A Robust PPML Framework for Three Servers

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Faculty of Engineering

BY

Mahak Pancholi

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

July, 2020

Declaration of Originality

I, **Mahak Pancholi**, with SR No. **04-04-00-10-42-18-1-15493** hereby declare that the material presented in the thesis titled

A Robust PPML Framework for Three Servers

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-2020**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

© Mahak Pancholi
July, 2020
All rights reserved

DEDICATED TO

My Dear

Ma

Acknowledgements

Firstly, I thank my advisor, Dr. Arpita Patra, for being a great research mentor and guiding me throughout my journey here at IISc. Her enthusiasm for research is a constant source of inspiration for me. I would also like to extend my gratitude towards my incredible labmates. They are truly a dream team; understanding, supportive, pushing me to do my best. A shout out to Ajith and Nishat, my research buddies and semi-mentors. I thank you for the great brainstorming sessions and for teaching me everything I know about research. I want to thank Divya for being like a big sister to me, and Protik for being the calm presence in my life. His aplomb and support has got me through some tough spots. And to all my friends, without whom it would not have been so much fun, Pratheek, Pooja, Stanly, Swapnil, Harsh, I cannot thank you enough. Finally, I would like to thank my Mum and my brother for believing in me and never letting me settle for the ordinary.

Abstract

Machine Learning (ML) techniques are integral tools in many diverse fields from health care to self driving auto-mobiles. Consequently, performing ML tasks while maintaining data privacy, i.e Privacy Preserving Machine Learning, is an emergent field of research. To circumvent the high computing power generally required for PPML tasks, there has been a visible shift towards adoption of Secure Outsourced Computation (SOC), which allows computation to be outsourced on a set of external servers in a secure way. The services if such servers can be availed at pay-per-basis. In this work we propose a *robust* PPML framework for a range of ML algorithms in SOC setting, that guarantees output delivery to the users irrespective of any adversarial behaviour. Robustness is a highly desirable feature as it evokes user participation without the fear of denial of service. Our framework relies on a highly efficient, maliciously-secure, three-party computation (3PC) over rings that provides guaranteed output delivery (GOD) in the honest-majority setting. To the best of our knowledge, our work is the first robust PPML framework in 3PC setting. The state-of-the-art work in the same setting, BLAZE, only provides fairness, i.e it ensures either all or none receive the output, whereas GOD ensures guaranteed output delivery no matter what. Moreover, we do this with an overhead of just one element in the preprocessing phase, while the communication cost remains the same in the online phase. We also demonstrate the practical efficiency by benchmarking two important applications– i) ML algorithms: Logistic Regression and Neural Network, and ii) Biometric matching, both over a 64-bit ring in WAN setting.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning (ML) has enjoyed a boom in interest as advancements in the area have wide-scale, and far reaching applications in real life. More interesting and useful applications of ML are found in critical and impactful fields such as health care wherein ML tools are increasingly being sought out to aide experts in medical diagnosis. However, the wide scale deployment of ML as a service is hampered by many challenges. First and foremost, is the requirement of high accuracy of the algorithms which is necessary in mission critical fields such as health care. Accuracy can generally be improved by availability of data from varied sources. This entails sharing of information and data between organisations which might not be possible either because of policies like the European Union General Data Protection Regulation (GDPR), or because of the sensitive nature of the data as is the case with medical and financial records. Another concern for the service provider is the leakage of model parameters rendering its services redundant. Hence, the crucial requirement of security and privacy accompanied with the huge interest in ML has led to flourishing research in the field of privacy-preserving machine learning (PPML). Wide spread adoption of ML tools is also inhibited by the high computational demands of the ML algorithms. PPML, furthermore, makes to the already compute-intensive ML algorithms even more demanding. Many everyday users lack the needed computing resources to obtain high accuracy. It is more preferable to outsource an ML task to a better equipped, and more powerful server. However, at the same time, the privacy of the data should not be compromised. Towards this, Secure Outsourced Computation (SOC) promises to provide a viable solution. It allows end-users to securely outsource computation to a set of specialized cloud servers and avail its services on a pay-per-use basis, guaranteeing that individual data of the end-users remain private, while tolerating reasonable collusion amongst the servers.

Realisation of PPML in an SOC setting, can be done by relying on techniques form Secure

Multiparty Computation (MPC), which allows n mutually distrusting parties to perform computations together on their private inputs, so that an adversary controlling at most t parties, cannot learn any information beyond what is already known and allowed by the computation. MPC for a small number of parties in the honest majority setting, specifically the setting of 3 parties (3PC) with one corruption, has become popular over the last few years, as it allows highly efficient constructions that use only light-weight primitives [2, 3, 23, 29, 14, 33, 26, 10, 7]. The setting of 3PC with honest majority also allows strong security guarantees such as fairness (the adversary gets the output if and only if the honest parties do), and in the presence of a broadcast channel Guaranteed Output Delivery (GOD) (all parties obtain the output irrespective of adversary's behaviour).

1.1 Related Work

MPC protocols for a small population can be cast into two domains: low latency protocols [35, 10, 11], and high throughput protocols [6, 2, 3, 23, 14, 12, 34, 36, 8, 1, 21]. In the 3PC setting, [2, 12] provide efficient semi-honest protocols, wherein ASTRA [12] improved upon [2] by casting the protocols in the preprocessing model and provided a fast online phase. It further provided security with fairness in the malicious setting with an improved online phase compared to [3]. Later, a maliciously-secure 3PC protocol based on distributed zero-knowledge techniques was proposed by [7] providing abort security. Further, building on [7] and enhancing the security to GOD, Boyle et al. [8] proposed a concretely efficient 3PC protocol with an amortized communication cost of 3 field elements (can be extended to work over rings) per multiplication gate. Concurrently, BLAZE [36] provided a fair protocol in the preprocessing model, which required communicating 3 ring elements in each phase. However, BLAZE eliminated the reliance on the computationally intensive distributed zero-knowledge system (whose efficiency kicks in for large circuit or many multiplication gates) from the online phase and pushed it to the preprocessing phase. This resulted in a faster online phase compared to [8].

In the PPML domain, MPC has been used for various ML algorithms such as Decision Trees [30], Linear Regression [20, 37], k-means clustering [27, 9], SVM Classification [41, 39], Logistic Regression [38]. In the 3PC SOC setting, the works of ABY3 [31] and SecureNN [40], provide security with abort. This was followed by ASTRA [12], which improves upon ABY3 and achieves security with fairness. ASTRA presents primitives to build protocols for Linear Regression and Logistic Regression inference. Recently, BLAZE improves over the efficiency of ASTRA and additionally tackles training for the above ML tasks, which requires building additional PPML building blocks, such as truncation and bit to arithmetic conversions.

1.2 Robustness in Machine Learning

In this work, we strongly motivate the need for robustness, i.e guaranteed output delivery, over fairness in the domain of PPML. Robustness provides the guarantee of output delivery to all protocol participants, no matter how the adversary misbehaves. It becomes extremely crucial for real world deployment and usage of PPML techniques. Consider the following scenario wherein an ML model owner wishes to provide inference service. The model owner shares the model parameters between the servers, while the end-users share their queries. A protocol that provides security with abort or fairness will not suffice as in both the cases the adversary (controlling one of the servers maliciously) can act in a way so that the protocol results in an abort which means that the user will not get the desired output. This leads to denial of service and heavy economic losses for the service provider. From the point of view of data providers who want to collaboratively build a model on their data, more training data leads to a better, more accurate model, which enables them to provide better ML services and, consequently, attract more clients. A robust framework encourages active involvement from multiple data providers. Hence, for seamless adoption of PPML solutions in real world, robustness of the protocol is of utmost importance. The prior result of [15] suggests that an honest-majority amongst the servers is necessary to achieve robustness. We are interested in the smallest possible honest majority setting of three servers with one corruption (3PC).

1.2.1 Computation over Rings

We choose to build our protocols over rings. Generally, computer architectures have their primitive data-types over 32 and 64 bit rings. As a result, to boost efficiency of arithmetic computation, they have specialised optimizations for such data types. A protocol designed to work for rings can fully leverage these optimizations. Many prior works, in a bid to boost efficiency, are designed over rings [6, 18, 3, 21, 12] as opposed to fields, which are usually 10-20x slower since they have to rely on external libraries. Hence, our protocol too is well suited for implementation in the real-world architectures.

1.3 Our Contribution

We propose an efficient PPML framework in the outsourced computation with 3 servers, at most one of which can be maliciously corrupt, via secure MPC over rings $\mathbb{Z}_{2^\ell}$. Concretely, we provide all required building blocks for PPML with respect to logistic regression (training and prediction) and neural networks (prediction), such as dot-product, truncation, bit extraction (given arithmetic sharing of a value v, this is used to generate boolean sharing of the most

significant bit (msb) of the value), bit to arithmetic sharing conversion (converts the boolean sharing of a single bit value to its arithmetic sharing), bit injection (computes the arithmetic sharing of b·v, given the boolean sharing of a bit b and the arithmetic sharing of a ring element v).

To achieve GOD guarantees, we introduce a new primitive called Joint Message Passing (jmp) that allows two servers to relay a common message to the third server such that either the relay is successful or an honest server (alternatively, a conflicting pair) is identified. Interestingly, jmp requires communication of just one element (amortized) per relay. Without any extra cost, it allows us to replace several private communications, that may lead to abort, either because the malicious sender does not send anything or sends a wrong message.

We leverage jmp in design of a new 3PC multiplication protocol that provides GOD, with an additional cost of just one element in the processing phase compared to the state-of-the-art protocol of BLAZE [36], and the same online cost. Since our multiplication protocol is built in the preprocessing model, the cost of heavy machinery for distributed zero-knowledge proofs, as used in [8, 7], is pushed to the preprocessing phase, leaving the online phase computationally light. The cost of zero-knowledge proofs in the aforementioned works gets amortized only for large circuits which are unlikely in the online phase of PPML. However, as the main task of the preprocessing phase is to generate random multiplication triples, which can be done in batches of millions, the cost of zero-knowledge machinery gets amortized. We also provide an improved and more efficient protocol for input sharing and output reconstruction for an SOC setting. We then modify the protocols for PPML building blocks in order to exploit jmp and obtain GOD. The comparison of concrete communication complexity with the closest related work (BLAZE), appears in Table 1.1. Finally, we demonstrate the practicality of our protocols by benchmarking Biometric Matching and PPML. For the latter, Logistic Regression (training and inference) and Neural Networks (inference) are considered. The NN training requires mixed-world conversions [32, 13, 19], which we leave as future work.

