emulates $\mathcal{F}_{\mathsf{Zero}}$ functionality and gives the values A, B,

and Γ to $\mathcal{A}$.

– $\mathcal{S}_{\mathsf{DotP}}$ receives $\mathsf{H}(\gamma_{\mathsf{xy},1}), \mathsf{H}(\gamma_{\mathsf{xy},2})$ and $\mathsf{H}(\gamma_{\mathsf{xy},3})$ from $\mathcal{A}$ on behalf of P_2, P_3 and P_1 respectively. $\mathcal{S}_{\mathsf{DotP}}$ sets $\mathsf{flag} = 1$ if any of the received values is inconsistent.

Online Phase: There is nothing to simulate as P_0 has no role during the online phase.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{DotP}}$ invokes $\mathcal{F}_{\mathsf{DotP}}$ with input $(\{[\![\mathsf{x}_i]\!]_{P_0}, [\![\mathsf{y}_i]\!]_{P_0}\}_{i\in[d]})$ on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{DotP}}$ with input $\perp$ on behalf of P_0.

Figure 39: Simulator for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 40.

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

Offline Phase:

– $\mathcal{S}_{\mathsf{DotP}}$ samples $\lambda_{\mathsf{z},j}$ for $j \in \{2,3\}$ honestly using the shared keys obtained.

– $\mathcal{S}_{\mathsf{DotP}}$ emulates $\mathcal{F}_{\mathsf{Zero}}$ functionality and gives the value A to $\mathcal{A}$.

– $\mathcal{S}_{\mathsf{DotP}}$ sends $\gamma_{\mathsf{xy},3}$ and $\mathsf{H}(\gamma_{\mathsf{xy},3})$ to $\mathcal{A}$ on behalf of P_2 and P_0 respectively. It receives $\gamma_{\mathsf{xy},2}$ from $\mathcal{A}$ on behalf of P_3 and sets $\mathsf{flag} = 1$ if the received value is inconsistent.

Online Phase:

– $\mathcal{S}_{\mathsf{DotP}}$ simulates the computation of m'_{z} shares honestly on behalf of P_2 and P_3.

– $\mathcal{S}_{\mathsf{DotP}}$ sends $\mathsf{m}'_{\mathsf{z},1}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},1})$ to $\mathcal{A}$ on behalf of P_2 and P_3 respectively. It receives $\mathsf{m}'_{\mathsf{z},3}$ and $\mathsf{H}(\mathsf{m}'_{\mathsf{z},2})$ from $\mathcal{A}$ on behalf of P_3 and P_2 respectively. $\mathcal{S}_{\mathsf{DotP}}$ sets $\mathsf{flag} = 1$ if any of the received values is inconsistent.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{DotP}}$ invokes $\mathcal{F}_{\mathsf{DotP}}$ with input $(\{[\![\mathsf{x}_i]\!]_{P_1}, [\![\mathsf{y}_i]\!]_{P_1}\}_{i\in[d]})$ on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{DotP}}$ with input $\perp$ on behalf of P_1.

Figure 40: Simulator for the case of corrupt P_1

f) Bit to Arithmetic Sharing (Bit2A): The ideal functionality realising protocol Π_{Bit2A} is presented in Fig. 41.

Functionality $\mathcal{F}_{\mathsf{Bit2A}}$

$\mathcal{F}_{\mathsf{Bit2A}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Bit2A}}$ receives $[\![\cdot]\!]^{\mathbf{B}}$-share of a bit b from the parties as input. If $\mathcal{F}_{\mathsf{Bit2A}}$ receives $\perp$ from $\mathcal{S}$, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Compute $\mathsf{b} = \mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b},1} \oplus \lambda_{\mathsf{b},2} \oplus \lambda_{\mathsf{b},3}$. Let b' denotes the value of bit b over an arithmetic ring $\mathbb{Z}_{2^\ell}$. Randomly select $\lambda'_{\mathsf{b},1}, \lambda'_{\mathsf{b},2}, \lambda'_{\mathsf{b},3}$ from $\mathbb{Z}_{2^\ell}$ and set $\mathsf{m}'_{\mathsf{b}} = \mathsf{b}' + \lambda'_{\mathsf{b},1} + \lambda'_{\mathsf{b},2} + \lambda'_{\mathsf{b},3}$. The output shares are set as:

