MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Ajith S

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

July, 2021

Declaration of Originality

I, **Ajith S**, with SR No. **04-04-00-10-12-17-1-14980** hereby declare that the material presented in the thesis titled

MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2017-2021**.
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: 28^{th} July, 2021 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: Arpita Patra Advisor Signature

DEDICATION

I dedicate my sincere efforts to my sweet and loving

Achan, Amma & Devu

whose love, affection and words of encouragement guided me in achieving success and honor,

Along with all my beloved

Teachers

for being a great source of inspiration

Acknowledgements

I begin by thanking the Almighty for blessing me with the strength to fulfil my commitment to research during my PhD.

It gives me immense pleasure to express my deep sense of gratitude and respect to my research supervisor, Professor Arpita Patra, to accept me into her family. I joined this family in 2014 as an M.Tech (Research) student, and she has been a constant source of inspiration ever since. I would love to address her as my elder sister in place of my advisor. It has been a great experience to work under the supervision of Prof. Arpita, who treats her students as her family, bridging the gap between faculty and student. Her support and guidance helped me all along during my research and in completing my thesis.

I am greatly indebted to Ashish Choudhury, Professor at IIIT Bangalore, for being a significant contributor in kick-starting my research. It is one of the questions posed by him during my initial journey, which led to the line of research that constitutes the contributions of this thesis. His genuine kindness, warmth and strong work ethic have left a lasting impression in my life, which I will cherish forever.

The CrIS lab has been like my second home, and I am grateful to all my lab mates with whom I have had a great time. I would like to extend warm gratitude to Divya Ravi and Megha Byali for being my strong support system on both academic and personal fronts. Also, a special mention to Nishat Koti, with whom I have shared a lot of time working on research problems.

During my studies, I got an opportunity to visit the ENCRYPTO group at the Technical University of Darmstadt. I owe my deepest gratitude to Prof. Thomas Schneider for hosting me and giving me a chance to interact with his group. These internships have played a key role in boosting my confidence, and my interactions with them have helped me develop new insights and expand my research horizon. I would like to use this opportunity to thank Rahul Rachuri and Hossein Yalame for co-authoring some of my works.

I am thankful to Ms Padmavathi and Ms Kushael at the CSA office for their constant help and support on the administrative front. I am also thankful to Prof. Deepak D'Souza and Prof. Bhavana Kanukurthi of CSA, IISc, for their support during my initial PhD journey.

Acknowledgements

I am grateful for the Google PhD fellowship for supporting my research. My sincere acknowledgements to IACR, NDSS, Alan Turing Institute, and the institute GARP sponsorship for supporting my travel to international conferences.

My heartfelt thanks to all the teachers who have taught me since first grade. I'm also grateful to my undergraduate, graduate and post-graduate family. Finally, I would love to express profound gratitude to my parents and my wife for encouraging me with unfailing support and advice throughout my research. Their unwavering love and affection mean the world to me, and I dedicate this thesis to them.

Abstract

In the modern era of computing, machine learning tools have demonstrated their potential in vital sectors, such as healthcare and finance, to derive proper inferences. The sensitive and confidential nature of the data in such sectors raises genuine concerns for data privacy. This motivated the area of Privacy-preserving Machine Learning (PPML), where privacy of data is guaranteed. Typically, machine learning techniques require significant computing power, which leads clients with limited infrastructure to rely on the method of Secure Outsourced Computation (SOC). In the SOC setting, the computation is outsourced to a set of specialized and powerful cloud servers and the service is availed on a pay-per-use basis. In this thesis, we design an efficient platform, MPCLeague, for PPML in the SOC setting using Secure Multi-party Computation (MPC) techniques.

MPC, the holy-grail problem of secure distributed computing, enables a set of n mutually distrusting parties to perform joint computation on their private inputs in a way that no coalition of t parties can learn more information than the output (privacy) or affect the true output of the computation (correctness). While MPC, in general, has been a subject of extensive research, the area of MPC with a small number of parties has drawn popularity of late mainly due to its application to real-time scenarios, efficiency and simplicity. This thesis focuses on designing efficient MPC frameworks for 2, 3 and 4 parties, with at most one corruption and supports ring structures.

Our platform aims at achieving the most substantial security notion of robustness, where the honest parties are guaranteed to obtain the output irrespective of the behaviour of the corrupt parties. A robust protocol prevents the corrupt parties from repeatedly causing the computations to rerun, thereby upholding the trust in the system. While on the roadmap to attain robustness, our frameworks also demonstrate constructions with improved performance that achieve relaxed notions of security: security with abort and fairness. A fair protocol enforces the restriction that either all parties or none of them receive the output. On the other hand, honest parties may not receive the output while corrupt parties do for the case of security with abort.

Abstract

The general structure of the computation involves the execution of the protocol steps once the participating parties have supplied their inputs. Finally, the output is distributed to all the parties. However, to enhance practical efficiency, many recent works resort to the preprocessing paradigm, which splits the computation into two phases; a preprocessing phase where input-independent (but function-dependent), computationally heavy tasks can be computed, followed by a fast online phase. Since the same functions in ML are evaluated several times, this paradigm naturally fits the case of PPML, where the ML algorithm is known beforehand.

At the heart of this thesis are four frameworks – ASTRA, SWIFT, Tetrad, ABY2.0 - catered to different settings.

– ASTRA: We begin with the setting of 3 parties (3PC), which forms the base case for honest majority. If a majority of the participating parties are honest, then the setting is deemed an honest majority setting. In the set of 3 parties, at most one party can be corrupt, and this framework tackles semi-honest corruption, where the corrupt party follows the protocol steps but tries to glean more information from the computation. ASTRA acts as a stepping stone towards achieving a stronger security guarantee against active corruption. Our protocol requires communication of 2 ring elements per multiplication gate during the online phase, attaining a per-party cost of less than one element. This is achieved for the first time in the regime of 3PC.

– SWIFT: Designed for 3 parties, this framework tackles one active corruption where the corrupt party can arbitrarily deviate from the computation. Building on ASTRA, SWIFT provides a multiplication that improves the communication to 6 ring elements from 21 over the state-of-the-art, besides improving security from abort to robustness. In the regime of malicious 3PC, SWIFT is the first robust and efficient PPML framework. It achieves a dot product protocol with communication independent of the vector size for the first time.

– Tetrad: Designed for 4 parties in the honest majority, the fair multiplication protocol in Tetrad requires communication of only 5 ring elements instead of 6 in the state-of-the-art. The fair framework is then extended to provide robustness without inflating the costs. A notable contribution is the design of the multiplication protocol that supports on-demand applications where the function to be computed is not known in advance.

– ABY2.0: Moving on to the stronger corruption model where a majority of the parties can be corrupt, we explore the base case of 2 parties (2PC). Since we aim to achieve robustness which is proven to be impossible in active corruption, we restrict ourselves to semi-honest

corruption. The prime contribution of this framework is the scalar product for which the online communication is two ring elements irrespective of the vector dimension. This is a feature achieved for the first time in the 2PC literature.

Our frameworks provide the following contributions in addition to the ones mentioned above. First, we support multi-input multiplication for arithmetic and boolean worlds, improving the online phase in rounds and communication. Second, all our frameworks except SWIFT, incorporate truncation without incurring any overhead. Finally, we introduce efficient instantiation of garbled-world, tailor-made for the mixed-protocol framework for the first time. The mixed-protocol approach, combining arithmetic, boolean and garbled style computations, has demonstrated its potential in several practical use-cases like PPML. To facilitate the computation, we also provide the conversion mechanisms to switch between the computation styles.

The practicality of our framework is argued through improvements in the benchmarking of widely used ML algorithms – Linear Regression, Logistic Regression, Neural Networks, and Support Vector Machines. We propose two variants for each of our frameworks, with one variant aiming to minimise the execution time while the other focuses on the monetary cost.

The concrete efficiency gains of our frameworks coupled with the stronger security guarantee of robustness make our platform an ideal choice for a real-time deployment of privacy-preserving machine learning techniques.

Publications based on this Thesis

The work in this dissertation is primarily related to the following articles. Publications in cryptography usually order authors alphabetically (using surnames) and conferences are more common than journals[1].

Accepted Papers

1. Nishat Koti, Arpita Patra, Rahul Rachuri and **Ajith Suresh**. *Tetrad: Actively Secure 4PC for Secure Training and Inference* [87]. NDSS'22 (CORE A*), PPML'21 (CCS)

2. Nishat Koti, Mahak Pancholi, Arpita Patra and **Ajith Suresh**. *SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning* [85]. USENIX Security'21 (CORE A*), PRIML/PPML'20 (NeurIPS), DPML'21 (ICLR)

3. Arpita Patra, Thomas Schneider, **Ajith Suresh** and Hossein Yalame. *ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation* [113]. USENIX Security '21 (CORE A*), PPML'21 (CCS), PriML'21 (NeurIPS), PPML'21 (CRYPTO)

4. Nishat Koti, Arpita Patra and **Ajith Suresh**. *MPCLeague: Robust and Efficient Mixed-protocol Framework for 4-party Computation* [86]. IEEE S&P 2021 (Poster), DPML'21 (ICLR)

5. Harsh Chaudhari, Rahul Rachuri and **Ajith Suresh**. *Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning* [38]. NDSS'20 (CORE A*)

6. Arpita Patra and **Ajith Suresh**. *BLAZE: Blazing Fast Privacy-Preserving Machine Learning* [110]. NDSS'20 (CORE A*)

7. Harsh Chaudhari, Ashish Choudhury, Arpita Patra and **Ajith Suresh**. *ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction* [37]. ACM CCSW'19, PPML'19 (CCS)

[1] Workshops without proceedings are marked in this colour.

Publications outside this Thesis

Accepted Papers

1. Arpita Patra, Thomas Scheider, Ajith Suresh and Hossein Yalame. *SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation* [112]. IEEE HOST'21

2. Megha Byali, Harsh Chaudhari, Arpita Patra and Ajith Suresh. *FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning* [32]. PoPETS'20 (CORE A)

Preprints / Manuscripts

1. Nishat Koti, Shravani Patil, Arpita Patra and Ajith Suresh. *MPClan: Protocol Suite for Privacy-Conscious Computations*. Under Submission

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of the contemporary era of computing, machine learning techniques have proven their mettle in diverse sectors, such as finance and healthcare, that involve multi-party computation (MPC) to derive genuine inferences. Increased concerns about privacy coupled with policies such as European Union General Data Protection Regulation (GDPR) make it harder for multiple parties to collaborate on machine learning (ML) computations. The emerging field of privacy-preserving machine learning (PPML) addresses this issue by offering tools to let parties perform computations without sacrificing the privacy of the underlying data. PPML can be deployed across various domains such as healthcare, recommendation systems, etc., with works like [5] demonstrating practicality.