Building Blocks	Ref.	Pre.	Online		Security
		Comm. (ℓ)	Rounds	Comm. (ℓ)	
Multiplication	[7]	1	1	2	Abort
	[8]	-	3	3	GOD
	BLAZE	3	1	3	Fair
	This	**4**	**1**	**3**	GOD
Dot Product	BLAZE	$3\mathsf{n}$	1	3	Fair
	This	$\mathbf{3n+1}$	**1**	**3**	GOD
Dot Product with Truncation	BLAZE	$3\mathsf{n}+2$	1	3	Fair
	This	$\mathbf{3n+3}$	**1**	**3**	GOD
Bit Extraction	BLAZE	9	$1+\log \ell$	9	Fair
	This	**12**	$\mathbf{1}+\log \ell$	**9**	GOD
Bit to Arithmetic	BLAZE	9	1	4	Fair
	This	**11**	**1**	**4**	GOD
Bit Injection	BLAZE	12	2	7	Fair
	This	**15**	**2**	**7**	GOD

Table 1.1: Comparison of this work with its closest competitors in terms of Communication and Round Complexity

1.4 Organization of Thesis

The rest of the thesis is organised as follows:

1. In chapter 2 we introduce our system model, preliminaries and notations used. We also briefly describe well defined primitives like collision resistant hash function, commitment scheme, etc.

2. In chapter 3 we give the details of our constructs in the 3PC setting. This includes a new and crucial primitive: Joint Message Passing(jmp). This primitive is central to all of our constructions. This is followed by the secret-sharing schemes, addition protocol and a new multiplication protocol. Together these protocols describe how an arithmetic can be securely and robustly evaluated in a 3PC setting.

3. The details of the essential ML building blocks such as dot product, MSB extraction, truncation, etc required for a robust machine learning framework are included chapter 4.

4. Chapter 5 includes the benchmarking results for our framework for popular applications of Biometric Matching and PPML. For the latter, Logistic Regression (training and inference) and Neural Networks (inference) are considered.

Chapter 2

Preliminaries

We consider a set of three servers $\mathcal{P} = \{P_0, P_1, P_2\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, malicious adversary that can corrupt at most one server. We use a broadcast channel for 3PC which is inevitable [16]. The function to be computed is expressed as a public circuit ckt, and is evaluated over an arithmetic ring $\mathbb{Z}_{2^\ell}$ or boolean ring $\mathbb{Z}_2$. To deal with floating-point values, we use Fixed-Point Arithmetic (FPA)[32, 31, 12, 11, 13, 36] representation in which a decimal value is represented as an ℓ-bit integer in signed 2's complement representation. The most significant bit (msb) represents the sign bit and x least significant bits are reserved for the fractional part. The ℓ-bit integer is then treated as an element of $\mathbb{Z}_{2^\ell}$ and operations are performed modulo 2ℓ.We set $\ell = 64$ and $x = 13$, leaving $\ell - x - 1$ bits for the integral part.

2.1 GOD functionality

Guaranteed Output Delivery allows all the parties to compute the output irrespective of the behaviour of the adversary. We prove the security of our protocols in the standard real/ideal world paradigm in which we compare the view of the adversary in the real world and ideal world. The ideal world execution proceeds as follows. It consists of a set of three servers, an ideal adversary $\mathcal{S}$ who can corrupt at most one server, and a functionality f, which is efficiently represented as an arithmetic circuit ckt. The servers want to compute a publicly known function f on their secret inputs. Each server sends its input to an incorruptible trusted third party (TTP), who computes the desired function f on these inputs and sends the output back to the parties. The real world execution involves the servers executing the prescribed protocol Π, which entails interaction between the servers, and a real world adversary $\mathcal{A}$ who controls at

most one server. We let $\text{IDEAL}_{f,\mathcal{S}}(1^\kappa, z)$ denote the output of the honest parties and the view of the ideal-world adversary $\mathcal{S}$ from the ideal execution with respect to the security parameter 1^κ and auxiliary input z. Similarly, let $\text{REAL}_{\Pi,\mathcal{A}}(1^\kappa, z)$ denote the output of the honest parties and the view of the adversary $\mathcal{A}$ from the real execution of the protocol Π with respect to the security parameter and auxiliary input z. We say that the protocol Π securely computes f for every PPT real world adversary $\mathcal{A}$, there exists a PPT ideal world adversary $\mathcal{S}$, corrupting the same parties, such that the following two distributions are computationally indistinguishable $\text{IDEAL}_{f,\mathcal{S}}(1^\kappa, z) \equiv_c \text{REAL}_{\Pi,\mathcal{A}}(1^\kappa, z)$. The ideal functionality $\mathcal{F}_{\mathsf{3PC}}$ for evaluating ckt in the 3PC setting appears in Fig. 2.1.

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

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

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

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

Figure 2.1: 3PC: Ideal functionality for evaluating a function f

2.2 Shared Key Setup

Let $f : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function PRF, with co-domain X being $\mathbb{Z}_{2^\ell}$. The set of keys established between the servers for 3PC is as follows:

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

Suppose P_0, P_1 wish to sample a random value $r \in \mathbb{Z}_{2^\ell}$ non-interactively. To do so they invoke $F_{k_{01}}(id_{01})$ and obtain r. Here, id_{01} denotes a counter maintained by the servers, and is updated after every PRF invocation. The appropriate keys used to sample is implicit from the context, from the identities of the pair that sample or from the fact that it is sampled by all, and, hence, is omitted. The functionality $\mathcal{F}_{\mathsf{setup}}$ (put reference) appears below (Fig.2.2) and can be instantiated using any standard MPC protocol in the respective setting.

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

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

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

Figure 2.2: 3PC: Ideal functionality for shared-key setup

2.3 Collision Resistant Hash

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

2.4 Commitment Scheme

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

Chapter 3

Robust 3PC Protocols

In this section, we present a robust and efficient 3PC protocol with security against one malicious adversary. We first introduce the sharing semantics for three servers. This is followed by the details of our new Joint Message Passing (jmp) primitive, which plays a crucial role in obtaining the strongest security guarantee of GOD. Finally we describe our protocols in the three server setting.

3.1 Sharing Semantics

We use the following variants of secret-sharing schemes in this work. The sharing schemes are either over $\mathbb{Z}_{2^\ell}$ (*Arithmetic* sharing), or over $\mathbb{Z}_{2^1}$ (*Boolean* sharing). The latter is denoted as $[\![\cdot]\!]^{\mathbf{B}}$.

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

3.1.1 Linearity of the sharing schemes

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

and $[\![\cdot]\!]$-sharing as well.

3.2 Joint Message Passing primitive

The primitive jmp allows two servers to relay a common value to the third server such that either the value is relayed correctly or an honest server is identified. Interestingly, for a message of ℓ elements, it only incurs a communication of ℓ elements (in amortized sense). jmp is crucial for achieving GOD and lies at the heart of all of our 3PC constructions. The ideal functionality for jmp appears in Fig. 3.1.

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

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

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

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

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

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

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

Figure 3.1: 3PC: Ideal functionality for jmp primitive

Given two servers P_i, P_j possessing a common value $v \in \mathbb{Z}_{2^\ell}$, protocol Π_{jmp} proceeds as follows: First, P_i sends v to P_k while P_j sends a hash of the same $(\mathsf{H}(\mathsf{v}))$ to P_k. If P_k receives consistent values, the protocol is done. For the case of an inconsistency, Π_{jmp} allows the servers to identify a set of disputing servers and an honest server who will be employed as the TTP. At a high level, this is accomplished by first making all the servers aware of the inconsistency observed by P_k , after which they broadcast $\mathsf{H}(\mathsf{v})$, wherein P_k broadcasts the hash of v received from P_i . Two servers are said to be in dispute if the values broadcast by them are not consistent.

First we discuss how P_k informs the other servers that it receive inconsistent values.Towards this, each P_s for $s \in \{i, j, k\}$ maintains an inconsistency bit b_s initialized to 0. P_k sets bit b_k to 1 and send this to both P_i , P_j . In the next round, P_i, P_j mutually exchanges the bit b_k received from P_k.P_i (resp.P_j) sets b_i (resp.b_j) to 1 if it sees at least one variant of b_k bit with value 1 in any of the two rounds. In the third round, all the servers with inconsistency bit set to 1 broadcast hash of the value $(\mathsf{H}(\mathsf{v}))$. Here, P_k will be broadcasting the hash of the value received from P_i . If P_k does not broadcast hash, this implies that everything is correct at P_k 's end and the protocol is complete. Hence, servers proceed to identify a TTP, only if at least two servers broadcast, one of which is P_k . We observe that, in case of a genuine inconsistency

there will be at least two broadcasts; a corrupt sender might simply choose not to broadcast. Now, we discuss in detail how the servers identify a TTP. Let H_s be the hash broadcast by $P_s \in \{P_i, P_j, P_k\}$.If any of the senders P_i, P_j do not broadcast the hash, while P_k does, we can safely assume P_k to be the TTP. Consider the case where all three servers broadcast. If $\mathsf{H}_i \neq \mathsf{H}_j$, this implies that one of P_i or P_j is corrupt, and hence P_k is identified as the TTP. Similarly, if $\mathsf{H}_i \neq \mathsf{H}_k$, P_j is chosen as the TTP, and if $H_j \neq \mathsf{H}_k$, P_i is chosen as the TTP. Finally, if all the hash values broadcast are consistent, we let P_i to be the TTP. To see this, we show that this case can never occur when P_i is corrupt. A corrupt P_i implies that all other servers are honest. Thus, an inconsistency is a result of P_i sending a different v' to P_k , which causes P_k to set its inconsistency bit b_k to be 1. This is followed by the two round of exchange, as discussed above, at the end of which P_j also sets its bit to 1. Now, the servers broadcast the hash of v. Notice that P_k broadcasts $\mathsf{H}(\mathsf{v}')$ while P_j broadcasts $\mathsf{H}(\mathsf{v})$. Hence, no matter what P_i broadcasts, all the hash values will not be consistent. Thus, such a case never arises when P_i is corrupt.
We ensure that, in each step, a party raises a public alarm (via broadcast) accusing a party who is silent when it is not supposed to be, and the protocol terminates immediately by labelling the party as TTP who is neither the complainer nor the accused. We say that P_i, P_j $\mathsf{jmp\text{-}send}$ v to P_k when they invoke $\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{v})$.
The formal protocol is as follows (Fig.3.2).

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

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

Figure 3.2: 3PC: Joint Message Passing Protocol

Looking ahead, all our protocols will use jmp and consequently our final construction, either of general MPC or of any PPML task, will have several calls to jmp. For the amortization to work, the broadcast of values is executed once and for all for a fixed ordered pair of senders in the end.