$$[\![\mathsf{b}]\!]'_{P_0} = (\lambda'_{\mathsf{b},1}, \lambda'_{\mathsf{b},2}, \lambda'_{\mathsf{b},3}) \quad [\![\mathsf{b}]\!]'_{P_1} = (\mathsf{m}'_{\mathsf{b}}, \lambda'_{\mathsf{b},2}, \lambda'_{\mathsf{b},3})$$

$$[\![\mathsf{b}]\!]'_{P_2} = (\mathsf{m}'_{\mathsf{b}}, \lambda'_{\mathsf{b},3}, \lambda'_{\mathsf{b},1}) \quad [\![\mathsf{b}]\!]'_{P_3} = (\mathsf{m}'_{\mathsf{b}}, \lambda'_{\mathsf{b},1}, \lambda'_{\mathsf{b},2})$$

Output to adversary: If $\mathcal{S}$ sends $\mathtt{abort}$, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, [\![\mathsf{b}']\!]_{\mathcal{S}})$ to the adversary $\mathcal{S}$, where $[\![\mathsf{b}']\!]_{\mathcal{S}}$ denotes the share of b' corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, [\![\mathsf{b}']\!]_i)$, where $[\![\mathsf{b}']\!]_i$ denotes the share of b' corresponding to the honest party P_i.

Figure 41: Functionality for protocol Π_{Bit2A}

The simulator for the case of corrupt P_0 appears in Fig. 42.

Simulator $\mathcal{S}_{\mathsf{Bit2A}}$

Offline Phase:

– Corresponding to the invocation of $\Pi_{\mathsf{aSh}}(P_0, \mathsf{u})$ protocol, $\mathcal{S}_{\mathsf{Bit2A}}$ receives v_3 from $\mathcal{A}$ on behalf of both P_1 and P_2. $\mathcal{S}_{\mathsf{Bit2A}}$ sets $\mathsf{flag} = 1$ if the received values mismatch.

– $\mathcal{S}_{\mathsf{Bit2A}}$ performs the check honestly and sets $\mathsf{flag} = 1$ if the verification fails.

Online Phase: The steps corresponding to Π_{vSh} and Π_{Mult} are simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 30) and $\mathcal{S}_{\mathsf{Mult}}$ (Fig. 36) respectively, for the case of corrupt P_0.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{Bit2A}}$ invokes $\mathcal{F}_{\mathsf{Bit2A}}$ with input $([\![\mathsf{b}]\!]^{\mathbf{B}}_{P_0})$ on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{Bit2A}}$ with input $\perp$ on behalf of P_0.

Figure 42: Simulator for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 43.

Simulator $\mathcal{S}_{\mathsf{Bit2A}}$

Offline Phase:

– Corresponding to the invocation of $\Pi_{\mathsf{aSh}}(P_0, \mathsf{u})$ protocol, $\mathcal{S}_{\mathsf{Bit2A}}$ sends v_3 to $\mathcal{A}$ on behalf of P_0. It then sends $\mathsf{H}(\mathsf{v}_3)$ to $\mathcal{A}$ on behalf of P_2 and receives $\mathsf{H}(\mathsf{v}'_3)$ back. $\mathcal{S}_{\mathsf{Bit2A}}$ sets $\mathsf{flag} = 1$ if it receives inconsistent hash value.

– $\mathcal{S}_{\mathsf{Bit2A}}$ receives $(\mathsf{x}_1, \mathsf{y}_1)$ from $\mathcal{A}$ on behalf of P_3, performs the check honestly and sets $\mathsf{flag} = 1$ if the verification fails.

Online Phase: The steps corresponding to Π_{vSh} and Π_{Mult} are simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 31) and $\mathcal{S}_{\mathsf{Mult}}$ (Fig. 37) respectively, for the case of corrupt P_1.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{Bit2A}}$ invokes $\mathcal{F}_{\mathsf{Bit2A}}$ with input $([\![\mathsf{b}]\!]^{\mathbf{B}}_{P_1})$ on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{Bit2A}}$ with input $\perp$ on behalf of P_1.

Figure 43: Simulator for the case of corrupt P_1

g) Bit Injection (BitInj): The ideal functionality realising protocol Π_{BitInj} is presented in Fig. 44.