The primary challenge that inhibits widespread adoption of PPML is that the additional demand on privacy makes the already compute-intensive ML algorithms all the more demanding in terms of high computing power and other complexity measures such as communication complexity that the privacy-preserving techniques entail. Many everyday end-users are not equipped with computing infrastructure capable of efficiently executing these algorithms. It is economical and convenient for end-users to outsource an ML task to more powerful and specialized systems. However, even while outsourcing to servers, the privacy of data must be ensured. This is addressed by the Secure Outsourced Computation (SOC) paradigm and thus is an apt fit for the moment's need. SOC allows end-users to securely outsource computation to a set of specialized and powerful cloud servers and avail of its services on a pay-per-use basis. SOC guarantees that individual data of the end-users remain private, tolerating reasonable collusion amongst the servers. Both the training and prediction phases of PPML can be realized in the SOC setting. The common approach of outsourcing followed in the PPML literature, as well as

by our work, requires the users to secret-share[1] their inputs between the set of hired (untrusted) servers, who jointly interact and compute the secret-shared output, and reconstruct it towards the users. Of late, MPC based techniques [102, 101, 120, 133, 97, 37, 32, 38, 110] have been gaining interest, where a server enacts the role of a party in the MPC protocol.

MPC [137, 64, 18], the holy-grail problem of secure distributed computing, enables a set of n mutually distrusting parties to perform joint computation on their private inputs in a way that no coalition of t parties can learn more information than the output (privacy) or affect the true output of the computation (correctness). The distrust among the parties is formalized by having an *adversary* that may corrupt some of the parties. We usually consider a *monolithic or centralized* adversary, i.e., if two or more parties are corrupted, we assume that they collude with each other. We denote the corruption threshold of the adversary by t. Under the adversary's control, the parties are called "corrupt", and the remaining parties are called "honest". This thesis focuses on designing efficient MPC frameworks for 2, 3 and 4 parties, with at most one corruption.

1.1 System Model

Adversarial Model The various traits of the adversary introduce several unique settings where MPC is explored in the literature. This thesis considers a static adversary that decides on the set of t parties it would corrupt before the protocol begins. Moreover, the adversary is computationally bounded, meaning that it is restricted to run within probabilistic polynomial time. Based on the type of corruption, an adversary can be primarily categorized into two: i) *passive / semi-honest* - where the corrupt parties follow the protocol specifications but try to learn more information than what is allowed as per the security guarantees of the protocol, and ii) *active/malicious* - where the adversary exercises total control over the corrupt parties who may deviate from the protocol steps in any arbitrary manner.

High-throughput vs Low-latency MPC MPC protocols can be categorized as high-throughput [7, 57, 8, 101, 37, 2, 38, 110, 85, 113] and low-latency [103, 109, 30, 31] protocols. The low-latency protocols are built using garbled circuits (GC) [138, 12, 82, 140] and result in constant-round solutions. Secret-sharing (SS) based solutions have been used for high-throughput protocols, but require a number of communication rounds linear in the multiplicative depth of the circuit. However, less communication than GC-based protocols facilitates several instances of SS-based protocols to be executed in parallel, leading to high

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

throughput. While high-throughput protocols enable efficient computation of functions such as addition, multiplication and dot-product, other functions such as division are best performed using garbled circuits. Activation functions such as ReLU used in neural networks (NN) alternate between multiplication and comparison, wherein multiplication is better suited to the arithmetic world and comparison to the boolean world. Hence, MPC protocols working over different representations (arithmetic/boolean/garbled circuit based) can be mixed to achieve better efficiency. The characteristics of the categories mentioned above put forth the need for a mixed-protocol framework [51, 102, 101, 120, 121, 38, 55, 113], where the protocol is split into blocks. Each block is executed in one of the following three worlds: i) Arithmetic, ii) Boolean, and iii) Garbled. While the arithmetic world performs operations on ℓ-bit rings (or fields), both boolean and garbled world perform operations on bits. Also, arithmetic and boolean worlds operate using an SS-based approach, while the garbled world uses a GC-based approach.

Almost all high-throughput protocols evaluate a circuit that represents the function f to be computed in a secret-shared fashion. Informally, the parties jointly maintain the invariant that for each wire in the circuit, the exact value over that wire is available in a secret-shared fashion among the parties so that the adversary learns no information about the exact value from the shares of the corrupt parties. Upon completion of the circuit evaluation, the parties jointly reconstruct the secret-shared function output. Intuitively, the security holds as no intermediate value is revealed during the computation. The deployed secret-sharing schemes are typically linear, ensuring non-interactive evaluation of the linear gates. The communication is required *only* for the non-linear (i.e.multiplication) gates in the circuit. The focus then turns on improving the communication overhead per multiplication gate. Recent literature has seen a range of customized linear secret-sharing schemes over a small number of parties, boosting the performance for multiplication gate spectacularly.

Pre-processing Paradigm To enhance practical efficiency, MPC protocols resort to the pre-processing paradigm, which splits the computation into two phases; a pre-processing phase where input-independent (but function-dependent), computationally heavy tasks can be computed, followed by a fast online phase utilizing the pre-processing computation [10]. Since the same functions in ML are evaluated several times, this paradigm naturally fits the case of PPML, where the ML algorithm is known beforehand. The parties can batch together the pre-computations and generate a large volume of pre-processing data to support the execution of multiple online phases. There are constructions abound that show effectiveness of this paradigm both in the theoretical [10, 13, 14, 19, 39] and practical [48, 51, 78, 49, 79, 38, 110] regime.

Fields vs Rings In yet another direction to improve practical efficiency, secure computation for arithmetic circuits over rings has gained momentum of late, while traditionally, fields have been the default choice. Computation over rings models computation in real-life computer architectures such as computation over CPU words of 32 or 64 bits. Moreover, operating over rings eliminates the need for external libraries to operate over fields (10×-100× slower) than real-world system architectures based on 32-bit and 64-bit rings. The benchmarking results of [124] and the works of [42, 20, 51, 49] have showcased the efficiency improvements of protocols compared to rings over their field counterparts. Further, recent works [78, 44, 50, 55, 76] propose MPC protocols over 32 or 64 bit rings to leverage CPU optimizations.

Security Guarantees Works such as [101, 133, 98] typically go for active security with abort, where the adversary can act maliciously to obtain the output and make honest parties abort. The stronger notion of fairness guarantees that either all or none of the parties obtain the output. This provides an incentive to the adversary to behave honestly in resources-expensive tasks such as PPML, as creating an abort scenario to cause a rerun will waste its resources. In cases where the risk of failure for the system is too high, for instance, when deploying PPML for healthcare applications, participants might want to avoid the case when none of them receives the output. The way to tackle this issue is to modify protocols to guarantee that the correct output is always delivered to the participants irrespective of an adversary's misbehaviour. This is provided by guaranteed output delivery (GOD) or robustness. A robust protocol prevents the adversary from repeatedly causing the computations to rerun, thereby upholding the trust in the system.

Robustness is crucial for real-world deployment and usage of PPML techniques. Consider the following scenario wherein an ML model owner wishes to provide inference service. The model owner shares the model parameters between the servers, while the end-users share their queries. A protocol that provides security with abort or fairness will not suffice. In both cases, a malicious adversary can lead to the protocol aborting, resulting in the user not obtaining the desired output. This leads to denial of service and heavy economic losses for the service provider. For data providers, as more training data leads to more accurate models, collaboratively building a model enables them to provide better ML services, and consequently, attract more clients. A robust framework encourages active involvement from multiple data providers. Hence, for the seamless adoption of PPML solutions in the real world, the protocol's robustness is of utmost importance.

MPC for small number of parties While MPC, in general, has been a subject of extensive research, the area of MPC with a small number of parties [103, 51, 7, 102, 36, 101, 30] has drawn popularity of late mainly due to its efficiency and simplicity. Furthermore, most real-time applications involve up to 5 parties. Applications such as statistical and financial data analysis [22], email-filtering [89], distributed credential encryption [103], Danish sugar beet auction [23] involve 3 parties. Well-known MPC frameworks such as VIFF [59], Sharemind [20] have been explored with three parties. Recent advances in secure machine learning (ML) based on MPC have shown applications with small number of parties [102, 101, 120, 133, 97, 37, 32, 38, 110, 113]. MPC with small parties aids in solving MPC over a large population via server-aided computation, where a small number of servers jointly hold the input data of the large population and run an MPC protocol evaluating the desired function.

Our protocols designed for 2, 3 and 4 parties operating over rings are cast in the pre-processing paradigm and achieve robustness. Before moving on to the contributions of the thesis, we outline the relevant literature next.

1.2 Related Work

In the regime of PPML using MPC, the initial works considered the widely-used ML algorithms such as Decision Trees [93], K-Means Clustering [74, 28], Support Vector Machines [139, 132], Linear Regression [53, 54, 123] and Logistic Regression [128]. However, these solutions are far from practical reach due to the huge performance overheads that they incur. We next discuss the literature concerning the following three algorithms – Linear Regression, Logistic Regression, and Neural Networks, which are the focus of this thesis. The initial set of practical solutions for these algorithms were proposed in the dishonest majority (two-party) setting and are discussed below.

Linear Regression: Privacy-preserving linear regression on the two server model was first proposed by Nikolaenko et al. [105]. Their solution focused on horizontally partitioned data and used a combination of linearly homomorphic encryption (LHE) and garbled circuits. Later, Gascon et al. [58] and Giacomelli et al.[60] extended these results to vertically partitioned data. Both papers, however, confine the problem to solving a linear system using Yao's garbled circuit protocol, which has a substantial training time overhead and cannot be applied to non-linear models. SecureML [102] then used stochastic gradient descent (SGD) for training, as well as a mix of arithmetic, binary, and Yao sharing (using the ABY [51] framework) over two parties, to increase the performance of linear regression over horizontally partitioned data. Furthermore, they present a unique design for approximation fixed-point multiplication that avoids boolean

operations for truncating decimal numbers while providing state-of-the-art performance for training linear regression models.

Logistic Regression: Wu et al. [136] explored privacy-preserving logistic regression and proposed approximating the logistic function with polynomials and training the model with LHE, with the complexity being exponential in the degree of the approximation polynomial. Aono et al. [6] considered a different security model where an additional untrusted server collects and mixes encrypted data from several clients and delivers it to a trusted client who trains the model on the plaintext on clear.

Neural Networks: Privacy-preserving solutions for neural networks have also been studied. For the case of training, Shokri and Shmatikov [125] proposed a scheme where the two servers locally train their model using the horizontally partitioned data. Instead of exchanging the training data, they only share the changes in a portion of the coefficients in the locally trained model. Although the system is very efficient (no cryptographic operations are required), the leakage resulting from sharing these coefficient changes remains unclear, and no formal security guarantees are provided. The privacy-preserving training of neural networks was also considered in the work of SecureML [102], where the ABY framework was customized to achieve a new approximate fixed-point multiplication protocol that avoids binary circuits. For the case of inference, the works of [61, 69, 35, 25] consider fully homomorphic or somewhat homomorphic encryption to evaluate the model on encrypted data, while [95, 122] uses a combination of LHE and garbled circuits.