Lemma 3.1 (Communication)**.** *Protocol* Π_{jmp} *(Fig. 3.2) requires* 1 *round and an amortized communication of* 1ℓ *bits overall.*

Proof. Server P_i sends the value v to P_k while P_j sends hash of the same. This accounts for 1 round and communication of 1ℓ bits. Then P_k sends back its inconsistency bit to P_i, P_j, who then exchange it; this takes two rounds in all. The hash of the value v and distribution of inconsistency bit can be combined for several instances of this protocol and hence gets amortized over multiple instances. This is followed by a servers broadcasting hashes on their values and selecting a TTP based on it, which takes 1 round. This too can be clubbed for multiple instances and hence amortizes the cost and rounds. □

3.3 Input Sharing Protocol

Protocol Π_{sh} (Fig. 3.3) allows a server P_i to generate $[\![\cdot]\!]$-shares of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. In the preprocessing phase, P_0, P_j for $j \in \{1,2\}$ along with P_i sample a random $[\![\alpha_\mathsf{v}]\!]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2, P_i sample random $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$. This allows P_i to know both α_v and γ_v in clear. During the online phase, if $P_i = P_0$, then P_0 sends $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ to P_1. P_0, P_1 then jmp-send β_v to P_2 to complete the secret sharing. If $P_i = P_1$, P_1 sends $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ to P_2. Then P_1, P_2 jmp-send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0. The case for $P_i = P_2$ proceeds similar to that of P_1. The correctness of the shares held by each server is assured by the guarantees of Π_{jmp}.

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

Preprocessing:

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

Online:

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

Figure 3.3: 3PC: Generating $[\![\mathsf{v}]\!]$-shares by server P_i

Lemma 3.2 (Communication)**.** *Protocol* Π_{sh} *(Fig. 3.3) is non-interactive in the preprocessing phase and requires* 2 *round and an amortized communication of* 2ℓ *bits in the online phase.*

Proof. During the preprocessing phase, servers non-interactively sample the shares of α_v and γ_v values using the shared key setup. In the online phase, when $P_i = P_0$, he/she computes β_v and sends it to P_1, resulting in 1 round and 1ℓ bits communicated. They then execute Π_{jmp} to provide P_2 with β_v, which requires additional 1 round in an amortized sense, and 1ℓ bits to be communicated. For the case when $P_i = P_1$, she sends β_v to P_2, resulting in 1 round and a communication of 1ℓ ring elements. This is followed by one invocation of Π_{jmp} to relay $\beta_\mathsf{v} + \gamma_\mathsf{v}$ towards P_0. This again requires an additional 1 round and 1ℓ bits. The analysis is similar in the case of $P_i = P_2$. □

3.4 Joint Sharing Protocol

Protocol Π_{jsh} (Fig. 3.4) allows two servers P_i, P_j to jointly generate a $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is known to both. Towards this, servers execute the preprocessing of Π_{sh} (Fig. 3.3) to generate $[\alpha_\mathsf{v}]$ and γ_v. If $(P_i, P_j) = (P_1, P_0)$, then P_1, P_0 jmp-send $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ to P_2. The case when $(P_i, P_j) = (P_2, P_0)$ proceeds similarly. The case for $(P_i, P_j) = (P_1, P_2)$ is optimized further as follows: servers locally set $[\alpha_\mathsf{v}]_1 = [\alpha_\mathsf{v}]_2 = 0$. P_1, P_2 together sample random $\gamma_\mathsf{v} \in \mathbb{Z}_{2^\ell}$, set $\beta_\mathsf{v} = \mathsf{v}$ and jmp-send $\beta_\mathsf{v} + \gamma_\mathsf{v}$ to P_0.

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

Preprocessing:

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

Online:

- If $(P_i, P_j) = (P_1, P_0)$: P_0, P_1 computes $\beta_\mathsf{v} = \mathsf{v} + [\alpha_\mathsf{v}]_1 + [\alpha_\mathsf{v}]_2$. Servers execute $\Pi_{\mathsf{jmp}}(P_0, P_1, P_2, \beta_\mathsf{v})$ to enable P_2 obtain β_v.
- If $(P_i, P_j) = (P_2, P_0)$: Similar to the case above.
- If $(P_i, P_j) = (P_1, P_2)$: P_1, P_2 locally set $\beta_\mathsf{v} = \mathsf{v}$. Servers then execute $\Pi_{\mathsf{jmp}}(P_1, P_2, P_0, \mathsf{v} + \gamma_\mathsf{v})$ to enable P_0 obtain $\beta_\mathsf{v} + \gamma_\mathsf{v}$.

Figure 3.4: $[\![\cdot]\!]$-sharing of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ by servers P_i, P_j

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

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

Table 3.1: The columns depict the three distinct possibility of input contributing pairs. The first row shows the assignment to various components of the sharing. The last row, along with three sub-rows, specify the shares held by the three servers.

Lemma 3.3 (Communication)**.** *Protocol* Π_{jsh} *(Fig. 3.4) is non-interactive in the preprocessing phase and requires* 1 *round and an amortized communication of at most* 1ℓ *bits in the online phase.*

Proof. In this protocol, servers execute Π_{jmp} protocol once. Hence the overall cost follows from that of an instance of the Π_{jmp} protocol (Lemma 3.1). □

3.5 Addition Protocol

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

3.6 Multiplication Protocol

Given the $[\![\cdot]\!]$-shares on input wires x, y, protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$ (Fig. 3.5) allows servers to securely evaluate $[\![\mathsf{z}]\!]$, where $\mathsf{z} = \mathsf{xy}$. Our protocol provides stronger security notion of GOD at the cost of 4ℓ and 3ℓ elements communicated (amortized) in the preprocessing and the online phase, respectively. We first explain the protocol in the semi-honest setting. During the preprocessing phase, P_0, P_j for $j \in \{1, 2\}$ sample random $[\alpha_{\mathsf{z}}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample random $\gamma_{\mathsf{z}} \in \mathbb{Z}_{2^\ell}$. In addition, P_0 locally computes $\Gamma_{\mathsf{xy}} = \alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$ and generates $[\cdot]$-sharing of the same between P_1, P_2. Since

$$\begin{aligned}\beta_{\mathsf{z}} &= \mathsf{z} + \alpha_{\mathsf{z}} = \mathsf{xy} + \alpha_{\mathsf{z}} = (\beta_{\mathsf{x}} - \alpha_{\mathsf{x}})(\beta_{\mathsf{y}} - \alpha_{\mathsf{y}}) + \alpha_{\mathsf{z}} \\ &= \beta_{\mathsf{x}}\beta_{\mathsf{y}} - \beta_x\alpha_{\mathsf{y}} - \beta_y\alpha_{\mathsf{x}} + \Gamma_{\mathsf{xy}} + \alpha_{\mathsf{z}} \qquad (3.1)\end{aligned}$$

holds, servers P_1, P_2 locally compute $[\beta_{\mathsf{z}}]_j = (j-1)\beta_{\mathsf{x}}\beta_{\mathsf{y}} - \beta_x [\alpha_{\mathsf{y}}]_j - \beta_y [\alpha_{\mathsf{x}}]_j + [\Gamma_{\mathsf{xy}}]_j + [\alpha_{\mathsf{z}}]_j$ during the online phase and mutually exchange their shares to reconstruct β_{z}. P_1 then sends $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0, completing the semi-honest protocol. The correctness that asserts $\mathsf{z} = \mathsf{xy}$ or in other words $\beta_{\mathsf{z}} - \alpha_{\mathsf{z}} = \mathsf{xy}$ holds due to Eq. 3.1.

In case of a malicious adversary, the following issues arise. We explain how to handle each issue in a robust way.

1) When P_0 is corrupt, the $[\cdot]$-sharing of Γ_{xy} performed by P_0 might not be correct, i.e. $\Gamma_{\mathsf{xy}} \neq \alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$.

2) When P_1 (or P_2) is corrupt, the $[\cdot]$-share of β_{z} handed over to the fellow honest evaluator during the online phase might not be correct, causing reconstruction of an incorrect β_{z}.

3) When P_1 is corrupt, the value $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ that is sent to P_0 during the online phase may not be correct.

The last issue is trivially handled by making P_1, P_2 jmp-send $\beta_z + \gamma_z$ to P_0 (after β_z is computed). This either leads to success or a TTP selection. We explain the rest of the cases assuming that P_0, P_i posses the correct and consistent values for $[\chi]_i, [\Gamma_{xy}]_i$ and servers P_1, P_2 have the correct ψ. P_0 first computes $\beta_z^\star$ and sends it to P_1, P_2, who compute β_z as: $\beta_z = \beta_z^\star + \beta_x\beta_y - \psi$. However, this does not solve the problem; a corrupt P_0 can send an incorrect value for $\beta_z^\star$ causing the honest evaluators to compute a wrong value for β_z. To fix this, we make use of the observation that servers P_0, P_i, for $i \in \{1, 2\}$, given correct and consistent $[\cdot]$-shares of χ and Γ_{xy}, can compute a $[\cdot]$-share of $\beta_z^\star$ locally as: $[\beta_z^\star]_i = -(\beta_x + \gamma_x)[\alpha_y]_i - (\beta_y + \gamma_y)[\alpha_x]_i + 2[\Gamma_{xy}]_i + [\alpha_z]_i + [\chi]_i$. For P_1 to obtain $\beta_z^\star$ in clear it needs $[\beta_z^\star]_2$, which is available with both P_0, P_2. Hence, the servers jmp-send $[\beta_z^\star]_2$ to P_1. This guarantees that P_1 obtains the correct $\beta_z^\star$ from which it can compute β_z, or, in case of a dispute, all servers identify a TTP. Similarly, servers jmp-send $[\beta_z^\star]_1$ to allow P_2 to compute correct β_z. In case a TTP is identified, all servers send their inputs in clear to the TTP, who evaluates the circuit and sends back the output. With these modifications, the online phase of the multiplication protocol is now robust.

For the online phase to proceed successfully, the servers P_0, P_i need correct and consistent values for $[\chi]_i, [\Gamma_{xy}]_i$ and servers P_1, P_2 need the correct ψ. We now explain how to generate these values correctly in the preprocessing phase. To extract the required values, the following relation is exploited. d, e, f set as $d = (\gamma_x - \alpha_x)$, $e = (\gamma_y - \alpha_y)$ and $f = (\gamma_x\gamma_y + \psi) - \chi$, form a multiplication triple. Servers compute a $\langle\cdot\rangle$-sharing of f from $\langle\cdot\rangle$-sharing of d, e, which are set according to the table 3.2.