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

$\mathcal{F}_{\mathsf{BitInj}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{BitInj}}$ receives $([\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{v}]\!])$ from the parties as input. If $\mathcal{F}_{\mathsf{BitInj}}$ receives $\perp$ from $\mathcal{S}$, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Compute $\mathsf{b} = \mathsf{m_b} \oplus \lambda_{\mathsf{b},1} \oplus \lambda_{\mathsf{b},2} \oplus \lambda_{\mathsf{b},3}$, $\mathsf{v} = \mathsf{m_v} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2} - \lambda_{\mathsf{v},3}$ and set $(\mathsf{bv}) = \mathsf{b} \cdot \mathsf{v}$. Randomly select $\lambda_{(\mathsf{bv}),1}, \lambda_{(\mathsf{bv}),2}, \lambda_{(\mathsf{bv}),3}$ from $\mathbb{Z}_{2^\ell}$ and set $\mathsf{m}_{(\mathsf{bv})} = (\mathsf{bv}) + \lambda_{(\mathsf{bv}),1} + \lambda_{(\mathsf{bv}),2} + \lambda_{(\mathsf{bv}),3}$. The output shares are set as:

$$[\![(\mathsf{bv})]\!]_{P_0} = (\lambda_{(\mathsf{bv}),1}, \lambda_{(\mathsf{bv}),2}, \lambda_{(\mathsf{bv}),3})$$
$$[\![(\mathsf{bv})]\!]_{P_1} = (\mathsf{m}_{(\mathsf{bv})}, \lambda_{(\mathsf{bv}),2}, \lambda_{(\mathsf{bv}),3})$$
$$[\![(\mathsf{bv})]\!]_{P_2} = (\mathsf{m}_{(\mathsf{bv})}, \lambda_{(\mathsf{bv}),3}, \lambda_{(\mathsf{bv}),1})$$
$$[\![(\mathsf{bv})]\!]_{P_3} = (\mathsf{m}_{(\mathsf{bv})}, \lambda_{(\mathsf{bv}),1}, \lambda_{(\mathsf{bv}),2})$$

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, [\![(\mathsf{bv})]\!]_{\mathcal{S}})$ to the adversary $\mathcal{S}$, where $[\![(\mathsf{bv})]\!]_{\mathcal{S}}$ denotes the share of (bv) corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, [\![(\mathsf{bv})]\!]_i)$, where $[\![(\mathsf{bv})]\!]_i$ denotes the share of (bv) corresponding to the honest party P_i.

Figure 44: Functionality for protocol Π_{BitInj}

The simulator for the case of corrupt P_0 appears in Fig. 45.

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

Offline Phase:

- Corresponding to the invocation of $\Pi_{\mathsf{aSh}}(P_0, \mathsf{y}_j)$ protocol, $\mathcal{S}_{\mathsf{BitInj}}$ receives $\mathsf{y}_{j,3}$ from $\mathcal{A}$ on behalf of both P_1 and P_2. $\mathcal{S}_{\mathsf{BitInj}}$ sets $\mathsf{flag} = 1$ if the received values mismatch.
- $\mathcal{S}_{\mathsf{BitInj}}$ performs the check honestly and sets $\mathsf{flag} = 1$ if the verification fails.

Online Phase: The steps corresponding to Π_{vSh} are simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 30), for the case of corrupt P_0.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{BitInj}}$ invokes $\mathcal{F}_{\mathsf{BitInj}}$ with input $([\![\mathsf{b}]\!]^{\mathbf{B}}_{P_0}, [\![\mathsf{v}]\!]_{P_0})$ on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{BitInj}}$ with input $\perp$ on behalf of P_0.

Figure 45: Simulator for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 46.

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

Offline Phase:

- Corresponding to the invocation of $\Pi_{\mathsf{aSh}}(P_0, \mathsf{u})$ protocol, $\mathcal{S}_{\mathsf{BitInj}}$ sends $\mathsf{y}_{j,3}$ to $\mathcal{A}$ on behalf of P_0. It then sends $\mathsf{H}(\mathsf{y}_{j,3})$ to $\mathcal{A}$ on behalf of P_2 and receives $\mathsf{H}(\mathsf{y}'_{j,3})$ back. $\mathcal{S}_{\mathsf{BitInj}}$ sets $\mathsf{flag} = 1$ if it receives inconsistent hash value.
- $\mathcal{S}_{\mathsf{BitInj}}$ emulates $\mathcal{F}_{\mathsf{Zero}}$ functionality and gives A to $\mathcal{A}$.
- $\mathcal{S}_{\mathsf{BitInj}}$ receives z_2 from $\mathcal{A}$ on behalf of P_3, performs the check honestly and sets $\mathsf{flag} = 1$ if the verification fails.