Departing from the dishonest majority setting, a performance breakthrough in the above-mentioned PPML algorithms was observed in ABY3 [101], which explored the honest majority setting for three parties. After that, a plethora of works followed, such as [37, 133, 110, 38, 32, 134, 85, 46, 87], which explored the setting of small population with honest-majority and showcased real-time efficiency even for complex neural-network architectures such as LeNet [91] and VGG16 [127].

While the literature above tackles only the line of works in PPML via MPC, other dimensions such as differential privacy, model attacks and defense mechanisms, etc., are relevant. However, the literature elaborating on the line of development in these areas is quite vast to be briefly explained in this section, and we refer the reader to [129, 100, 96, 33] for a detailed overview of the same. Next, we provide an elaborate summary of the most relevant related work that focuses on MPC frameworks for PPML.

Honest Majority ABY3 [101] was the first framework for the case of 3 parties, supporting both training and inference. It had variants for both passive and active security, with the former

being based on [7] and the latter on [57, 8]. ASTRA [37] improved upon the 3PC of [7, 57, 8] by proposing faster protocols for the online phase with active security. As a result, secure inference of ASTRA is faster than ABY3. Building on [24], BLAZE [110] proposed an actively secure framework that supports the inference of neural networks. BLAZE pushes the expensive zero-knowledge part of the computation to the preprocessing phase, making its online phase faster than that of [24]. SWIFT (3PC) improved upon BLAZE by using the distributed zero-knowledge protocol of [27], thereby achieving GOD. In an orthogonal line of work, [133, 134] focused on enhancing the efficiency of actively secure protocols for large convolutional neural networks, supporting training and inference.

In the high-throughput setting for 4PC, [66] explores protocols for the security notions of abort. Inspired by the theoretical GOD construction in [66], [32] proposed practical protocols with GOD for secure inference. Trident [38] improved protocols (in terms of communication) compared to [66] with a focus on security with fairness. In addition, it was the first work to propose a mixed-protocol framework for the case of 4 parties. More recently, [98] improved over [66] to provide support for fixed-point arithmetic with applications to graph parallel computation, albeit with abort security. Improving the security of Trident to GOD, SWIFT [85] presented an efficient, robust PPML framework with protocols as fast as Trident. SWIFT only supports the secure inference of neural networks and lacks conversions similar to Trident and the garbled world. Fantastic Four [46] also provides robust 4PC protocols which are on par with SWIFT. While they claim to provide a better security model called *private robustness* compared to SWIFT, it has been shown in SWIFT that the two security models are theoretically equivalent.

In the regime of constant-round protocols, [103] presents 3PC protocols in the honest majority setting satisfying security with abort, which require communicating one garbled circuit and three rounds of interaction. The work of [72] presents a robust 4-party computation protocol (4PC) with GOD in 2-rounds (which is optimal) at the expense of 12 garbled circuits. Further, [30] presents efficient 3PC and 4PC constructions providing security notions of fairness and GOD.

Dishonest Majority The works of [48, 77] proposed efficient SS-based solutions for the dishonest majority setting over fields, which was then extended to the ring setting in [44]. The solution involves the generation of Beaver multiplication triples [10] in the setup phase and evaluation of the circuit (multiplication gates) in the online phase using the generated triples. For the 2PC case, the approach mentioned above requires two public reconstructions among the parties per multiplication gate in the online phase. Later, works like [78, 79, 107] focused

on improving the setup cost using techniques like Oblivious Transfer (OT) and Homomorphic Encryption (HE). [17] improved the number of public reconstructions required in the online phase from two to one using a function-dependent preprocessing but requires additional communication of four ring elements in the preprocessing phase.

In this line of work, the GMW protocol [64] takes a function represented as a Boolean circuit (i.e., $\ell = 1$), and the values are secret-shared using XOR-based secret sharing. To precompute, a multiplication triple, the solution of [9] proposed a solution which uses 1-out-of-2 Oblivious Transfer (OT), which was later improved by factor 1.2× by [52] using the 1-out-of-N OT extension of [81].

Mixed-protocols A mixed-protocol framework for MPC was first shown to be practical, in the 2-party dishonest majority setting, by TASTY [83, 68]. TASTY was a passively secure compiler supporting generation of protocols based on homomorphic encryption and garbled circuits. This was followed by ABY [51], which proposed a mixed protocol framework, also with passive security, combining the arithmetic, boolean and garbled worlds. The recent work of ABY2 [113] improves upon the ABY framework, providing a faster online phase with applications to PPML. The work of [121, 55] proposed efficient mixed world conversions for the case of n parties with a dishonest majority. Both works have active security, with [121] supporting the inference of SVMs, and [55] supporting neural network inference.

In the honest majority setting, ABY3 [101] extended the idea to 3 parties and provided specialized protocols for the case of PPML. ABY3 was the first work to support secure training in the case of 3 parties, while Trident [38, 87] extended it to the 4-party setting.

HyCC [29] provides a compiler to automatically partition a function (specified in ANSI C) into sub-functions such that each sub-function is evaluated with either Arithmetic sharing, Boolean sharing or GCs. The partitioning takes into account the real-world setup, such as the network between the parties. The work of [73] has shown a method to find an optimal partitioning in polynomial time.

Multi-Input Multiplication In the boolean setting, [52] extended two-input AND gates to the general N-input case using lookup tables. [106] extended the multiplication from two-input to arbitrary input using Beaver triple extension with a focus on minimizing the online rounds. However, the online communication of [106] scale with the fan-in of the multiplication gates. [113] improved [106] and achieved an online communication of 2 ring elements. Recently, [87] extended the technique of [113] to the four-party honest majority setting.

1.3 The Contribution of this Thesis

In the dominion of PPML consisting of a small number of parties which is of practical interest to the community, we propose **MPCLeague**, an efficient and robust PPML platform for 2,3 and 4 parties with different corruption thresholds. In the honest majority setting, we explore protocols with three and four parties, amongst which at most one can be maliciously corrupt. In the dishonest majority setting, we consider the two-party setting with only semi-honest corruption as achieving robustness with malicious corruption is proven to be impossible in the dishonest-majority setting [40]. While some of our protocols are the first of a kind in their setting (robust 3PC and 4PC), the rest of the protocols improve upon their counterparts in the literature by several orders of magnitude.

A major contribution of the thesis lies in unifying the protocol design of all four settings. This results in much simpler protocols and brings in efficiency improvements over the prior versions [37, 110, 38, 85, 113]. All our protocols fall back to a generalized architecture of 3 layers as shown in Figure 1.1. The first layer forms the foundation of our constructions designed using MPC protocols, which is then built upon by the second layer to obtain the building blocks. Finally, layer 3 utilizes layers 1 and 2 to give rise to the realization of privacy-preserving ML algorithms, thus forming the end goal of our architecture. We elaborate on this next, starting with the base layer.

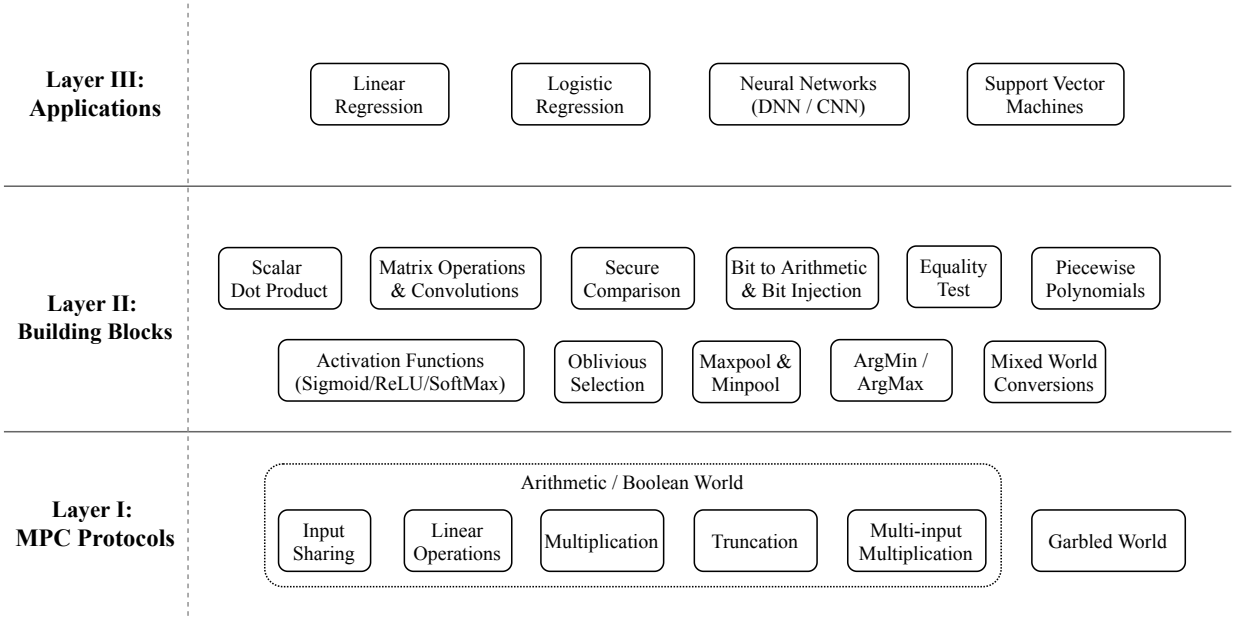


Figure 1.1: Three-layer Architecture of **MPCLeague**

1.3.1 Layer I

Layer I consisting of MPC protocols (ASTRA, SWIFT, Tetrad, ABY2.0) form the basis of our architecture. We aim to realize efficient primitive operations such as input sharing, multiplication, and output reconstruction for all the considered frameworks. Although inspired by Beaver's multiplication-triple method [10], our multiplication protocol, which lies at the heart of this layer, adopts a new perspective that aids in realizing several efficient primitives discussed in §2.1. We believe that our new perspective can bring several further optimizations where Beaver's randomization technique is currently being used.

To preserve privacy, we rely on computation and evaluation using our customized secret-sharing technique. This technique has two main advantages: It allows our protocols to be cast in the preprocessing paradigm leading to a blazing fast online phase. Further, it helps in minimizing the number of parties that need to be active for the majority of the computation in the online phase (cf. Table 1.1). We use the sharing over both $\mathbb{Z}_{2^\ell}$ and its special instantiation $\mathbb{Z}_{2^1}$ and refer to them as *arithmetic* and *boolean* sharing respectively.