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

Table 3.2: The $\langle\cdot\rangle$-sharing of values d and e

This is followed by an execution of a maliciously secure of [7], henceforth referred to as Π_{mulZK}, as a result of which they receive $\langle f \rangle = ([\lambda_f], f + \lambda_f)$. χ, Γ_{xy} and ψ are then extracted as follows:

$$[\chi]_1 = [\lambda_f]_1 \text{ and } [\chi]_2 = [\lambda_f]_2 \quad \rightarrow \chi = [\lambda_f]_1 + [\lambda_f]_2$$

$$\gamma_x\gamma_y + \psi = f + \lambda_f \quad \rightarrow \psi = f + \lambda_f - \gamma_x\gamma_y$$

$$[\Gamma_{xy}]_j = \gamma_x[\alpha_y]_j + \gamma_y[\alpha_x]_j + [\psi]_j - [\chi]_j \quad [j \in \{1, 2\}]$$

Here, $[\cdot]$-shares of ψ are generated by P_1, P_2 non-interactively; P_1, P_2 together sample a random $r \in \mathbb{Z}_{2^\ell}$, P_1 sets $[\psi]_1 = r$, while P_2 sets $[\psi]_2 = \psi - r$. Γ_{xy} set in this way provides P_1, P_2 with $[\cdot]$-shares of Γ_{xy}.

Finally, to complete the picture two subtle issues need to tackled. Note that, in the online phase, P_0, P_1, P_2 should have consistent $[\cdot]$-shares of Γ_{xy}, which cannot be done non-interactively. We observe that P_0 does not have the same shares as those possessed by P_1, P_2 owing to the re-randomization of these shares that took place while extracting χ and ψ. To enable P_0 to obtain consistent shares as those held by P_1, P_2, the following observation is used.

The protocol Π_{mulZK} first runs a semihonest protocol and then verifies correctness in zero-knowledge.The underlying semihonest protocol requires P_0 to compute and distribute $[\cdot]$-shares of $\lambda_{\mathsf{de}} = \lambda_{\mathsf{d}}\lambda_{\mathsf{e}} = \alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$. P_0 obtains consistent $[\cdot]$-share of Γ_{xy} from the $[\cdot]$-shares of λ_{de} computed in the underlying semi-honest protocol in Π_{mulZK}. $[\Gamma_{\mathsf{xy}}]$ as set by P_1, P_2 can be viewed as re-randomized shares of $\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$. P_0, P_i for $i \in \{1, 2\}$ retain $[\lambda_{\mathsf{de}}]_i$ from the underlying semi-honest protocol in Π_{mulZK}. Using these shares, P_1 computes $\Delta = [\lambda_{\mathsf{f}}]_1 - [\Gamma_{\mathsf{xy}}]_1$ and P_2 compute the $\Delta = [\Gamma_{\mathsf{xy}}]_2 - [\lambda_{\mathsf{f}}]_2$. Servers jmp-send Δ to P_0. Using $[\cdot]$-shares of λ_{xy} and Δ, P_0 can locally re-compute consistent $[\cdot]$-shares of Γ_{xy} as: $[\Gamma_{\mathsf{xy}}]_1 = [\lambda_{\mathsf{f}}]_1 - \Delta$ and $[\Gamma_{\mathsf{xy}}]_2 = [\lambda_{\mathsf{f}}]_2 + \Delta$. Secondly, the protocol Π_{mulZK} as in [7] gives a security with abort. However, we need a robust version in order to get an overall robust protocol. We use techniques from [8] to get a modified version Π_{mulZK} that provides security with guaranteed output delivery. Precisely, in the verification phase, instead of simply exchanging shares of the proof, a verifier and the prover jmp-send the missing share of the proof to the second verifier.

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

Preprocessing:

- Servers P_0, P_j for $j \in \{1,2\}$ together sample a random $[\alpha_{\mathsf{z}}]_j \in \mathbb{Z}_{2^\ell}$, while P_1, P_2 sample a random $\gamma_{\mathsf{z}} \in \mathbb{Z}_{2^\ell}$.
- Servers in $\mathcal{P}$ locally compute $\langle\cdot\rangle$-sharing of $\mathsf{d} = \gamma_{\mathsf{x}} - \alpha_{\mathsf{x}}$ and $\mathsf{e} = \gamma_{\mathsf{y}} - \alpha_{\mathsf{y}}$ by setting the shares as (as per Tale 3.2):

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

- Servers in $\mathcal{P}$ execute $\Pi_{\mathsf{mulZK}}(\mathcal{P}, \mathsf{d}, \mathsf{e})$ to generate $\langle\mathsf{f}\rangle = \langle\mathsf{de}\rangle$.
- P_0, P_j for $j \in \{1,2\}$ locally set $[\chi]_j = [\lambda_{\mathsf{f}}]_j$, while P_1, P_2 set $\psi = \mathsf{f} + \lambda_{\mathsf{f}} - \gamma_{\mathsf{x}}\gamma_{\mathsf{y}}$. P_0 then computes $\chi = [\chi]_1 + [\chi]_2$.
- P_1, P_2 sample random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and set $[\psi]_1 = \mathsf{r}, [\psi]_2 = \psi - \mathsf{r}$.
- P_j for $j \in \{1,2\}$ set $[\Gamma_{\mathsf{xy}}]_j = \gamma_{\mathsf{x}}\,[\alpha_{\mathsf{y}}]_j + \gamma_{\mathsf{y}}\,[\alpha_{\mathsf{x}}]_j + [\psi]_j - [\chi]_j$
- P_1 sets $\Delta = [\lambda_{xy}]_1 - [\Gamma_{\mathsf{xy}}]_1$, P_2 sets $\Delta = [\Gamma_{\mathsf{xy}}]_2 - [\lambda_{xy}]_2$. Servers execute $\Pi_{\mathsf{jmp}}(P_1, P_2, P_0, \Delta)$.
- If a TTP is identified in the previous step, then all servers send their inputs in clear to the TTP who computes the circuit and distributes the output. Else, P_0 computes $[\Gamma_{\mathsf{xy}}]_1 = [\lambda_{xy}]_1 - \Delta$ and $[\Gamma_{\mathsf{xy}}]_2 = [\lambda_{xy}]_2 + \Delta$.

Online:

- P_0, P_j, for $j \in \{1,2\}$, compute $[\beta^\star_{\mathsf{z}}]_j = -(\beta_{\mathsf{x}} + \gamma_{\mathsf{x}})\,[\alpha_{\mathsf{y}}]_j - (\beta_{\mathsf{y}} + \gamma_{\mathsf{y}})\,[\alpha_{\mathsf{x}}]_j + [\alpha_{\mathsf{z}}]_j + 2\,[\Gamma_{\mathsf{xy}}]_j + [\chi]_j$.
- Servers in $\mathcal{P}$ execute $\Pi_{\mathsf{jmp}}(P_0, P_1, P_2, [\beta^\star_{\mathsf{z}}]_1)$ and $\Pi_{\mathsf{jmp}}(P_0, P_2, P_1, [\beta^\star_{\mathsf{z}}]_2)$ in parallel.
- If a TTP is not identified then :
- P_j for $j \in \{1,2\}$ computes $[\beta^\star_{\mathsf{z}}] = [\beta^\star_{\mathsf{z}}]_1 + [\beta^\star_{\mathsf{z}}]_2$, $\beta_{\mathsf{z}} = \beta^\star_{\mathsf{z}} + \beta_{\mathsf{x}}\beta_{\mathsf{y}} - \psi$.
- Servers execute $\Pi_{\mathsf{jmp}}(P_1, P_2, P_0, \beta_{\mathsf{z}} + \gamma_{\mathsf{z}})$.
- If a TTP is identified in any of the previous step, then all servers send their inputs in clear to the TTP who computes the circuit and distributes the output.

Figure 3.5: Multiplication Protocol

Lemma 3.4 (Communication)**.** *Protocol* Π_{mult} *(Fig. 3.5) requires an amortized cost of* 4ℓ *bits in the preprocessing phase, and* 1 *round and amortized cost of* 3ℓ *bits in the online phase.*

Proof. In the preprocessing phase, the servers first generate α_{v} and γ_{v} non-interactively using shared key setup. This is followed by one execution of Π_{mulPre}, and one jmp-send for Δ, which requires an amortized communication cost of 4ℓ bits. During the online phase, servers P_0, P_1

locally compute share $\beta^\star$ which is relayed to P_2 using Π_{jmp}. Simultaneously, P_0, P_2 relay the second share of $\beta^\star$ to P_1. This requires 1 round and amortized cost of 2ℓ bits. This is followed by computation of $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ by P_1, P_2 who send it to P_0 with yet another invocation of Π_{jmp}, which again requires 1 round and communication cost of 1ℓ bits. However, sending of $\beta_{\mathsf{v}} + \gamma_{\mathsf{v}}$ can be delayed till the end of the protocol, and will require only one round for the entire circuit and can be amortized. Thus, the total number of rounds required for multiplication is 1. □

3.7 Output Reconstruction

Protocol $\Pi_{\mathsf{rec}}(\mathcal{P}, [\![\mathsf{v}]\!])$ (Fig. 3.6) allows servers to robustly reconstruct a secret $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ from its $[\![\cdot]\!]$-shares. Notice that each server has all but one share required to reconstruct v in clear. This missing share is held by the other two servers. For reconstruction of v, P_0, P_1, P_2 need γ_{v}, $[\![\alpha_{\mathsf{v}}]\!]_2$ and $[\![\alpha_{\mathsf{v}}]\!]_1$, respectively. To reconstruct robustly, servers first agree on a commitment on these shares. Concretely, in the preprocessing phase, P_0, P_1 together, using their shared randomness, prepare a commitment on $[\![\alpha_{\mathsf{v}}]\!]_1$. They then execute Π_{jmp} to provide P_2 with the correct commitment. Similarly, using Π_{jmp}, P_0, P_2 together commit $[\![\alpha_{\mathsf{v}}]\!]_2$ towards P_1 and P_1, P_2 together commit γ_{v} towards P_0. At this point, note that the agreement on commitments takes place in the preprocessing phase, which makes the online phase very efficient. Now, in the online phase, each server receives the missing share along with the opening to the commitment from the other two servers. The commitments allow the recipient to identify the correct missing share. Moreover, it is guaranteed to get at least one correct missing share, commitment pair, which helps to reconstruct the output robustly.