Online Phase: The steps corresponding to Π_{vSh} are simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 31), for the case of corrupt P_1.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{BitInj}}$ invokes $\mathcal{F}_{\mathsf{BitInj}}$ with input $([\![\mathsf{b}]\!]^{\mathbf{B}}_{P_1}, [\![\mathsf{v}]\!]_{P_1})$ on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{BitInj}}$ with input $\perp$ on behalf of P_1.

Figure 46: Simulator for the case of corrupt P_1

h) Multiplication with Truncation: The ideal functionality realising protocol Π_{MultTr} is presented in Fig. 47.

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

$\mathcal{F}_{\mathsf{MultTr}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{MultTr}}$ receives $[\![\cdot]\!]$-shares of values x and y from the parties as input. If $\mathcal{F}_{\mathsf{MultTr}}$ receives $\perp$ from $\mathcal{S}$, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Compute $\mathsf{x} = \mathsf{m_x} - \lambda_{\mathsf{x},1} - \lambda_{\mathsf{x},2} - \lambda_{\mathsf{x},3}$, $\mathsf{y} = \mathsf{m_y} - \lambda_{\mathsf{y},1} - \lambda_{\mathsf{y},2} - \lambda_{\mathsf{y},3}$ and set $\mathsf{z}^{\mathsf{t}} = (\mathsf{xy})^{\mathsf{t}}$. Randomly select $\lambda_{\mathsf{z}^{\mathsf{t}},1}, \lambda_{\mathsf{z}^{\mathsf{t}},2}, \lambda_{\mathsf{z}^{\mathsf{t}},3}$ from $\mathbb{Z}_{2^\ell}$ and set $\mathsf{m}_{\mathsf{z}^{\mathsf{t}}} = \mathsf{z}^{\mathsf{t}} + \lambda_{\mathsf{z}^{\mathsf{t}},1} + \lambda_{\mathsf{z}^{\mathsf{t}},2} + \lambda_{\mathsf{z}^{\mathsf{t}},3}$. The output shares are set as:

$$[\![\mathsf{z}^{\mathsf{t}}]\!]_{P_0} = (\lambda_{\mathsf{z}^{\mathsf{t}},1}, \lambda_{\mathsf{z}^{\mathsf{t}},2}, \lambda_{\mathsf{z}^{\mathsf{t}},3}) \qquad [\![\mathsf{z}^{\mathsf{t}}]\!]_{P_1} = (\mathsf{m}_{\mathsf{z}^{\mathsf{t}}}, \lambda_{\mathsf{z}^{\mathsf{t}},2}, \lambda_{\mathsf{z}^{\mathsf{t}},3})$$
$$[\![\mathsf{z}^{\mathsf{t}}]\!]_{P_2} = (\mathsf{m}_{\mathsf{z}^{\mathsf{t}}}, \lambda_{\mathsf{z}^{\mathsf{t}},3}, \lambda_{\mathsf{z}^{\mathsf{t}},1}) \qquad [\![\mathsf{z}^{\mathsf{t}}]\!]_{P_3} = (\mathsf{m}_{\mathsf{z}^{\mathsf{t}}}, \lambda_{\mathsf{z}^{\mathsf{t}},1}, \lambda_{\mathsf{z}^{\mathsf{t}},2})$$

Output to adversary: If $\mathcal{S}$ sends `abort`, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, [\![\mathsf{z}^{\mathsf{t}}]\!]_{\mathcal{S}})$ to the adversary $\mathcal{S}$, where $[\![\mathsf{z}^{\mathsf{t}}]\!]_{\mathcal{S}}$ denotes the share of z^{t} corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, [\![\mathsf{z}^{\mathsf{t}}]\!]_i)$, where $[\![\mathsf{z}^{\mathsf{t}}]\!]_i$ denotes the share of z^{t} corresponding to the honest party P_i.

Figure 47: Functionality for protocol Π_{MultTr}

The simulator for the case of corrupt P_0 appears in Fig. 48.