In most MPC-based PPML frameworks, we observe that a large part of the computation is done over the arithmetic and boolean worlds. The garbled world is used only to perform the non-linear operations (e.g. softmax) that are expensive in the arithmetic/boolean world and switched back immediately after. Leveraging this observation, we propose tailor-made garbled world protocols with *end-to-end* conversion techniques. These protocols have the following advantages over the standalone variants – i) no use of commitments for the inputs, and ii) no requirement of an explicit input sharing and output reconstruction phase, as explained later in the thesis.

Inspired by [113, 106], we extend our multiplication protocol to the multi-input case, allowing multiplication of 3 and 4 inputs in one online round. Naively, performing a 4-input multiplication follows a tree-based approach, and the required communication is that of three 2-input multiplications and two online rounds. Our contribution lies in keeping the communication and the round of the online phase the same as that of 2-input multiplication (i.e. invariant of the number of inputs) by trading off the preprocessing cost. Looking ahead, multi-input multiplication, when coupled with the optimized parallel prefix adder circuit from [113], brings in a 2× improvement in online rounds. It also cuts down the online communication of secure comparison, impacting PPML applications.

1.3.2 Layer II

Layer II defines the building blocks that form the core of our architecture. The primary building blocks constitute scalar dot product, secure comparison, piece-wise polynomials and mixed world conversions. Although our building blocks improve over the state-of-the-art, our main contributions lie in the efficient realization of scalar dot product and mixed world conversions highlighted below.

A naive approach to perform the dot product operation on two d-length vectors is to perform d multiplications followed by adding the results. However, this leads to communication proportional to the length of the vectors. Our constructions remove the dependency of the communication on the length of the vectors in the setting of 3 and 4 parties. This is achieved for the first time in the setting of 3 parties with one active corruption. Moreover, in the 2PC literature, our construction achieves an online communication independent of the vector length for the first time.

# Parties	Reference[a]	#Active Parties[b]	Security	Dot Product[c]		Dot Product with Truncation		Conversions[d]		
				Comm_{pre}[e]	Comm_{on}	Comm_{pre}	Comm_{on}	A	B	G
3	ABY3 [101]	3	semi-honest	–	3ℓ	$\approx 6\ell$	4ℓ	✓	✓	✓
	ASTRA [37]	2	semi-honest	1ℓ	2ℓ	1ℓ	2ℓ	✓	✓	✓
	ABY3 [101]	3	Abort	$12\mathsf{d}\ell$	$9\mathsf{d}\ell$	$12\mathsf{d}\ell + 84\ell$	$9\mathsf{d}\ell + 3\ell$	✓	✓	✓
	SWIFT [110, 85]	2	Robust	3ℓ	3ℓ	9ℓ	3ℓ	✓	✓	✓
4	Mazloom et al. [98]	4	Abort	2ℓ	4ℓ	2ℓ	4ℓ	✓	✓	✗
	Trident [38]	3	Fair	3ℓ	3ℓ	6ℓ	3ℓ	✓	✓	✓
	Tetrad [87]	2	Fair	2ℓ	3ℓ	2ℓ	3ℓ	✓	✓	✓
	SWIFT (4PC) [85]	2	Robust	3ℓ	3ℓ	4ℓ	3ℓ	✓	✓	✗
	Fantastic Four [46] (Best)[f]	4	Robust	–	6ℓ	ℓ	9ℓ	✓	✓	✗
	Fantastic Four [46] (Worst)	3	Robust	–	$6(\ell+\kappa)$	$\approx 80\ell + 76\kappa$	$9\ell + 6\kappa$	✓	✓	✗
	Tetrad [87]	2	Robust	2ℓ	3ℓ	2ℓ	3ℓ	✓	✓	✓
2	SecureML [102]	2	semi-honest	$2\mathsf{d}\ell(\kappa+\ell)$	$4\mathsf{d}\ell$	$2\mathsf{d}\ell(\kappa+\ell)$	$4\mathsf{d}\ell$	✓	✓	✓
	ABY2.0 [113]	2	semi-honest	$2\mathsf{d}\ell(\kappa+\ell)$	2ℓ	$2\mathsf{d}\ell(\kappa+\ell)$	2ℓ	✓	✓	✓

[a] Amortized costs are reported for 1 million operations [b] parties that carry out most of the computation during online phase [c] ℓ - size of ring in bits, κ - security parameter, d - length of the vectors. [d] A, B, G indicate support for arithmetic, boolean, and garbled worlds respectively [e] 'Comm' - communication, 'pre' - preprocessing, 'on' - online [f] cf. §5.2.6.1 for details

Table 1.1: Comparison of MPC frameworks (small no. of parties) for PPML.

For an operation that requires computing over the garbled domain in the mixed-world computation, the standard approach is to first switch from *Arithmetic to Garbled* and evaluate the garbled circuit to obtain a garbled-shared output. These shares are brought back to the arithmetic domain using a *Garbled to Arithmetic* conversion. Deviating from the standard approach, we propose new end-to-end conversion techniques that improve the round complexity by 2×. On a high level, our approach is to modify the garbled circuit such that the output

is in the arithmetic domain. This eliminates the need for an explicit *Garbled to Arithmetic* conversion, saving in both communication and rounds in the online phase. More generally, end-to-end conversions are of the form "x-Garbled-x" where x can be either arithmetic or boolean and need a single round for the garbled world.

We summarize and compare the efficiency of layer II protocols with the state-of-the-art in Table 1.1. We showcase the cost for a dot product operation in that table as it forms the fundamental building block of most PPML algorithms. As most computations in the PPML domain operate on decimal values, we provide the cost comparison for dot-product with truncation in the table. Finally, we highlight the conversions supported by our protocols and that of the stat-of-the-art.

1.3.3 Layer III

Layer III constitutes the realizations of the PPML algorithms that are widely used. We are the first to propose a robust PPML framework in the literature of three and four parties. We demonstrate the practicality of the framework, which combines the arithmetic, boolean, garbled worlds via benchmarking over a Wide Area Network (WAN), instantiated using n1-standard-64 instances of Google Cloud. We consider the training and inference phases of linear regression, logistic regression and deep neural networks such as LeNet [91] and VGG16 [127] along with the inference phase of Support Vector Machines.

The implementation section is presented through the lens of deployment scenarios with two different goals. Participants in the first scenario are interested in the shortest online runtime for the computation, whereas participants in the second one want to minimize the deployment cost. Correspondingly, there are variants of our framework that cater to both scenarios. The time-optimized (T) variant has the fastest online phase considering online runtime as the metric. On the other hand, the cost-optimized (C) variant aims at minimizing deployment cost. This is measured via *monetary cost* [116], which helps to capture the effect of the total runtime of the parties, and communication together.

1.4 Organization of the Thesis

The thesis is categorized into three parts. Each part represents a layer of the architecture (Figure 1.1) consisting of chapters devoted to ASTRA, SWIFT, Tetrad, ABY2.0 frameworks. Moreover, chapters in each part are preceded by an overview. Table 1.2 summarizes the organization of these chapters.

Framework	Setting	Security	3-Layer Architecture (Figure 1.1)		
			Layer I	Layer II	Layer III
ASTRA	3PC	semi-honest	Chapter 3	Chapter 7	Chapter 11
SWIFT	3PC	robust	Chapter 4	Chapter 8	Chapter 12
Tetrad	4PC	robust	Chapter 5	Chapter 9	Chapter 13
ABY2.0	2PC	semi-honest	Chapter 6	Chapter 10	Chapter 14

Table 1.2: Organization of the thesis

The preliminaries and conclusion of the thesis appear in Chapter 2 and 15 respectively.

Chapter 2

Preliminaries

This chapter presents the relevant background, including the notation, definitions, security model and an overview of some of the standard primitives used in our constructions.

2.1 High Level Overview of Our Approach

The MPC protocols in our framework rely on the well-known Beaver's circuit randomization technique [10] but use a different perspective of the technique. This section presents a high-level overview of our scheme and a side-by-side comparison with Beaver's technique. The highlight of our scheme is its effectiveness towards efficient realizations for multiple input multiplication gates and dot product operations, as will be explained later in this thesis. For simplicity, consider two parties P_1, P_2 with values a, b secret-shared among them who want to compute a multiplication gate with output $\mathsf{z} = \mathsf{ab}$.

Beaver's technique [10] on gate inputs (cf. left of Figure 2.1) In Beaver's[10] circuit randomization technique (cf. left side of Figure 2.1), the inputs of the multiplication gate are randomized first and the corresponding correlated randomness is generated independently (preferably in a setup phase). In detail, parties interactively generate an additive sharing of the multiplication triple $(\delta_\mathsf{a}, \delta_\mathsf{b}, \delta_\mathsf{ab})$ with $\delta_\mathsf{ab} = \delta_\mathsf{a}\delta_\mathsf{b}$ during the setup phase before the actual inputs are known. Now, we can write

$$\begin{aligned}\mathsf{a} \cdot \mathsf{b} &= ((\mathsf{a} + \delta_\mathsf{a}) - \delta_\mathsf{a})((\mathsf{b} + \delta_\mathsf{b}) - \delta_\mathsf{b}) \\ &= (\mathsf{a} + \delta_\mathsf{a})(\mathsf{b} + \delta_\mathsf{b}) - (\mathsf{a} + \delta_\mathsf{a})\delta_\mathsf{b} - (\mathsf{b} + \delta_\mathsf{b})\delta_\mathsf{a} + \delta_\mathsf{ab}.\end{aligned}$$

Let $\Delta_\mathsf{a} = (\mathsf{a} + \delta_\mathsf{a})$ and $\Delta_\mathsf{b} = (\mathsf{b} + \delta_\mathsf{b})$ be the randomized versions of the input values of a

multiplication gate. Then, during the online phase, parties locally compute an additive sharing of Δ_{a} using additive shares of a and δ_{a}. Similarly, an additive sharing of Δ_{b} is computed. This is followed by the parties mutually exchanging the shares of Δ_{a} and Δ_{b} to enable public reconstruction of Δ_{a} and Δ_{b}. Then using the above equation, parties can locally compute a sharing of $\mathsf{a}\cdot\mathsf{b}$. Note that this method requires reconstruction of two elements per multiplication gate. We observe that the communication is required for enabling parties to obtain the value of Δ_{a} and Δ_{b} in clear.