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

Preprocessing:

- P_0, P_j, for $j \in \{1,2\}$, generate commitment of $[\![\alpha_\mathsf{v}]\!]_j$ ($\mathsf{Com}([\![\alpha_\mathsf{v}]\!]_j)$) and send it to P_{3-j}. P_1, P_2 generate commitment of γ_v ($\mathsf{Com}(\gamma_\mathsf{v})$) and send it to P_0. Servers use randomness sampled from the PRF key-setup for the commitments.
- Servers in $\mathcal{P}$ execute $\Pi_{\mathsf{jmp}}(P_0, P_1, P_2, \mathsf{Com}([\![\alpha_\mathsf{v}]\!]_1))$, $\Pi_{\mathsf{jmp}}(P_0, P_2, P_1, \mathsf{Com}([\![\alpha_\mathsf{v}]\!]_2))$ and $\Pi_{\mathsf{jmp}}(P_1, P_2, P_0, \mathsf{Com}(\gamma_\mathsf{v})$ in parallel.

Online:

- If a TTP has been identified then parties send inputs to the TTP in clear. The TTP will evaluate the circuit and distribute the output to all other servers.
- If no TTP has been identified,
- P_0, P_1 open $\mathsf{Com}([\![\alpha_\mathsf{v}]\!]_1)$ towards P_2.
- P_0, P_2 open $\mathsf{Com}([\![\alpha_\mathsf{v}]\!]_2)$ towards P_1.
- P_1, P_2 open $\mathsf{Com}(\gamma_\mathsf{v})$ towards P_0.
- Servers reconstruct the output using the share that is consistent with the commitment.

Figure 3.6: Reconstructing the secret v from $[\![\cdot]\!]$-shares

Lemma 3.5 (Communication)**.** *Protocol* Π_{rec} *(Fig. 3.6) requires* 1 *round and amortized communication of* 6ℓ *bits in the online phase.*

Proof. In the preprocessing phase, servers generate commitments on shares of α_v and γ_v and then invoke three instances of Π_{jmp} protocol parallelly in order to provide all with consistent commitments. The commitment can be instantiated using a hash function and can be clubbed for multiple instances amortizing its cost. During the online phase, each server gets an opening from other two servers, which requires 1 round and overall 6ℓ bits to be communicated. □

3.8 Input Sharing and Output Reconstruction in an SOC setting

Protocol $\Pi_{\mathsf{sh}}^{\mathsf{SOC}}$ (Fig. 3.7) extends input sharing to the SOC setting and allows a user U to generate the $[\![\cdot]\!]$-shares of its input v among the three servers. Note that the necessary commitments to facilitate the sharing are generated in the preprocessing phase by the servers which are then communicated to U, along with the opening, in the online phase. U selects the commitment forming the majority (for each share) owing to the presence of an honest majority among the servers, and accepts the corresponding shares. Analogously, protocol $\Pi_{\mathsf{rec}}^{\mathsf{SOC}}$ (Fig. 3.7) allows the servers to reconstruct a value v towards user U. In either of the protocols, if at any point,

a TTP is identified, then servers signal the TTP's identity to U. U selects the TTP as the one forming a majority and sends its input in clear to the TTP, who computes the function output and sends it back to U.

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

Input Sharing:

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

Output Reconstruction:

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

Figure 3.7: 3PC: Input Sharing and Output Reconstruction in SOC Setting

3.9 The Complete 3PC

In order compute an arithmetic circuit over $\mathbb{Z}_{2^\ell}$, the servers first execute the setup phase where they obtain the respective shared keys via $\mathcal{F}_{\mathsf{setup}}$. then, in the input sharing phase, $P_i \in \mathcal{P}$ shares its input x_i by executing Π_{sh} (Fig. 3.3) protocol. This is followed by the circuit evaluation phase, where severs evaluate the gates in the circuit in the topological order, with addition gates (and multiplication-by-a-constant gates) being computed locally, whereas multiplication gates being computed by invoking Π_{mult} (Fig. 3.5) protocol. Finally, servers run Π_{rec} protocol (Fig. 3.6) on the output wires to reconstruct the function output.

Chapter 4

Building Blocks for PPML

In this chapter we provide details on robust realizations of the following building blocks for PPML in 3-server setting– i) Dot Product, ii) Truncation, iii) Dot Product with Truncation, iv) Secure Comparison, and v) Non-linear Activation functions– Sigmoid and ReLU.

4.1 MSB Extraction

The Bit Extraction Protocol, Π_{bitext} allows servers to compute the boolean sharing of the most significant bit (msb) of a value v given its arithmetic sharing $[\![\mathsf{v}]\!]$. To compute the msb, we use the optimized 2-input Parallel Prefix Adder (PPA) boolean circuit proposed by ABY3 [31]. The PPA circuit consists of $2\ell - 2$ AND gates and has a multiplicative depth of $\log \ell$. Let

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

Table 4.1: The $[\![\cdot]\!]^{\mathbf{B}}$-sharing corresponding to i^{th} bit of $\mathsf{v}_0 = \beta_{\mathsf{v}}, \mathsf{v}_1 = -[\alpha_{\mathsf{v}}]_1$ and $\mathsf{v}_2 = -[\alpha_{\mathsf{v}}]_2$. Here $i \in \{0,\ldots,\ell-1\}$.

$\mathsf{v}_0 = \beta_{\mathsf{v}}, \mathsf{v}_1 = -[\alpha_{\mathsf{v}}]_1$ and $\mathsf{v}_2 = -[\alpha_{\mathsf{v}}]_2$. Then $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$. Servers first locally compute the boolean shares corresponding to each bit of the values $\mathsf{v}_0, \mathsf{v}_1$ and v_2 according to Table 4.1. It has been shown in ABY3 that $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$ can also be expressed as $\mathsf{v} = 2c + s$ where $\mathsf{FA}(\mathsf{v}_0[i],\mathsf{v}_1[i],\mathsf{v}_2[i]) \rightarrow (c[i],s[i])$ for $i \in \{0,\ldots,\ell-1\}$. Here FA denotes a Full Adder circuit while s and c denote the sum and carry bits respectively. To summarize, servers execute ℓ instances of FA in parallel to compute $[\![c]\!]^{\mathbf{B}}$ and $[\![s]\!]^{\mathbf{B}}$. The FA's are executed independently and require one round of communication. The final result is then computed as $\mathsf{msb}(2[\![c]\!]^{\mathbf{B}} + [\![s]\!]^{\mathbf{B}})$ using the optimized PPA circuit.

Lemma 4.1 (Communication). *Protocol* Π_{bitext} *requires a communication cost of* 12ℓ *bits in the preprocessing phase and require* $\log \ell + 1$ *rounds and an amortized communication of* 9ℓ *bits in the online phase.*

4.1.1 Bit2A Conversion protocol

Given the boolean sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, protocol Π_{bit2A} (Fig. 4.1) allows servers to compute the arithmetic sharing $[\![\mathsf{b}^{\mathsf{R}}]\!]$. Here b^{R} denotes the equivalent value of b over ring $\mathbb{Z}_{2^\ell}$. It can be seen that $\mathsf{b}^{\mathsf{R}} = (\beta_{\mathsf{b}} \oplus \alpha_{\mathsf{b}})^{\mathsf{R}} = \beta_{\mathsf{b}}^{\mathsf{R}} + \alpha_{\mathsf{b}}^{\mathsf{R}} - 2\beta_{\mathsf{b}}^{\mathsf{R}}\alpha_{\mathsf{b}}^{\mathsf{R}}$. Also $\alpha_{\mathsf{b}}^{\mathsf{R}} = ([\alpha_{\mathsf{b}}]_1 \oplus [\alpha_{\mathsf{b}}]_2)^{\mathsf{R}} = [\alpha_{\mathsf{b}}]_1^{\mathsf{R}} + [\alpha_{\mathsf{b}}]_2^{\mathsf{R}} - 2\,[\alpha_{\mathsf{b}}]_1^{\mathsf{R}}\,[\alpha_{\mathsf{b}}]_2^{\mathsf{R}}$. During the preprocessing phase, P_0, P_j for $j \in \{1,2\}$ execute Π_{jsh} on $[\alpha_b]_j^{\mathsf{R}}$ to generate $[\![[\alpha_b]_j^{\mathsf{R}}]\!]$. Servers then execute Π_{mult} on $[\![[\alpha_b]_1^{\mathsf{R}}]\!]$ and $[\![[\alpha_b]_2^{\mathsf{R}}]\!]$ to generate $[\![[\alpha_b]_1^{\mathsf{R}}\,[\alpha_b]_2^{\mathsf{R}}]\!]$ followed by locally computing $[\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$. During the online phase, P_1, P_2 execute Π_{jsh} on $\beta_{\mathsf{b}}^{\mathsf{R}}$ to jointly generate $[\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!]$. Servers then execute Π_{mult} protocol on $[\![\beta_{\mathsf{b}}^{\mathsf{R}}]\!]$ and $[\![\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$ to compute $[\![\beta_{\mathsf{b}}^{\mathsf{R}}\alpha_{\mathsf{b}}^{\mathsf{R}}]\!]$ followed by locally computing b^{R}. The formal details for Π_{bit2A} protocol appears in Fig. 4.1.

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

Preprocessing:

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

Online:

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

Figure 4.1: 3PC: Bit2A Protocol

Lemma 4.2 (Communication). *Protocol* Π_{bit2A} *(Fig. 4.1) requires an amortized communication cost of* 11ℓ *bits in the preprocessing phase and requires* 1 *round and an amortized communication of* 4ℓ *bits in the online phase.*

4.2 Bit Injection

Given the binary sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, and the arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, protocol Π_{BitInj} computes $[\![\cdot]\!]$-sharing of bv. Towards this, servers first execute Π_{bit2A} on $[\![\mathsf{b}]\!]^{\mathbf{B}}$

to generate $[\![\mathsf{b}]\!]$. This is followed by servers computing $[\![\mathsf{bv}]\!]$ by executing Π_{mult} protocol on $[\![\mathsf{b}]\!]$ and $[\![\mathsf{v}]\!]$.