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

Offline Phase:

- The steps corresponding to offline phase of Π_{Mult} are simulated similar to offline phase of $\mathcal{S}_{\mathsf{Mult}}$ (Fig. 36), for the case of corrupt P_0.
- Corresponding to the invocation of $\Pi_{\mathsf{aSh}}(P_0, \mathsf{r}^{\mathsf{t}})$ protocol, $\mathcal{S}_{\mathsf{MultTr}}$ receives $\mathsf{r}^{\mathsf{t}}_3$ from $\mathcal{A}$ on behalf of both P_1 and P_2. $\mathcal{S}_{\mathsf{MultTr}}$ sets $\mathsf{flag} = 1$ if the received values mismatch.
- $\mathcal{S}_{\mathsf{MultTr}}$ performs the check honestly and sets $\mathsf{flag} = 1$ if the verification fails.

Online Phase: There is nothing to simulate as P_0 has no role during the online phase.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{MultTr}}$ invokes $\mathcal{F}_{\mathsf{MultTr}}$ with input $([\![\mathsf{x}]\!]_{P_0}, [\![\mathsf{y}]\!]_{P_0})$ on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{MultTr}}$ with input $\perp$ on behalf of P_0.

Figure 48: Simulator for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 49.

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

Offline Phase:

- The steps corresponding to offline phase of Π_{Mult} are simulated similar to offline phase of $\mathcal{S}_{\mathsf{Mult}}$ (Fig. 37), for the case of corrupt P_1.
- Corresponding to the invocation of $\Pi_{\mathsf{aSh}}(P_0, \mathsf{r}^{\mathsf{t}})$ protocol, $\mathcal{S}_{\mathsf{MultTr}}$ sends $\mathsf{r}_3^{\mathsf{t}}$ to $\mathcal{A}$ on behalf of P_0. It then sends $\mathsf{H}(\mathsf{r}_3^{\mathsf{t}})$ to $\mathcal{A}$ on behalf of P_2 and receives $\mathsf{H}(\mathsf{r}_3^{\mathsf{t}\prime})$ back. $\mathcal{S}_{\mathsf{MultTr}}$ sets $\mathsf{flag} = 1$ if it receives inconsistent hash value.
- $\mathcal{S}_{\mathsf{MultTr}}$ receives $(\mathsf{m}_1, \mathsf{H}(\mathsf{c}))$ from $\mathcal{A}$ on behalf of P_2, performs the check honestly and sets $\mathsf{flag} = 1$ if the verification fails.

Online Phase:

- $\mathcal{S}_{\mathsf{MultTr}}$ simulates the computation of m_{z}' shares honestly on behalf of P_2 and P_3.
- $\mathcal{S}_{\mathsf{MultTr}}$ sends $\mathsf{m}_{\mathsf{z},1}'$ and $\mathsf{H}(\mathsf{m}_{\mathsf{z},1}')$ to $\mathcal{A}$ on behalf of P_2 and P_3 respectively. It receives $\mathsf{m}_{\mathsf{z},3}'$ and $\mathsf{H}(\mathsf{m}_{\mathsf{z},2}')$ from $\mathcal{A}$ on behalf of P_3 and P_2 respectively. $\mathcal{S}_{\mathsf{MultTr}}$ sets $\mathsf{flag} = 1$ if any of the received values is inconsistent.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{MultTr}}$ invokes $\mathcal{F}_{\mathsf{MultTr}}$ with input $(\llbracket \mathsf{x} \rrbracket_{P_1}, \llbracket \mathsf{y} \rrbracket_{P_1})$ on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{MultTr}}$ with input $\perp$ on behalf of P_1.

Figure 49: Simulator for the case of corrupt P_1

i) Secure Comparison: The ideal functionality realising protocol Π_{BitExt} is presented in Fig. 50.

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

$\mathcal{F}_{\mathsf{BitExt}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{BitExt}}$ receives $\llbracket \cdot \rrbracket$-share of value v from the parties as input. If $\mathcal{F}_{\mathsf{BitExt}}$ receives $\perp$ from $\mathcal{S}$, then send $\perp$ to every party, else proceed with the computation.