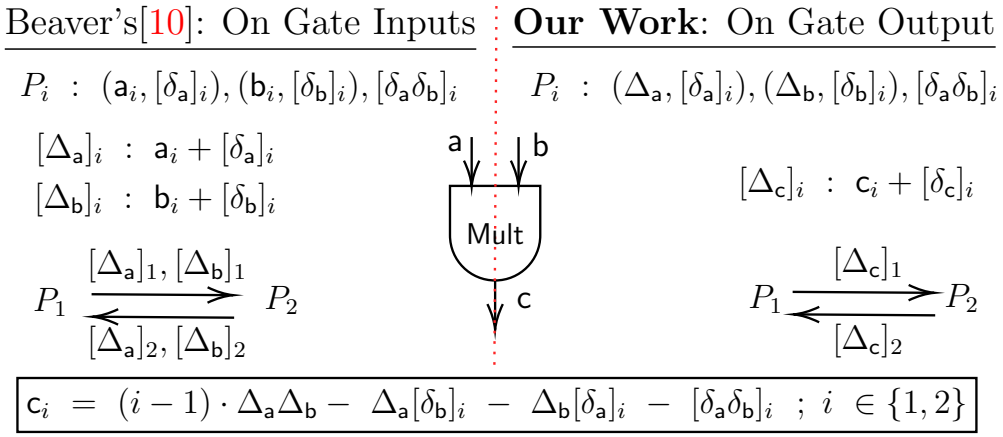


Figure 2.1: High level overview of Beaver's[10] and **Our Work**

Our technique on gate outputs (cf. right of Figure 2.1) With this insight, we modify the sharing semantics so that the parties are ensured to have the Δ value as a part of their share, corresponding to every wire value (including the inputs of a multiplication gate). As a result, the reconstructions of Δ_{a} and Δ_{b} are no longer required. This may give the wrong impression that no communication is required for evaluating a multiplication gate. It is true that now the parties can locally evaluate the additive sharing of $\mathsf{z} = \mathsf{ab}$. But to proceed further, a sharing for z according to the new sharing semantics needs to be generated. This requires both parties to obtain Δ_{z} in the clear. Hence, the parties locally compute an additive sharing of Δ_{z} using the shares of z computed earlier and mutually exchange their shares to reconstruct Δ_{z}.

Our technique, in summary, shifts the need for reconstruction (which alone causes communication for a multiplication gate) from per input wire to the *output* wire alone for a multiplication gate. For a traditional 2-input multiplication gate, we reduce the number of reconstructions (each involves sending two elements) from 2 to 1. As a result, we improve communication by a factor of $2\times$. The impact is much higher for an N-input multiplication gate and a scalar product of two N-dimensional vectors. For scalar product, Beaver's circuit re-randomization

required $2N$ reconstructions, whereas our techniques need a *single* one, offering a gain of $2N\times$. Our constructions can be generalized to the n-party scenario (which is out of scope for this work) and bring a significant pay-off, as the cost per reconstruction depends linearly on the number of parties.

2.2 Parameters and Notation

In our framework, we have $n \in \{2, 3, 4\}$ parties, denoted by $\mathcal{P}$ that are connected by pair-wise private and authentic channels in a synchronous network, and an adversary that can corrupt at most one party. Our protocols are designed to work over an ℓ-bit ring denoted by $\mathbb{Z}_{2^\ell}$. κ denotes the computational security parameter. In our implementation, we use $\ell = 64$ and $\kappa = 128$. Our protocols are cast into an *input-independent* preprocessing phase and an *input-dependent* online phase. Our protocols work over the arithmetic ring $\mathbb{Z}_{2^\ell}$ or boolean ring $\mathbb{Z}_{2^1}$.

Secure Outsourced Computation (SOC) In the secure outsourced computation (SOC) setting, the servers hired to carry out the computation enact the role of the parties mentioned above. For ML training, data owners who want to train a model collaboratively secret-share their data among the servers. For ML inference, a data owner shares its model while the client shares its query among the servers. The servers carry out the computation on secret-shared data and obtain the output in a secret-shared fashion. In the case of training, the output is reconstructed towards the data owners, whereas in the case of inference, the output is reconstructed towards the client. We assume that the corrupt server can collude with an arbitrary number of data-owners in the case of training. In contrast, we assume that the corrupt server can collude with the model owner or the client for inference. In the case of inference, since the query response is available in the clear to the client, we do not guarantee the privacy of the training data against attacks such as attribute inference, membership inference, or model inversion [56, 131, 126]. This is an orthogonal problem, and we consider it as an out-of-scope of this thesis.

Dealing with decimal values For applications such as machine learning where the inputs are decimal numbers, we use the Fixed-Point Arithmetic (FPA) [101, 37, 110, 38, 32] representation to embed the value in the underlying ring $\mathbb{Z}_{2^\ell}$. Decimal value is treated as an ℓ-bit integer in signed 2's complement representation. The most significant bit (msb) represents the sign bit, and x least significant bits are reserved for the fractional part. The ℓ-bit integer is then

treated as an element of $\mathbb{Z}_{2^\ell}$, and operations are performed modulo 2^ℓ. For our implementation, we use $\ell = 64$, and $x = 13$, with $\ell - x - 1$ bits for the integral part.

Vectors and Matrices For a vector $\vec{\mathbf{a}}$, a_i denotes the i^{th} element in the vector. For two vectors $\vec{\mathbf{a}}$ and $\vec{\mathbf{b}}$ of length d, the dot product is given by, $\vec{\mathbf{a}} \odot \vec{\mathbf{b}} = \sum_{i=1}^{\mathsf{d}} \mathsf{a}_i \mathsf{b}_i$. Given two matrices $\mathbf{A}, \mathbf{B}$, the operation $\mathbf{A} \circ \mathbf{B}$ denotes the matrix multiplication.

Notation 2.1 *For a bit* $\mathsf{b} \in \{0,1\}$, b^{R} *denotes the representation of the bit value* b *over the arithmetic ring* $\mathbb{Z}_{2^\ell}$. *In detail, all the bits of* b^{R} *will be zero except for the least significant bit, which is set to* b.

Table 10.2 depicts notation that we use throughout the thesis.

nPC	n-party computation; $n \in \{2,3,4\}$ in this thesis
$\mathcal{P}$	Set of all parties performing secure computation; 2PC: $\mathcal{P} = \{P_1, P_2\}$, 3PC: $\mathcal{P} = \{P_1, P_2, P_3/P_0\}$, 4PC: $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$
$\mathbb{Z}_{2^\ell}$	Ring of size ℓ bits; $\ell = 64$ in this thesis
κ	Symmetric security parameter; $\kappa = 128$ in this thesis
a_i	i^{th} element of vector $\vec{\mathbf{a}}$
$\vec{\mathbf{a}} \odot \vec{\mathbf{b}}$	Scalar dot product between vectors $\vec{\mathbf{a}}$ and $\vec{\mathbf{b}}$ of length d
$\mathbf{X} \circ \mathbf{Y}$	Multiplication of two matrices $\mathbf{X}$ and $\mathbf{Y}$
$s \in \{\mathbf{A}, \mathbf{B}, \mathbf{G}\}$	Type of sharing: **A**rithmetic, **B**oolean, or **G**arbled
b^{R}	Representation of the bit value $\mathsf{b} \in \{0,1\}$ over the arithmetic ring $\mathbb{Z}_{2^\ell}$
$\overline{\mathsf{b}}$	Complement value $1 \oplus \mathsf{b}$ for bit $\mathsf{b} \in \{0,1\}$
$\mathsf{H}(\cdot)$	A *collision-resistant* hash function
PRF	Pseudo-random Function
FPA	Fixed-point Arithmetic; x denotes the precision and $x = 13$ in this thesis
msb / lsb	Most / Least Significant Bit
OT	Oblivious Transfer
cOT_ℓ^n	n instances of Correlated OT on ℓ-bit strings
HE	Homomorphic Encryption
PPT	Probabilistic-polynomial Time
PPA	Parallel-prefix Adder

Table 2.1: Notations used throughout this thesis.

2.3 Definitions

Definition 2.1 *(Negligible functions) A function* negl *is negligible iff* $\forall c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}$ *such that* $\forall n > n_0, \mathsf{negl}(n) < n^{-c}$.

2.4 Security Model

We prove the security of our protocols using the real-world/ ideal-word simulation paradigm [63, 92]. The security of protocols is analyzed by comparing what an adversary can do in the real world execution of the protocol with what it can do in an ideal world execution that is considered secure by definition (where there exists a trusted third party, denoted as ttp). In the ideal world, the parties send their inputs to the trusted third party over perfectly secure channels that carries out the computation and send the output to the parties. Informally, a protocol is said to be secure if whatever an adversary can do in the real world can also be done in the ideal world. We refer the readers to [34, 62, 41, 92] for further details regarding the security model.

Let $\mathcal{A}$ denote the probabilistic polynomial time (PPT) real-world adversary corrupting at most one party in $\mathcal{P}$, $\mathcal{S}$ denote the corresponding ideal world adversary, and $\mathcal{F}$ denote the ideal functionality. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S}}(1^\kappa, z)$ denote the joint output of the honest parties and $\mathcal{S}$ from the ideal execution with respect to the security parameter κ and auxiliary input z. Similarly, let $\text{REAL}_{\Pi,\mathcal{A}}(1^\kappa, z)$ denote the joint output of the honest parties and $\mathcal{A}$ from the real world execution. We say that the protocol Π securely realizes $\mathcal{F}$ if for every PPT adversary $\mathcal{A}$ there exists an ideal world adversary $\mathcal{S}$ corrupting the same parties such that $\text{IDEAL}_{\mathcal{F},\mathcal{S}}(1^\kappa, z)$ and $\text{REAL}_{\Pi,\mathcal{A}}(1^\kappa, z)$ are computationally indistinguishable.

Definition 2.2 *For $n \in \mathbb{N}$, let $\mathcal{F}$ be a functionality and let Π be a n-party protocol. We say that Π securely realizes $\mathcal{F}$ if 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:*

$$\text{IDEAL}_{\mathcal{F},\mathcal{S}} \overset{c}{\approx} \text{REAL}_{\Pi,\mathcal{A}}.$$

We analyze the security guarantees of correctness and privacy separately in all our security proofs since we consider deterministic functionalities alone in this thesis [92].

Ideal Functionalities. [41, 65] For the secure computation of a function f using MPC, we define the ideal functionalities $\mathcal{F}_{\text{SEMI}}$, $\mathcal{F}_{\text{ABORT}}$, $\mathcal{F}_{\text{FAIR}}$, and $\mathcal{F}_{\text{GOD}}$ in Fig. 2.2, Fig. 2.3, Fig. 2.4, and Fig. 2.5 respectively.

Functionality $\mathcal{F}_{\text{SEMI}}$

Every party $P_i \in \mathcal{P}$ ($i \in [n]$) sends its input x_i to the functionality.

Input: On message (Input, x_i) from P_i ($i \in [n]$), do the following: if $(\mathsf{Input}, *)$ already received from P_i, then ignore the current message. Otherwise, record $x'_i = x_i$ internally.

Output: Compute $y = f(x'_1, \ldots, x'_n)$ and send (Output, y) to all parties.