Lemma 4.3 (Communication)**.** *Protocol* Π_{BitInj} *requires an amortized communication cost of* 15ℓ *bits in the preprocessing phase and requires* 2 *rounds and an amortized communication of* 7ℓ *bits in the online phase.*

Proof. Protocol Π_{BitInj} is essentially an execution of Π_{bit2A} (Lemma 4.1) followed by one invocation of Π_{mult} (Lemma 3.4) and the costs follow. □

4.3 Dot Product

Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol Π_{dotp} (Fig. Fig. 4.2) allows servers in $\mathcal{P}$ to generate $[\![\cdot]\!]$-sharing of $\vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ robustly. By $[\![\cdot]\!]$-sharing of a vector $\vec{\mathbf{x}}$ of size n, we mean that each element $\mathsf{x}_i \in \mathbb{Z}_{2^\ell}$ in the vector, for $i \in [\mathsf{n}]$, is $[\![\cdot]\!]$-shared. Servers P_0, P_1 locally compute a $[\cdot]$ share of $\beta^\star_{\mathsf{z}}$ as: $[\beta^\star_{\mathsf{z}}]_1 = -\sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})\,[\alpha_{\mathsf{y}_i}]_1 - \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})\,[\alpha_{\mathsf{x}_i}]_1 + [\alpha_{\mathsf{z}}]_1 + 2\,[\Gamma_{\mathsf{xy}}]_1 + [\chi]_1$ and $\mathsf{jmp\text{-}send}$ to allow P_2 to correctly reconstruct $\beta^\star_{\mathsf{z}}$. Similarly, P_1 reconstructs $\beta^\star_{\mathsf{z}}$. P_1, P_2 locally compute $\beta_{\mathsf{z}} = \beta^\star_{\mathsf{z}} + \sum_{i=1}^{\mathsf{n}} \beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i} - \psi$. Servers then execute Π_{jmp} to provide P_0 with correct $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$. The formal details are as follows.

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

Preprocessing:

- For each pair $(\mathsf{x}_i, \mathsf{y}_i)$, such that $i \in [\mathsf{n}]$ and $\mathsf{z}_i = \mathsf{x}_i\mathsf{y}_i$, servers in $\mathcal{P}$ execute preprocessing phase of $\Pi_{\mathsf{mult}}(\mathcal{P}, \mathsf{x}_i, \mathsf{y}_i)$, so that P_0 obtains χ_i, and P_s obtains $[\Gamma_{\mathsf{x}_i\mathsf{y}_i}]_s$ and ψ_i for $s \in \{1, 2\}$. P_0, P_s also store $[\lambda_{x_iy_i}]_s$ from the underlying protocol.
- P_0 computes $\chi = \sum_{i=1}^{\mathsf{n}} \chi_i$. P_0, P_s for $s \in \{1, 2\}$ compute $[\lambda_{\mathsf{xy}}]_s = \sum_{i=1}^{\mathsf{n}} [\lambda_{\mathsf{x}_i\mathsf{y}_i}]_s$. P_s for $s \in \{1, 2\}$ computes $\psi = \sum_i \psi_i$ and $[\Gamma_{\mathsf{xy}}]_s = \sum_{i=1}^{\mathsf{n}} [\Gamma_{\mathsf{x}_i\mathsf{y}_i}]_s$.
- P_1 sets $\Delta = [\lambda_{xy}]_1 - [\Gamma_{\mathsf{xy}}]_1$, P_2 sets $\Delta = [\Gamma_{\mathsf{xy}}]_2 - [\lambda_{xy}]_2$. P_1, P_2 $\mathsf{jmp\text{-}send}$ Δ to P_0 who locally recomputes $[\Gamma_{\mathsf{xy}}]_1$ and $[\Gamma_{\mathsf{xy}}]_2$.
- P_0, P_s for $s \in \{1, 2\}$ compute $[\alpha_{\mathsf{z}}]_s = \sum_{i=1}^{\mathsf{n}} [\alpha_{\mathsf{z}_i}]_s$. P_1, P_2 compute $\gamma_{\mathsf{z}} = \sum_{i=1}^{\mathsf{n}} \gamma_{\mathsf{z}_i}$. P_0 computes

Online Phase:

- P_0, P_s for $s \in \{1, 2\}$ compute $[\beta^\star_{\mathsf{z}}]_s = -\sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{x}_i} + \gamma_{\mathsf{x}_i})\,[\alpha_{\mathsf{y}_i}]_s - \sum_{i=1}^{\mathsf{n}}(\beta_{\mathsf{y}_i} + \gamma_{\mathsf{y}_i})\,[\alpha_{\mathsf{x}_i}]_s + [\alpha_{\mathsf{z}}]_s + 2\,[\Gamma_{\mathsf{xy}}]_s + [\chi]_s$.
- P_0, P_s for $s \in \{1, 2\}$ $\mathsf{jmp\text{-}send}$ $[\beta^\star_{\mathsf{z}}]_s$ to P_{3-s}.
- P_1, P_2 locally compute $\beta^\star_{\mathsf{z}} = [\beta^\star_{\mathsf{z}}]_1 + [\beta^\star_{\mathsf{z}}]_2$ and $\beta_{\mathsf{z}} = \beta^\star_{\mathsf{z}} + \sum_{i=1}^{\mathsf{n}} \beta_{\mathsf{x}_i}\beta_{\mathsf{y}_i} - \psi$.
- P_1, P_2 $\mathsf{jmp\text{-}send}$ $\beta_{\mathsf{z}} + \gamma_{\mathsf{z}}$ to P_0.

Figure 4.2: Dot Product Protocol

Lemma 4.4 (Communication)**.** *Protocol* Π_{dotp} *requires an amortized communication cost of* $3\mathsf{n} + 1\ell$ *bits in the preprocessing phase and requires* 1 *rounds and an amortized communication of* 3ℓ *bits in the online phase.*

4.4 Truncation

For truncation, servers execute Π_{trgen} (Fig. 4.3) to generate a random pair of the form $([\mathsf{r}], [\![\mathsf{r}^d]\!])$. Here, r denotes a random ring element, while r^d represents the truncated value of r. By truncated value, we mean that the value is right-shifted by d bit positions, where d is the number of bits allocated for the fractional part in the FPA representation. Given $(\mathsf{r}, \mathsf{r}^d)$, the truncated value of v denoted by v^d can be computed from v as $\mathsf{v}^d = (\mathsf{v} - \mathsf{r})^d + \mathsf{r}^d$.

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

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

Figure 4.3: 3PC: Generating Random Truncated Pair $(\mathsf{r}, \mathsf{r}^d)$

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

Lemma 4.5 (Communication)**.** *Protocol* Π_{dotpt} *requires an amortized communication cost of* 2ℓ *bits in the preprocessing phase.*

4.5 Dot Product with Truncation

Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$, protocol Π_{dotpt} (Fig. 4.4) allows servers to generate $[\![\mathsf{z}^d]\!]$, where z^d denotes the truncated value of $\mathsf{z} = \vec{\mathsf{x}} \odot \vec{\mathsf{y}}$. The preprocessing phase now consists of the execution of one instance of Π_{trgen} (Fig. 4.3) and the preprocessing corresponding to Π_{dotp} (Fig. 4.2). At a high level, the online phase proceeds as follows: P_0, P_j for $j \in \{1, 2\}$ locally compute $[\mathsf{z}^\star - \mathsf{r}]_j$ (instead of $[\beta^\star_\mathsf{z}]_j$ as in Π_{dotp}) where $\mathsf{z}^\star = \beta^\star_\mathsf{z} - \alpha_\mathsf{z}$. P_0, P_1 $\mathsf{jmp\text{-}send}$ $[\mathsf{z}^\star - \mathsf{r}]_1$ to P_2 while P_0, P_2 $\mathsf{jmp\text{-}send}$ $[\mathsf{z}^\star - \mathsf{r}]_2$ to P_1. Both P_1, P_2 then compute $(\mathsf{z} - \mathsf{r})$ locally, truncate it to obtain $(\mathsf{z} - \mathsf{r})^d$ and execute Π_{jsh} to generate $[\![(\mathsf{z} - \mathsf{r})^d]\!]$. Finally, servers locally compute the result as $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z} - \mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$. We defer the formal details of the protocol Π_{dotpt} to §??.

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

Preprocessing:

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

Online:

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

Figure 4.4: 3PC: Dot Product Protocol with Truncation

Lemma 4.6 (Communication)**.** *Protocol* Π_{dotp} *requires an amortized communication cost of* $3\mathsf{n} + 3\ell$ *bits in the preprocessing phase and requires* 1 *rounds and an amortized communication of* 3ℓ *bits in the online phase.*

4.6 Secure Comparison

Secure comparison allows servers to compare two values $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ given their $[\![\cdot]\!]$-shares, i.e check whether $\mathsf{x} < \mathsf{y}$ or not. In fixed-point arithmetic representation, checking whether $\mathsf{v} = \mathsf{x} - \mathsf{y}$ gives the desired result. This is done by checking the msb of v in the following way. Servers first locally compute $[\![\mathsf{v}]\!] = [\![\mathsf{x}]\!] - [\![\mathsf{y}]\!]$, then extract the msb using protocol Π_{bitext} on $[\![\mathsf{v}]\!]$. In

case an arithmetic sharing is desired, servers can further apply the Bit2A protocol Π_{bit2A} on the outcome of Π_{bitext}.

4.7 Activation Function

We consider two of the most prominently used activation functions: i) Rectified Linear Unit (ReLU) and (ii) Sigmoid (Sig).

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

– *Sig:* In this work, we use the MPC-friendly variant of the Sigmoid function [32, 31, 12] as given below:

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

Note that $\mathsf{sig}(\mathsf{v}) = \overline{\mathsf{b}_1}\mathsf{b}_2(\mathsf{v} + 1/2) + \overline{\mathsf{b}_2}$, where $\mathsf{b}_1 = 1$ if $\mathsf{v} + 1/2 < 0$ and $\mathsf{b}_2 = 1$ if $\mathsf{v} - 1/2 < 0$. To compute $[\![\mathsf{sig}(\mathsf{v})]\!]$, servers proceed in a similar fashion as the ReLU, and hence, we skip the formal details.

Chapter 5

Security of the 3PC Constructions

This chapter involves detailed security proofs for all of our 3PC constructions. We prove security using the real-world/ ideal-word simulation based technique. We provide proofs in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model for the case of 3PC, where $\mathcal{F}_{\mathsf{setup}}$ (Fig. 2.2) denotes the ideal functionality for the three server shared-key setup.

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

In some of our sub protocols, adversary is able to decide on which among the honest parties should be chosen as the Trusted Third Party (TTP) in that execution of the protocol. To capture this, we consider *corruption-aware* functionalities [4] for the sub-protocols, where the functionality is provided the identity of the corrupt server as an auxiliary information.