Computation of output: Compute $\mathsf{v} = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2} - \lambda_{\mathsf{v},3}$ and set $\mathsf{b} = \mathsf{msb}(\mathsf{v})$ where msb denotes the most significant bit. Randomly select $\lambda_{\mathsf{b},1}, \lambda_{\mathsf{b},2}, \lambda_{\mathsf{b},3}$ from $\mathbb{Z}_{2^1}$ and set $\mathsf{m}_{\mathsf{b}} = \mathsf{b} \oplus \lambda_{\mathsf{b},1} \oplus \lambda_{\mathsf{b},2} \oplus \lambda_{\mathsf{b},3}$. The output shares are set as:

$$\llbracket \mathsf{b} \rrbracket^{\mathbf{B}}_{P_0} = (\lambda_{\mathsf{b},1}, \lambda_{\mathsf{b},2}, \lambda_{\mathsf{b},3}) \quad \llbracket \mathsf{b} \rrbracket^{\mathbf{B}}_{P_1} = (\mathsf{m}_{\mathsf{b}}, \lambda_{\mathsf{b},2}, \lambda_{\mathsf{b},3})$$

$$\llbracket \mathsf{b} \rrbracket^{\mathbf{B}}_{P_2} = (\mathsf{m}_{\mathsf{b}}, \lambda_{\mathsf{b},3}, \lambda_{\mathsf{b},1}) \quad \llbracket \mathsf{b} \rrbracket^{\mathbf{B}}_{P_3} = (\mathsf{m}_{\mathsf{b}}, \lambda_{\mathsf{b},1}, \lambda_{\mathsf{b},2})$$

Output to adversary: If $\mathcal{S}$ sends $\mathtt{abort}$, then send $(\mathsf{Output}, \perp)$ to all the parties. Otherwise, send $(\mathsf{Output}, \llbracket \mathsf{b} \rrbracket_{\mathcal{S}})$ to the adversary $\mathcal{S}$, where $\llbracket \mathsf{b} \rrbracket_{\mathcal{S}}$ denotes the share of b corresponding to the corrupt party.

Output to selected honest parties: Receive $(\mathsf{select}, \{I\})$ from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party P_i belongs to I, send $(\mathsf{Output}, \perp)$, else send $(\mathsf{Output}, \llbracket \mathsf{b} \rrbracket_i)$, where $\llbracket \mathsf{b} \rrbracket_i$ denotes the share of b corresponding to the honest party P_i.

Figure 50: Functionality for protocol Π_{BitExt}

The simulator for the case of corrupt P_0 appears in Fig. 51.

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

Offline Phase:

- $\mathcal{S}_{\mathsf{BitExt}}$ samples a random r on behalf of P_1, P_2 and set $\mathsf{x} = \mathsf{msb}(\mathsf{r})$.
- The steps corresponding to Π_{vSh} are simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 30), for the case of corrupt P_0.

Online Phase:

- The steps corresponding to Π_{Mult} and Π_{Rec} are simulated similar to $\mathcal{S}_{\mathsf{Mult}}$ (Fig. 36) and $\mathcal{S}_{\mathsf{Rec}}$ (Fig. 33) respectively, for the case of corrupt P_0.
- The steps corresponding to Π_{vSh} is simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 30), for the case of corrupt P_0.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{BitExt}}$ invokes $\mathcal{F}_{\mathsf{BitExt}}$ with input $(\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3})$ on behalf of P_0. Else it invokes $\mathcal{F}_{\mathsf{BitExt}}$ with input $\perp$ on behalf of P_0.

Figure 51: Simulator for the case of corrupt P_0

The simulator for the case of corrupt P_1 appears in Fig. 52.

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

Offline Phase:

- $\mathcal{S}_{\mathsf{BitExt}}$ samples a random r on behalf of P_2 and set $\mathsf{x} = \mathsf{msb}(\mathsf{r})$.
- The steps corresponding to Π_{vSh} are simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 31), for the case of corrupt P_1.

Online Phase:

- The steps corresponding to Π_{Mult} and Π_{Rec} are simulated similar to $\mathcal{S}_{\mathsf{Mult}}$ (Fig. 37) and $\mathcal{S}_{\mathsf{Rec}}$ (Fig. 34) respectively, for the case of corrupt P_1.
- The steps corresponding to Π_{vSh} is simulated similar to $\mathcal{S}_{\mathsf{vSh}}$ (Fig. 31), for the case of corrupt P_1.

If $\mathsf{flag} = 0$, $\mathcal{S}_{\mathsf{BitExt}}$ invokes $\mathcal{F}_{\mathsf{BitExt}}$ with input $(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v},2}, \lambda_{\mathsf{v},3})$ on behalf of P_1. Else it invokes $\mathcal{F}_{\mathsf{BitExt}}$ with input $\perp$ on behalf of P_1.

Figure 52: Simulator for the case of corrupt P_0