Figure 2.2: Semi-honest functionality for computing function f

Functionality $\mathcal{F}_{\text{ABORT}}$

Every honest party $P_i \in \mathcal{P}$ ($i \in [n]$) sends its input x_i to the functionality. Corrupted parties may send arbitrary inputs as instructed by the adversary. While sending the inputs, the adversary is also allowed to send a special `abort` command.

Input: On message (Input, x_i) from P_i ($i \in [n]$), do the following: if $(\mathsf{Input}, *)$ already received from P_i, then ignore the current message. Otherwise, record $x'_i = x_i$ internally. If x_i is outside P_i's domain, consider $x'_i = \texttt{abort}$.

Output to adversary: If there exists an $i \in [n]$ such that $x'_i = \texttt{abort}$, send $(\mathsf{Output}, \perp)$ to all the parties. Else, compute $y = f(x'_1, \ldots, x'_n)$ and send (Output, y) to the adversary.

Output to selected honest parties: Receive (select, I) from adversary, where I denotes a subset of the honest parties. If an honest party belongs to I, send (Output, y), else send $(\mathsf{Output}, \perp)$, where $y = f(x'_1, \ldots, x'_n)$. We require that I includes all honest parties in case the adversary corrupts no party actively.

Figure 2.3: Abort functionality for computing function f

Functionality $\mathcal{F}_{\text{FAIR}}$

Every honest party $P_i \in \mathcal{P}$ ($i \in [n]$) sends its input x_i to the functionality. Corrupted parties may send arbitrary inputs as instructed by the adversary. While sending the inputs, the adversary is also allowed to send a special `abort` command.

Input: On message (Input, x_i) from P_i ($i \in [n]$), do the following: if $(\mathsf{Input}, *)$ already received from P_i, then ignore the current message. Otherwise, record $x'_i = x_i$ internally. If x_i is outside P_i's domain, consider $x'_i = \texttt{abort}$.

Output: If there exists an $i \in [n]$ such that $x'_i = \texttt{abort}$, send $(\mathsf{Output}, \perp)$ to all the parties. Else, compute $y = f(x'_1, \ldots, x'_n)$ and send (Output, y) to all parties.

Figure 2.4: Fair functionality for computing function f

Functionality $\mathcal{F}_{\text{GOD}}$

Every honest party $P_i \in \mathcal{P}$ ($i \in [n]$) sends its input x_i to the functionality. Corrupted parties may send arbitrary inputs as instructed by the adversary.

Input: On message (Input, x_i) from P_i ($i \in [n]$), do the following: if $(\mathsf{Input}, *)$ already received from P_i, then ignore the current message. Otherwise, record $x'_i = x_i$ internally. If x_i is outside P_i's domain, consider x'_i to be some predetermined default value.

Output: Compute $y = f(x'_1, \ldots, x'_n)$ and send (Output, y) to all parties.

Figure 2.5: GOD functionality for computing function f

2.5 Primitives

2.5.1 Shared-Key Setup

To enable parties to non-interactively sample a random value, parties rely on a one-time shared key-setup [101, 37, 110, 38, 32, 85, 113], denoted by $\mathcal{F}_{\text{KEY}}$. The key-setup can be instantiated using any standard MPC protocol in the respective setting. The key-setup establishes random keys among the parties for a pseudo-random function (PRF) which can be instantiated, for instance, using AES in counter mode.

Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function (PRF), with co-domain X being $\mathbb{Z}_{2^\ell}$. In $\mathcal{F}_{\text{KEY}}$, the key $k_{\mathcal{P}}$ is established among all the parties in $\mathcal{P}$. In addition, the following set of keys are established depending on the underlying framework.

1. Three-party frameworks (ASTRA & SWIFT):
   - One key between every pair – k_{ij} for P_i, P_j.
2. Four-party framework (Tetrad):
   - One key between every pair – k_{ij} for P_i, P_j.
   - One key between every set of three parties – k_{ijk} for P_i, P_j, P_k.

A simple instantiation for the case of ASTRA with $\mathcal{P} = \{P_0, P_1, P_2\}$ is as follows. P_0 samples key $k_{0i}, k_{\mathcal{P}}$ and sends to P_i for $i \in \{1, 2\}$. P_1 samples k_{12} and sends to P_2. The instantiations for other frameworks can be derived similarly.

2.5.2 Collision Resistant Hash Function

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 $x_1, x_2 \in_R \{0,1\}^m$.

2.5.3 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 server 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 $(c, o) = (\mathcal{H}(x||r), x||r) = \mathsf{Com}(x; r)$.

2.5.4 Replicated Secret Sharing [43]

Informally, a t-out-of-n replicated secret sharing scheme distributes a secret among n parties in such a way that any group of $t+1$ or more parties can together reconstruct the secret but no group of fewer than $t+1$ parties can. We present the formal definition below.

Definition 2.3 *A t-out-of-n replicated secret sharing scheme, defined for a finite set of secrets K and a set of $\mathcal{P}$ parties, comprises of two protocols– Sharing* (Sh) *and Reconstruction* (Rec)*, with the following requirements:*

- *Correctness. The secret can be reconstructed by any set of $(t+1)$ parties via* Rec*. That is, $\forall s \in K$ and $\forall S = \{i_1, \ldots i_{t+1}\} \subseteq \{1, \ldots n\}$ of size $(t+1)$, $\Pr[\mathsf{Rec}(s_{i_1} \ldots s_{i_{t+1}}) = s] = 1$.*

- *Privacy. Any set of t parties cannot learn anything about the secret from their shares. That is: $\forall s^1, s^2 \in K$, $\forall S = \{i_1, \ldots i_t\} \subseteq \{1, \ldots n\}$ of size t, and for every possible vector of shares $\{s_j\}_{j \in S}$, $\Pr[\{\{\mathsf{Sh}(s^1)\}_S = \{s_j\}_{i_j \in S}] = \Pr[\{\{\mathsf{Sh}(s^2)\}_S = \{s_j\}_{i_j \in S}]$, where $\{\mathsf{Sh}(s^i)\}_S$ denotes the set of shares assigned to the set S as per* Sh *when s^i is the secret for $i \in \{1, 2\}$.*

2.5.5 Garbling scheme and properties

Here, we provide the pre-requisites for the two-party garbled circuit based computation of Yao [137]. All the garbled circuit computations in this thesis can be viewed as an instance

of a two-party case, and hence we omit the details for the multi-party case [12, 15]. As per Yao's garbling circuit paradigm [137], every wire in the circuit is assigned two κ-bit strings, called "keys", one each for bit value 0 and 1 on that wire. Let $(\mathsf{K}^0_{\mathsf{x}}, \mathsf{K}^1_{\mathsf{x}})$ denote the zero-key and one-key, respectively, on wire x in the circuit. For simplicity, the same notation is used for wire identity as well as the value on the wire. For instance, the key-pair for wire x is denoted as $(\mathsf{K}^0_{\mathsf{x}}, \mathsf{K}^1_{\mathsf{x}})$, while the key corresponding to bit x on the wire is denoted as $\mathsf{K}^{\mathsf{x}}_{\mathsf{x}}$. Then, each gate is constructed by encrypting the output-wire key with the appropriate input-wire keys. For example, for an AND gate with input wires x, y and output wire z, $\mathsf{K}^0_{\mathsf{z}}$ is double encrypted with keys $\mathsf{K}^0_{\mathsf{x}}, \mathsf{K}^0_{\mathsf{y}}$, with $\mathsf{K}^0_{\mathsf{x}}, \mathsf{K}^1_{\mathsf{y}}$, and with $\mathsf{K}^1_{\mathsf{x}}, \mathsf{K}^0_{\mathsf{y}}$, while $\mathsf{K}^1_{\mathsf{z}}$ is double encrypted with $\mathsf{K}^1_{\mathsf{x}}, \mathsf{K}^1_{\mathsf{y}}$. Give one key on each input wire, the output wire key can be obtained by decrypting the ciphertext which was encrypted using the corresponding input wire keys. These ciphertexts are provided in a permuted order so that the evaluating party does not learn which key, $\mathsf{K}^0_{\mathsf{z}}$ or $\mathsf{K}^1_{\mathsf{z}}$, it obtains after decryption.

A garbling scheme $\mathcal{G}$, consists of four algorithms $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ defined as follows:

1. $\mathsf{Gb}(1^\kappa, \mathsf{Ckt}) \to (\mathsf{GC}, e, d)$: Gb takes as input the security parameter κ and the circuit Ckt to be garbled, and outputs a garbled circuit GC, encoding information e and decoding information d.

2. $\mathsf{En}(x, e) \to \mathbf{X}$: En encodes input x using e to output encoded input $\mathbf{X}$. $\mathbf{X}$ is referred to as encoded input or encoded keys interchangeably.

3. $\mathsf{Ev}(\mathsf{GC}, \mathbf{X}) \to \mathbf{Y}$: Ev evaluates the garbled circuit GC on the encoded input $\mathbf{X}$ and produces the encoded output $\mathbf{Y}$.

4. $\mathsf{De}(\mathbf{Y}, d) \to y$: The encoded output $\mathbf{Y}$ is decoded into the clear output y by running the De algorithm on $\mathbf{Y}$ and d.

We rely on the following properties of garbling scheme [15] in our constructions.

1. A garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ is *correct* if for all input lengths $n \leq \mathsf{poly}(\kappa)$, circuits $C : \{0,1\}^n \to \{0,1\}^m$ and inputs $x \in \{0,1\}^n$, the following holds.

$$\mathsf{Pr}[\mathsf{De}(\mathsf{Ev}(\mathsf{GC}, \mathsf{En}(x, e)), d) \neq C(x) : (\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, C)] < \mathsf{negl}(\kappa)$$

2. A garbling scheme $\mathcal{G}$ is said to be *private* if for all $n \leq \mathsf{poly}(\kappa)$, circuit $C : \{0,1\}^n \to \{0,1\}^m$, there exists a PPT simulator $\mathcal{S}_{\mathbf{priv}}$ such that for all $x \in \{0,1\}^n$, for all PPT adversary $\mathcal{A}$ the following distributions are computationally indistinguishable.

- $\textsc{real}(C, x)$: run $(\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, C)$ and output $(\mathsf{GC}, \mathsf{En}(x, e), d)$.
- $\textsc{ideal}(C, C(x))$: run $(\mathsf{GC}', \mathbf{X}, d') \leftarrow \mathcal{S}_{\mathbf{priv}}(1^\kappa, C, C(x))$ and output $(\mathsf{GC}', \mathbf{X}, d')$.