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

5.0.1 Joint Message Passing (jmp) Protocol

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

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

- $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ initializes $\mathsf{ttp} = \perp$ and receives v_i from $\mathcal{A}$ on behalf of P_k.
- In case, $\mathcal{A}$ fails to send a value $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ broadcasts `"(accuse,`P_i`)"`, sets $\mathsf{ttp} = P_j$, $\mathsf{v}_i = \perp$, and skip to the last step.
- Else, it checks if $\mathsf{v}_i = \mathsf{v}$, where v is the value computed by $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ based on the interaction with $\mathcal{A}$, and using the knowledge of the shared keys. If the values are equal, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $b_k = 0$, else, sets $b_k = 1$, and sends the same to $\mathcal{A}$ on the behalf of P_k.
- If $\mathcal{A}$ broadcasts `"(accuse,`P_k`)"`, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{v}_i = \perp$, $\mathsf{ttp} = P_j$, and skips to the last step.
- $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ computes and sends b_j to $\mathcal{A}$ on behalf of P_j and receives $b_{\mathcal{A}}$ from $\mathcal{A}$ on behalf of honest P_j.
- If $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ does not receive a $b_{\mathcal{A}}$ on behalf of P_j, it broadcasts `"(accuse,`P_i`)"`, sets $\mathsf{v}_i = \perp$, $\mathsf{ttp} = P_k$. If $\mathcal{A}$ broadcasts `"(accuse,`P_j`)"`, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{v}_i = \perp$, $\mathsf{ttp} = P_k$. If ttp is set, skip to the last step.
- If $(\mathsf{v}_i = \mathsf{v})$ and $b_{\mathcal{A}} = 1$, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ broadcasts $\mathsf{H}_j = \mathsf{H}(\mathsf{v})$ on behalf of P_j.
- Else if $\mathsf{v}_i \neq \mathsf{v}_j$: $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ broadcasts $\mathsf{H}_j = \mathsf{H}(\mathsf{v})$ and $\mathsf{H}_k = \mathsf{H}(\mathsf{v}_i)$ on behalf of P_j and P_k, respectively. If $\mathcal{A}$ does not broadcast, $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{ttp} = P_k$. Else if, $\mathcal{A}$ broadcasts a value $\mathsf{H}_{\mathcal{A}}$:
  - If $\mathsf{H}_{\mathcal{A}} \neq \mathsf{H}_j$: $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{ttp} = P_k$.
  - Else if $\mathsf{H}_{\mathcal{A}} \neq \mathsf{H}_k$: $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ sets $\mathsf{ttp} = P_j$.
- $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ invokes $\mathcal{F}_{\mathsf{jmp}}$ on $(\mathsf{Input}, \mathsf{v}_i)$ and $(\mathsf{Select}, \mathsf{ttp})$ on behalf of $\mathcal{A}$.

Figure 5.1: Simulator $\mathcal{S}_{\mathsf{jmp4}}^{P_i}$ for corrupt sender P_i

The case for a corrupt receiver, P_k is provided in Fig. 5.2.

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

– $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ initializes $\mathsf{ttp} = \perp$, computes v honestly and sends v and $\mathsf{H}(\mathsf{v})$ to $\mathcal{A}$ on behalf of P_i and P_j, respectively.

– If $\mathcal{A}$ broadcasts `"(accuse,`P_i`)"`, set $\mathsf{ttp} = P_j$, else if $\mathcal{A}$ broadcasts `"(accuse,`P_j`)"`, set $\mathsf{ttp} = P_i$. If both messages are broadcast, set $\mathsf{ttp} = P_i$. If ttp is set skip to the last step.

– On behalf of P_i, P_j, $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ receives $b_{\mathcal{A}}$ from $\mathcal{A}$. Let b_i (resp. b_j) denote the bit received by P_i (resp. P_j) from $\mathcal{A}$.

– If $\mathcal{A}$ failed to send bit $b_{\mathcal{A}}$ to P_i, $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ broadcasts `"(accuse,`P_k`)"`, set $\mathsf{ttp} = P_j$. Similarly, for P_j. If both P_i, P_j broadcast `"(accuse,`P_k`)"`, set $\mathsf{ttp} = P_i$. If ttp is set, skip to the last step.

– If $b_i \vee b_j = 1$: $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ broadcasts $\mathsf{H}_i, \mathsf{H}_j$ where $\mathsf{H}_i = \mathsf{H}_j = \mathsf{H}(\mathsf{v})$ on behalf of P_i, P_j, respectively.

– If $\mathcal{A}$ does not broadcast $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ sets $\mathsf{ttp} = \perp$. If $\mathcal{A}$ broadcasts a value $\mathsf{H}_{\mathcal{A}}$:

- If $\mathsf{H}_{\mathcal{A}} \neq \mathsf{H}_i$: $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ sets $\mathsf{ttp} = P_j$.
- Else if $\mathsf{H}_{\mathcal{A}} = \mathsf{H}_i = \mathsf{H}_j$: $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ sets $\mathsf{ttp} = P_i$.

– $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ invokes $\mathcal{F}_{\mathsf{jmp}}$ on $(\mathsf{Input}, \perp)$ and $(\mathsf{Select}, \mathsf{ttp})$ on behalf of $\mathcal{A}$.

Figure 5.2: Simulator $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ for corrupt receiver P_k

5.0.2 Sharing Protocol

Here we give he simulation steps for Π_{sh}. The case for a corrupt P_0 is provided in Fig. 5.5.

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

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

Online:

– If the dealer $P_s = P_0$:

- $\mathcal{S}_{\mathsf{sh}}^{P_0}$ receives β_{v} on behalf of P_1 and sets $\mathsf{msg} = \mathsf{v}$ accordingly.
- Steps for Π_{jmp} protocol are simulated according to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ (Fig. 5.1), where P_0 plays the role of one of the senders.

– If the dealer $P_s = P_1$:

- $\mathcal{S}_{\mathsf{sh}}^{P_0}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$.
- Steps for Π_{jmp} protocol are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_k}$ (Fig. 5.2), with P_0 acting as the receiver.

– If the dealer if P_2 : Similar to the case when $P_s = P_1$.

Figure 5.3: Simulator $\mathcal{S}_{\mathsf{sh}}^{P_0}$ for corrupt P_0

The case for a corrupt P_1 is provided in Fig. 5.4. The case for a corrupt P_2 is similar.

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

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

Online:

- If dealer $P_s = P_1$: $\mathcal{S}_{\mathsf{sh}}^{P_1}$ receives β_{v} from $\mathcal{A}$ on behalf of P_2.
- If $P_s = P_0$: $\mathcal{S}_{\mathsf{sh}}^{P_1}$ sets $\mathsf{v} = 0$ by assigning $\beta_{\mathsf{v}} = \alpha_{\mathsf{v}}$ and sends β_{v} to $\mathcal{A}$ on behalf of P_s.
- If $P_s = P_2$: Similar to the case where $P_s = P_0$.
- Steps of Π_{jmp}, in all the steps above, are simulated similar to $\mathcal{S}_{\mathsf{jmp}}^{P_i}$ (Fig. 5.1), ie. the case of corrupt sender.

Figure 5.4: Simulator $\mathcal{S}_{\mathsf{sh}}^{P_1}$ for corrupt P_1

5.0.2.1 Joint Sharing Protocol

Here we give he simulation steps for Π_{jsh}. The case for a corrupt P_0 is provided in Fig. 5.5. The case for a corrupt P_1, P_2 is similar.

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

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

Online:

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

Figure 5.5: Simulator $\mathcal{S}_{\mathsf{jsh}}$ for corrupt P_0

5.0.3 Multiplication Protocol

Here we give he simulation steps for Π_{mult}. The case for a corrupt P_0 is provided in Fig. 5.6.

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

Preprocessing:

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

Online:

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

Figure 5.6: Simulator $\mathcal{S}_{\mathsf{mult}}^{P_0}$ for corrupt P_0

The case for a corrupt P_1 is provided in Fig. 5.7. The case for a corrupt P_2 is similar.

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

Preprocessing:

- $\mathcal{S}_{\mathsf{mult}}^{P_1}$ samples $[\alpha_{\mathsf{z}}]_1, \gamma_{\mathsf{z}}$ and $[\alpha_{\mathsf{z}}]_2$ on behalf of P_0, P_2. $\mathcal{S}_{\mathsf{mult}}^{P_1}$ generates the $\langle\cdot\rangle$-shares of d, e honestly.
- $\mathcal{S}_{\mathsf{mult}}^{P_1}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$, and extracts $\psi, [\chi]_1, [\chi]_2$ on behalf of P_0, P_2. It also extracts $[\Gamma_{\mathsf{xy}}]_2$ on behalf of P_2
- $\mathcal{S}_{\mathsf{mult}}^{P_1}$ participates in $\mathsf{jmp\text{-}send}$ to send Δ to P_0.

Online:

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

Figure 5.7: Simulator $\mathcal{S}_{\mathsf{mult}}^{P_1}$ for corrupt P_1

5.0.4 Reconstruction Protocol

Here we give he simulation steps for Π_{rec}. The case for a corrupt P_0 is provided in Fig. 5.8. The case for a corrupt P_1, P_2 is similar.

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

Preprocessing:

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

Online:

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

Figure 5.8: Simulator $\mathcal{S}_{\mathsf{rec}}$ for corrupt P_0

5.0.5 Truncation

Here we give he simulation steps for Π_{trgen}. The case for a corrupt P_0 is provided in Fig. 5.9.

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

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

Figure 5.9: Simulator $\mathcal{S}_{\mathsf{trgen}}^{P_0}$ for corrupt P_0

The case for a corrupt P_1 is provided in Fig. 5.10. The case for a corrupt P_2 is similar.

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

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

Figure 5.10: Simulator $\mathcal{S}_{\mathsf{trgen}}^{P_1}$ for corrupt P_1

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

Chapter 6

PPML Applications

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

Benchmarking Environment We use a 64-bit ring ($\mathbb{Z}_{2^{64}}$). The benchmarking is performed over a WAN that was instantiated using n1-standard-8 instances of Google Cloud[1], with machines located in East Australia (P_0), South Asia (P_1), South East Asia (P_2), and West Europe (P_3). The machines are equipped with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors supporting hyper-threading, with 8 vCPUs, and 30 GB of RAM Memory and with a bandwidth of 50 Mbps. The average round-trip time (rtt) was taken as the time for communicating 1 KB of data between a pair of parties, and the rtt values were

P_0-P_1	P_0-P_2	P_1-P_2
$151.40ms$	$59.95ms$	$92.94ms$

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

Biometric Matching Biometric computation is central to many real-world tasks such as face recognition [22, 25] and fingerprint-matching [5, 24]. The objective is, given a database $\mathbf{D}$ of m biometric samples stored as vectors $(\vec{\mathbf{s}}_1, \ldots, \vec{\mathbf{s}}_m)$ each of size n, and a user with its own sample $\vec{\mathbf{u}}$, identify the "closest" sample to $\vec{\mathbf{u}}$ in $\mathbf{D}$. This task can be accomplished by considering