3. A garbling scheme $\mathcal{G}$ is *authentic* if for all $n \leq \mathsf{poly}(\kappa)$, circuit $C : \{0,1\}^n \to \{0,1\}^m$, input $x \in \{0,1\}^n$ and for all PPT adversary $\mathcal{A}$, the following probability is $\mathsf{negl}(\kappa)$.

$$\mathsf{Pr}\left[\begin{array}{l}\hat{\mathbf{Y}} \neq \mathsf{Ev}(\mathsf{GC}, \mathbf{X}) \\ \wedge\, \mathsf{De}(\hat{\mathbf{Y}}, d) \neq \bot\end{array} : \begin{array}{c}\mathbf{X} = \mathsf{En}(x, e), (\mathsf{GC}, e, d) \leftarrow \mathsf{Gb}(\kappa, \mathsf{Ckt}), \\ \hat{\mathbf{Y}} \leftarrow \mathcal{A}(\mathsf{GC}, \mathbf{X})\end{array}\right]$$

Part I

Layer I: MPC Protocols

Introduction to Layer I

In this part, we provide the details of the Layer I blocks of our three-layer architecture (Fig. 1.1). Before going into the details of each of our frameworks, we provide an abstraction of the underlying secret sharing semantics. This is followed by an overview of the basic blocks of our MPC frameworks.

An Abstraction of Our Sharing Semantics

To enforce security, we perform computation on secret-shared data. For the arithmetic and boolean sharing, we follow replicated secret sharing (RSS), where a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is split into shares and is denoted by $[\![\cdot]\!]$. To leverage the benefits of the preprocessing paradigm, we associate meaning to the shares and demarcate the parties in terms of their roles. The parties are categorized into two sets – i) $\mathcal{P}_{\mathsf{on}}$ - online parties that perform the computation in the online phase, and ii) $\mathcal{P}_{\mathsf{ver}}$ - verifiers that help in generating preprocessing data and has almost no role in the online phase[1].

Framework	Parties[a]			$[\![\cdot]\!]$-shares of value v[b]			
	$\mathcal{P}$	$\mathcal{P}_{\mathsf{on}}$	$\mathcal{P}_{\mathsf{ver}}$	P_0	P_1	P_2	P_3
ASTRA	P_0, P_1, P_2	P_1, P_2	P_0	$\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^2$	–
SWIFT	P_1, P_2, P_3	P_1, P_2, P_3	–	–	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^3$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2$
Tetrad	P_0, P_1, P_2, P_3	P_1, P_2, P_3	P_0	$\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^3$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^2, \lambda_{\mathsf{v}}^3$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2$
ABY2.0	P_1, P_2	P_1, P_2	–	–	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1$	$\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^2$	–

[a] $\mathcal{P}_{\mathsf{on}}$ - Online parties, $\mathcal{P}_{\mathsf{ver}}$ - Verifiers, [b] $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$, $\lambda_{\mathsf{v}} = \lambda_{\mathsf{v}}^1 + \lambda_{\mathsf{v}}^2$ or $\lambda_{\mathsf{v}}^1 + \lambda_{\mathsf{v}}^2 + \lambda_{\mathsf{v}}^3$

Table 2.2: Sharing semantics ($[\![\cdot]\!]$) for value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ across various frameworks.

For every value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, we associate a mask denoted by λ_{v} and their sum is denoted by the masked value $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$. The share distribution is done in a specific manner to achieve

[1] Except operations like input sharing, output reconstruction, final stages of verification etc.

practical efficiency. The masked value m_v is given in clear to all the parties in $\mathcal{P}_{\mathsf{on}}$ and the mask λ_v is made available to them in a replicated fashion. For the case when there are p parties in $\mathcal{P}_{\mathsf{on}}$, the mask λ_v is split into p shares, denoted by $\lambda_\mathsf{v}^1, \ldots, \lambda_\mathsf{v}^p$, such that $\lambda_\mathsf{v} = \sum_{j=1}^{p} \lambda_\mathsf{v}^j$. Each party $P_j \in \mathcal{P}_{\mathsf{on}}$ gets all but one share of λ_v guaranteeing privacy.

On the other hand, parties in $\mathcal{P}_{\mathsf{ver}}$ obtain all the shares of the mask λ_v, enabling them to compute λ_v in clear. The parties in $\mathcal{P}_{\mathsf{ver}}$ are refrained from obtaining the mask m_v to ensure privacy. The sharing semantics for our frameworks are summarized in Table 2.2.

The idea of using a *masked* evaluation goes back to the work of Lindell et al. [94] in the context of multi-party garbling over boolean circuits. Here, a *masking bit* is assigned to every wire in the circuit to prevent the parties from knowing the actual value on the wire. Wang et al. [135] adopted this idea to achieve efficient authenticated two-party garbling schemes. Inspired from [135], Katz et al. [75] proposed an n-party semi-honest protocol in the dishonest majority setting using the idea of masked evaluation. Concretely, every party holds an n-out-of-n secret sharing of a random boolean mask along with the (public) masked value. The resultant protocol is then used to construct an efficient MPC-in-the-head style zero-knowledge protocol. In an orthogonal line of work, Ben-Efraim et al. [17] adopted this strategy and improved the online communication of SPDZ-style protocols (dishonest majority) by using function-dependent pre-processing.

The Complete MPC

In order to compute an arithmetic circuit ckt over $\mathbb{Z}_{2^\ell}$, parties first invoke the key-setup functionality $\mathcal{F}_{\text{KEY}}$ (§2.5.1) for the key distribution. The computation is divided mainly into three stages – i) Input sharing, ii) Evaluation, and iii) Output Reconstruction. Using the description of the ckt, parties prepare the necessary preprocessing data by invoking the preprocessing phase of the respective stages. Concretely, all the mask values (λ) for every wire in the ckt along with other input-independent data will be ready after the preprocessing.

During the online phase, $P_i \in \mathcal{P}$ shares its input v_i by executing the input sharing protocol Π_{Sh}. That is, using the mask λ_{v_i}, P_i computes the masked value $\mathsf{m}_{\mathsf{v}_i}$ and communicates it to the parties in $\mathcal{P}_{\mathsf{on}}$. This is followed by the circuit evaluation phase, where parties evaluate the gates in the circuit in the topological order, with addition gates (and multiplication-by-a-constant gates) being computed locally and multiplication gates being computed via the multiplication protocol Π_{Mult}. At every gate output wire z, the goal is to compute the masked value (m_z) using the shares of the input wires. Finally, parties execute the reconstruction protocol Π_{Rec} on the output wires to reconstruct the function output.

Other blocks in Layer I

Truncation Repeated multiplications in Fixed-Point Arithmetic (FPA) result in an overflow with the fractional part doubling up in size after each multiplication. This can result in the loss of significant bits of information eventually. The naive solution of choosing a large enough ring to avoid the overflow is impractical for ML algorithms where the number of sequential multiplications is large. To tackle this, truncation [102, 101, 110, 38, 32, 85] is used where the result of the multiplication is brought back to the FPA representation by chopping off the last x bits.

For a value $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$, SecureML [102] showed that the truncated value $\mathsf{v}/2^x$, denoted by v^t, can be computed as $\mathsf{v}_1^\mathsf{t} + \mathsf{v}_2^\mathsf{t}$. With high probability, a truncated value having at most one bit error in the least significant position is generated. It was shown in SecureML that accuracy drop for ML algorithms due to the one bit error is minimal. However, the method cannot be generalized to more than two parties. ABY3 [101] demonstrated the extension to 3-party setting with a generic design that uses a truncation pair of the form $(\mathsf{r}, \mathsf{r}^\mathsf{t})$. Here, r is a random value and r^t denotes its truncated version. Given this pair, z can be truncated by opening $\mathsf{z} - \mathsf{r}$ towards all, and computing z^t as $\mathsf{z}^\mathsf{t} = (\mathsf{z} - \mathsf{r})^\mathsf{t} + \mathsf{r}^\mathsf{t}$. Note that all operations are carried out on shares. The design of our multiplication protocol allows for truncation to be carried out this way without any additional overhead in communication.

Multi-input Multiplication Given the $[\![\cdot]\!]$-shares of values, $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} \in \mathbb{Z}_{2^\ell}$, we design 3-input and 4-input multiplication protocols in our frameworks. For the three-input case, the goal is to compute $\mathsf{z} = \mathsf{abc}$, without the need for performing two sequential multiplications (i.e. first $\mathsf{y} = \mathsf{ab}$ then yc). Similarly, $\mathsf{z} = \mathsf{abcd}$ for the four-input case. We remark that our multi-input multiplication, when coupled with the optimized parallel prefix adder circuit from [113], brings in a $2\times$ improvement in online rounds, as well as an improvement in online communication of secure comparison, as will be shown later in the thesis.

NOT operation in Boolean world Given the boolean shares of a bit $\mathsf{b} \in \{0, 1\}$, denoted by $[\![\mathsf{b}]\!]^\mathbf{B}$, parties can locally compute the boolean shares corresponding to its complement $\overline{\mathsf{b}}$. For this, parties locally set $\mathsf{m}_{\overline{\mathsf{b}}} = 1 \oplus \mathsf{m}_\mathsf{b}$ and the $\lambda_{\overline{\mathsf{b}}}$ shares are set to be the same as λ_b. It is easy to verify that $\overline{\mathsf{b}} = \mathsf{m}_{\overline{\mathsf{b}}} \oplus \lambda_{\overline{\mathsf{b}}} = (1 \oplus \mathsf{m}_\mathsf{b}) \oplus \lambda_\mathsf{b} = 1((\mathsf{m}_\mathsf{b} \oplus \lambda_\mathsf{b}) = 1 \oplus \mathsf{b}$. We use NOT to denote this operation.

Garbled World In our frameworks, we build GC-based protocol, tailor-made for PPML applications where only a small portion of the computation is done over the garbled world. We

propose 2 GC protocols – one requiring communication of 2 GC evaluations and one online round, and the other one requiring 1 GC and two rounds.

Garbled evaluation proceeds in three phases– i) Input phase, ii) Evaluation, and iii) Output phase. The input phase involves transferring the keys to the evaluators for every input to the GC. The evaluation consists of GC transfer followed by GC evaluation. Lastly, in the output phase, evaluators obtain the encoded output. Moreover, the state-of-the-art GC optimizations of free-XOR [82, 84], half gates [140, 67], and fixed AES-key [16] are deployed in our protocols.

Preliminary details about the garbling scheme and properties are described in §2.5.5. In the thesis, to simplify the presentation, we assume single bit values; for ℓ-bit values, each operation is performed ℓ times in parallel.

Chapter 3

ASTRA: 3PC Semi-honest Protocols

This chapter provides details for the Layer I blocks of our 3PC framework ASTRA. Some of the results in this chapter resulted in a publication at ACM CCSW'19 [37]. Comparison of ASTRA with passively secure 3PC PPML framework of ABY3 [101], in terms of the communication for multiplication, is presented in Table 3.1.