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

various distance metrics, the most prominent of which is the Euclidean Distance (ED). In this work, we consider ED as the metric, and hence the problem boils down to identifying a sample vector in $\mathbf{D}$ which has the least ED for $\vec{\mathbf{u}}$. Note that, in our setting, each entry in the database $\mathbf{D}$ is $[\![\cdot]\!]$-shared among the servers. The client with an input query $\vec{\mathbf{u}}$ generates $[\![\cdot]\!]$- shares of the same along with the servers. Let x_i denote the ith element in the vector $\vec{\mathbf{x}}$. As was introduced in [32], ED between two n length vectors $\vec{\mathbf{x}}$, $\vec{\mathbf{y}}$ is computed as $\mathbf{ED}_{\vec{\mathbf{x}}\vec{\mathbf{y}}} = \sum_{i=1}^{i=\mathsf{n}}(\mathsf{x}_i - \mathsf{y}_i)^2 = \vec{\mathbf{z}} \odot \vec{\mathbf{z}}$ where $\vec{\mathbf{z}} = ((\mathsf{x}_1 - \mathsf{y}_1), \ldots, (\mathsf{x}_\mathsf{n} - \mathsf{y}_\mathsf{n}))$. Hence, the servers first compute $[\![\cdot]\!]$-shares for vector $\vec{\mathbf{z}}$ locally, as $[\![\mathsf{z}_i]\!] = [\![\mathsf{x}_i]\!] - [\![\mathsf{y}_i]\!]$ for $i \in [\mathsf{n}]$, followed by an execution of Π_{dotp} on $[\![\vec{\mathbf{z}}]\!], [\![\vec{\mathbf{z}}]\!]$. For biometric computation, the servers create a distance vector $\mathbf{DV}$ by computing the ED between $\vec{\mathbf{u}}$ and every sample vector $\vec{\mathbf{s}}_i$ in $\mathbf{D}$, i.e $\mathbf{DV}_i = \mathbf{ED}_{\vec{\mathbf{u}}\vec{\mathbf{s}}_i}$ for $i \in [m]$. The next task now is to find the minimum among the m values in $\mathbf{DV}$.

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

Setting	Ref.	$m = 1024$			$m = 16384$		
		Pre.	Online		Pre.	Online	
		Com [KB]	R	Com [KB]	C [KB]	R	Com [KB]
3PC	BLAZE	1127.1	102	151.1	18036.0	142	2419.9
	This	1131.3	103	151.9	18041.8	143	2420.7

Table 6.1: Minimum ED distance. The values are reported for biometric samples of size 40.

Table 6.1 presents the benchmarking for biometric matching. Following SecureML [32], we chose the size of the biometric sample n to be 40. As is evident from the Table 6.1, we incur a minimal loss in performance over BLAZE but guarantee the security of GOD instead of fairness.

Privacy-preserving Machine Learning We consider training and inference for Linear Regression and Logistic Regression and inference for Neural Networks (NN). Note that, NN train-

ing requires additional tools to allow mixed world computations, which we leave as future work. We refer readers to SecureML [32], ABY3 [31], and BLAZE [36] for a detailed description of the training and inference steps for the aforementioned ML algorithms. All our benchmarking is done over the publicly available MNIST [28] dataset that has $\mathsf{n} = 784$ features. For training, we used a batch size of $B = 128$. In 3PC, we compare our results against the best-known framework BLAZE in this setting that provides fairness. Our results imply that we get GOD at no additional cost compared to BLAZE.

Benchmarking Parameter We use *throughput* (TP) as the benchmarking parameter following BLAZE and ABY3 [31] as it would help to analyse the effect of improved communication and round complexity in a single shot. Here, TP denotes the number of operations ("iterations" for the case of training and "queries" for the case of inference) that can be performed in unit time. We consider minute as the unit time since most of our protocols over WAN requires more than a second to complete. An *iteration* in ML training consists of a *forward propagation* phase followed by a *backward propagation* phase. In the former phase, servers compute the output from the inputs while in the latter, the model parameters are adjusted according to the difference in the computed output and the actual output. The inference can be viewed as one forward propagation of the algorithm alone.

Logistic Regression In Logistic Regression, one iteration comprises updating the weight vector $\vec{\mathbf{w}}$ using the gradient descent algorithm (GD). It is updated according to the function given below: $\vec{\mathbf{w}} = \vec{\mathbf{w}} - \frac{\alpha}{B}\mathbf{X}_i^T \circ (\mathsf{sig}(\mathbf{X}_i \circ \vec{\mathbf{w}}) - \mathbf{Y}_i)$. where α and $\mathbf{X}_i$ denote the learning rate, and a subset of batch size B, randomly selected from the entire dataset in the ith iteration, respectively. The forward propagation comprises of computing the value $\mathbf{X}_i \circ \vec{\mathbf{w}}$ followed by an application of a sigmoid function on it. The weight vector is updated in the backward propagation, which internally requires the computation of a series of matrix multiplications, and can be achieved using a dot product. The update function can be computed using $[\![\cdot]\!]$ shares as: $[\![\vec{\mathbf{w}}]\!] = [\![\vec{\mathbf{w}}]\!] - \frac{\alpha}{B}[\![\mathbf{X}_j^T]\!] \circ (\mathsf{sig}([\![\mathbf{X}_j]\!] \circ [\![\vec{\mathbf{w}}]\!]) - [\![\mathbf{Y}_j]\!])$. We summarize our results in Table 6.2.

Setting	Ref.	Pre.	Online (TP in $\times 10^3$)		
		Com [KB]	Latency (s)	Com [KB]	TP
3PC Training	BLAZE	4757.11	1.17	50.23	2525.36
	This	4760.29	1.23	50.31	2393.38
3PC Inference	BLAZE	18.69	1.08	0.25	2728.65
	This	19.71	1.08	0.28	2727.38

Table 6.2: Logistic Regression training and inference. TP is given in (#it/min) for training and (#queries/min) for inference.

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

NN Inference In this work, we consider a NN with two hidden layers, each consisting of 128 nodes each and an output layer with 10 nodes [31, 36]. Each of the layers is fully connected. Inference in NN requires several dot product calls followed by an application of the ReLU function. This process will be carried out for each layer in a sequential manner. 6.3 summarises our benchmarking results for NN inference and confirms that our performance is comparable to BLAZE.

Setting	Ref.	Pre.	Online (TP in $\times 10^3$)		
		Com [MB]	Latency (s)	Com [MB]	TP
3PC	BLAZE	351.70	2.81	4.91	26.40
Inference	**This**	353.52	2.91	4.91	26.40

Table 6.3: NN Inference. TP is given in (#queries/min).. Benchmarking is done over MNIST [28] dataset and the throughput (TP) is given in (#queries/min).

Bibliography

[1] Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. Cryptology ePrint Archive, Report 2019/1298, 2019. https://eprint.iacr.org/2019/1298. 2

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS*, pages 805–817, 2016. 2

[3] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *IEEE S&P*, pages 843–862, 2017. 2, 3

[4] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly secure multiparty computation. *J. Cryptology*, pages 58–151, 2017. 29

[5] Marina Blanton and Paolo Gasti. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS*, pages 190–209, 2011. 35

[6] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008. 2, 3

[7] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, pages 67–97, 2019. 2, 4, 5, 17, 18

[8] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM CCS*, pages 869–886, 2019. 2, 4, 5, 18

[9] Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. In *ACM CCS*, pages 486–497, 2007. 2

[10] Megha Byali, Arun Joseph, Arpita Patra, and Divya Ravi. Fast secure computation for small population over the internet. In *ACM CCS*, pages 677–694, 2018. 2

[11] Megha Byali, Carmit Hazay, Arpita Patra, and Swati Singla. Fast actively secure five-party computation with security beyond abort. In *ACM CCS*, pages 1573–1590, 2019. 2, 7

[12] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*, 2019. URL https://eprint.iacr.org/2019/429. 2, 3, 7, 28

[13] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. *NDSS*, 2020. URL https://arxiv.org/abs/1912.02631. 4, 7

[14] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*, pages 34–64, 2018. 2

[15] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *ACM STOC*, pages 364–369, 1986. 3

[16] Ran Cohen, Iftach Haitner, Eran Omri, and Lior Rotem. Characterization of secure multiparty computation without broadcast. *J. Cryptology*, pages 587–609, 2018. 7

[17] Cryptography and Privacy Engineering Group at TU Darmstadt. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils, 2017. 35

[18] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO*, pages 799–829, 2018. 3

[19] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015. 4

[20] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative scientific computations. In *IEEE CSFW-14*, pages 273–294, 2001. 2

[21] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! arithmetic 3pc for any modulus with active security. Cryptology ePrint Archive, Report 2019/164, 2019. https://eprint.iacr.org/2019/164. 2, 3

[22] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *PETS*, pages 235–253, 2009. 35

[23] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*, pages 225–255, 2017. 2

[24] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *ASIA CCS*, pages 437–446, 2013. 35

[25] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *ACM CCS*, pages 451–462, 2010. 35

[26] Yuval Ishai, Ranjit Kumaresan, Eyal Kushilevitz, and Anat Paskin-Cherniavsky. Secure computation with minimal interaction, revisited. In *CRYPTO*, pages 359–378, 2015. 2

[27] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *ACM SIGKDD*, pages 593–599, 2005. 2

[28] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/. vi, 37, 38

[29] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS*, pages 259–276, 2017. 2

[30] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. *J. Cryptology*, pages 177–206, 2002. 2

[31] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS*, pages 35–52, 2018. 2, 7, 23, 28, 37, 38

[32] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, pages 19–38, 2017. 4, 7, 28, 36, 37

[33] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *ACM CCS*, pages 591–602, 2015. 2

[34] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In *ACNS*, pages 321–339, 2018. 2

[35] Arpita Patra and Divya Ravi. On the exact round complexity of secure three-party computation. In *CRYPTO*, pages 425–458, 2018. 2

[36] Arpita Patra and Ajith Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. *NDSS*, 2020. URL https://eprint.iacr.org/2020/042. 2, 4, 7, 37, 38

[37] Ashish P. Sanil, Alan F. Karr, Xiaodong Lin, and Jerome P. Reiter. Privacy preserving regression modelling via distributed computation. In *ACM SIGKDD*, pages 677–682, 2004. 2

[38] Aleksandra B. Slavkovic, Yuval Nardi, and Matthew M. Tibbits. Secure logistic regression of horizontally and vertically partitioned distributed databases. In *ICDM*, pages 723–728, 2007. 2

[39] Jaideep Vaidya, Hwanjo Yu, and Xiaoqian Jiang. Privacy-preserving SVM classification. *Knowl. Inf. Syst.*, pages 161–178, 2008. 2

[40] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *PoPETs*, pages 26–49, 2019. 2

[41] Hwanjo Yu, Jaideep Vaidya, and Xiaoqian Jiang. Privacy-preserving SVM classification on vertically partitioned data. In *PAKDD*, pages 647–656, 2006. 2