Work	#Active Parties	Security	Multiplication		Multiplication with Truncation[a]		Conversions[b]
			Comm_{pre}	Comm_{on}[c]	Comm_{pre}	Comm_{on}	
ABY3 [101]	3	Semi-honest	–	3ℓ	$14\ell - 6x - 6$	4ℓ	A-B-G
ASTRA	2	Semi-honest	ℓ	2ℓ	ℓ	2ℓ	A-B-G

[a] ℓ - size of ring in bits, x - number of bits for the fractional part in FPA semantics.
[b] A, B, G indicate support for arithmetic, boolean, and garbled worlds respectively.
[c] 'Comm' - communication, 'pre' - preprocessing, 'on' - online

Table 3.1: Comparison of semi-honest 3PC frameworks for PPML

3.1 Preliminaries and Definitions

We consider 3 parties denoted by $\mathcal{P} = \{P_0, P_1, P_2\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, semi-honest adversary that can corrupt at most one party.

3.1.1 Sharing Semantics

For the arithmetic and boolean sharing, we follow a $(3, 1)$ replicated secret sharing (RSS), where a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is split into three shares. Two of the shares $(\lambda^1_{\mathsf{v}}, \lambda^2_{\mathsf{v}})$ can be generated in the

preprocessing phase independent of the value to be shared, and their sum can be interpreted as a mask (λ_{v}). The third share, dependent on v, can be computed in the online phase and can be treated as the masked value $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$.

Sharing Type	P_0	P_1	P_2
$[\cdot]$-sharing^a	–	v^1	v^2
$[\![\cdot]\!]$-sharing^b	$(\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2)$	$(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^1)$	$(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v}}^2)$

$^a\mathsf{v} = \mathsf{v}^1 + \mathsf{v}^2$ $\quad$ $^b\lambda_{\mathsf{v}} = \lambda_{\mathsf{v}}^1 + \lambda_{\mathsf{v}}^2$, $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$

Table 3.2: Semantics for $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ in ASTRA.

Next, we distinguish the three parties into two sets; the *eval* set $\mathcal{E} = \{P_1, P_2\}$ which is assigned the task of carrying out the computation, and is active throughout the online phase. The *helper* set $\mathcal{D} = \{P_0\}$, is used to assist $\mathcal{E}$ in preparing the preprocessing material, and so it is only active in the preprocessing phase. Complying with the roles and RSS format, the distribution is done as follows: $P_0 : \{\lambda_{\mathsf{v}}^1, \lambda_{\mathsf{v}}^2\}, P_1 : \{\lambda_{\mathsf{v}}^1, \mathsf{m}_{\mathsf{v}}\}$, and $P_2 : \{\lambda_{\mathsf{v}}^2, \mathsf{m}_{\mathsf{v}}\}$.

The RSS sharing semantics is presented in Table 3.2, denoted by $[\![\cdot]\!]$, along with the semantics for $[\cdot]$-sharing. Both the sharings used are linear i.e. given sharings of $\mathsf{v}_1, \ldots, \mathsf{v}_m$ and public constants $c_1, \ldots, c_m$, sharing of $\sum_{i=1}^m c_i \mathsf{v}_i$ can be computed non-interactively for an integer m.

Notation 3.1 *(a) For the $[\![\cdot]\!]$-shares of n values $\mathsf{a}_1, \ldots, \mathsf{a}_n$, $\gamma_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^{n} \lambda_{\mathsf{a}_i}$ and $\mathsf{m}_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^{n} \mathsf{m}_{\mathsf{a}_i}$ (b) We use superscripts $\mathbf{B}$, and $\mathbf{G}$ to denote sharing semantics in boolean, and garbled world, respectively– $[\![\cdot]\!]^{\mathbf{B}}$, $[\![\cdot]\!]^{\mathbf{G}}$. We omit the superscript for arithmetic world.*

Sharing semantics for boolean sharing over $\mathbb{Z}_2$ is similar to arithmetic sharing except that addition is replaced with XOR. The semantics for garbled sharing are described in §3.3 with the relevant context.

3.2 Arithmetic / Boolean 3PC

This section covers the details of our 3PC semi-honest protocol ASTRA over an arithmetic ring $\mathbb{Z}_{2^\ell}$. The protocol primarily consists of the following primitives – i) Sharing §3.2.1, ii) Multiplication §3.2.2, and iii) Reconstruction §3.2.3.

3.2.1 Sharing

Protocol Π_{Sh} (Fig. 3.1) enables P_i to generate $[\![\cdot]\!]$-share of a value v. During the preprocessing phase, λ-shares are sampled non-interactively using the pre-shared keys (cf. §2.5.1) in a way that P_i will get the entire mask λ_{v}. During the online phase, P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2. For the special case when P_0 wants to perform a $[\![\cdot]\!]$-sharing of v in the preprocessing, the communication can be optimized further. For this, parties set $\mathsf{m}_{\mathsf{v}} = 0$. P_0, P_1 sample λ^1_{v} non-interactively. P_0 computes and sends $\lambda^2_{\mathsf{v}} = -(\mathsf{v} + \lambda^1_{\mathsf{v}})$ to P_2.

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

Input(s): $P_i : \mathsf{v}$, **Output:** $[\![\mathsf{v}]\!]$.

Preprocessing: Sample as follows: $P_i, P_0, P_1 : \lambda^1_{\mathsf{v}}$, $P_i, P_0, P_2 : \lambda^2_{\mathsf{v}}$.

Online: P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2.

Figure 3.1: $[\![\cdot]\!]$-sharing of a value v by party P_i in ASTRA.

Lemma 3.1 (Communication) *Protocol* Π_{Sh} *(Fig. 3.1) requires a communication of at most* 2ℓ *bits and* 1 *round in the online phase.*

Proof: The preprocessing of Π_{Sh} is non-interactive as the parties sample non interactively using key setup $\mathcal{F}_{\text{KEY}}$ (§2.5.1). In the online phase, P_i sends m_{v} to P_1, P_2 resulting in 1 round and communication of at most 2ℓ bits ($P_i = P_0$). □

3.2.1.1 Joint Sharing

Protocol Π_{JSh} enables parties P_i, P_j to generate $[\![\cdot]\!]$-share of a value v. In ASTRA, protocol Π_{JSh} is used to enable P_1, P_2 generate $[\![\mathsf{v}]\!]$ non-interactively. For this, parties set $\lambda^1_{\mathsf{v}} = \lambda^2_{\mathsf{v}} = 0$ and $\mathsf{m}_{\mathsf{v}} = \mathsf{v}$.

3.2.2 Multiplication

Given the shares of a, b, the goal of the multiplication protocol is to generate shares of $\mathsf{z} = \mathsf{ab}$. The protocol is designed such that parties P_1, P_2 obtain a masked version of the output z, say $\mathsf{z} - \mathsf{r}$ in the online phase, and P_0 obtain the mask r in the preprocessing phase. Parties then generate $[\![\cdot]\!]$-sharing of these values, and locally compute $[\![\mathsf{z}-\mathsf{r}]\!] + [\![\mathsf{r}]\!]$ to obtain the final output.

Online Note that,

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= \mathsf{ab} - \mathsf{r} = (\mathsf{m_a} - \lambda_\mathsf{a})(\mathsf{m_b} - \lambda_\mathsf{b}) - \mathsf{r} \\ &= \mathsf{m_{ab}} - \mathsf{m_a}\lambda_\mathsf{b} - \mathsf{m_b}\lambda_\mathsf{a} + \gamma_\mathsf{ab} - \mathsf{r} \quad \text{(cf. notation 3.1)} \end{aligned} \tag{3.1}$$

In Eq 3.1, P_1, P_2 can compute $\mathsf{m_{ab}}$ locally, and hence we are interested in computing $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_{ab}}$. Let $\mathsf{y} = \mathsf{y}_1 + \mathsf{y}_2$, where y_1 and y_2 can be computed respectively by P_1 and P_2.

$$\begin{aligned} P_1 : \mathsf{y}_1 &= -\lambda_\mathsf{a}^1 \mathsf{m_b} - \lambda_\mathsf{b}^1 \mathsf{m_a} + [\gamma_\mathsf{ab} - \mathsf{r}]_1 \\ P_2 : \mathsf{y}_2 &= -\lambda_\mathsf{a}^2 \mathsf{m_b} - \lambda_\mathsf{b}^2 \mathsf{m_a} + [\gamma_\mathsf{ab} - \mathsf{r}]_2 \end{aligned} \tag{3.2}$$

The preprocessing is set up such that P_1, P_2 receive an additive sharing ($[\cdot]$) of $\gamma_\mathsf{ab} - \mathsf{r}$. Parties P_1, P_2 mutually exchange the missing share to reconstruct y and subsequently $\mathsf{z} - \mathsf{r}$.

Protocol $\Pi_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^\mathsf{t}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{ab}$.

Preprocessing:

1. P_0, P_j sample $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma_\mathsf{ab} - \mathsf{r}$ for $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
2. Party P_0: Computes $\mathsf{r} = \gamma_\mathsf{ab} - \mathsf{u}^1 - \mathsf{u}^2$. If $\mathsf{isTr} = 1$, sets $\mathsf{q} = \mathsf{r}^\mathsf{t}$, else $\mathsf{q} = \mathsf{r}$.
   Executes $\Pi_{\mathsf{Sh}}(P_0, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_{ab}}$.

1. Compute: $P_1 : \mathsf{y}_1 = -\lambda_\mathsf{a}^1 \mathsf{m_b} - \lambda_\mathsf{b}^1 \mathsf{m_a} + \mathsf{u}^1$, $P_2 : \mathsf{y}_2 = -\lambda_\mathsf{a}^2 \mathsf{m_b} - \lambda_\mathsf{b}^2 \mathsf{m_a} + \mathsf{u}^2$
2. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{m_{ab}}$.
3. P_1, P_2: If $\mathsf{isTr} = 1$, set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^\mathsf{t}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.
4. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^\mathsf{t}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 3.2: Multiplication with / without truncation in ASTRA.

Preprocessing Parties P_1, P_2 should obtain $[\gamma_\mathsf{ab} - \mathsf{r}]$ while P_0 should obtain r. For this, P_0, P_i for $i \in \{1, 2\}$ non-interactively sample $[\gamma_\mathsf{ab} - \mathsf{r}]_i$. This enables P_0 to obtain r in clear as it can compute γ_ab locally.

Lemma 3.2 (Communication) *Protocol* Π_{Mult} *(Fig. 3.2) (in* ASTRA*) requires* ℓ *bits of communication in the preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

Proof: During preprocessing, sampling of $\mathsf{u}^1, \mathsf{u}^2$ are performed non-interactively using $\mathcal{F}_{\text{KEY}}$. A communication of ℓ bits is required for the sharing of q by P_0. During online, P_1, P_2 exchange $\mathsf{y}_1, \mathsf{y}_2$ values in parallel resulting in a communication of 2ℓ bits and 1 round. □

3.2.2.1 Truncation

To accommodate truncation, the multiplication protocol is modified as follows. P_1, P_2 locally truncate $(\mathsf{z} - \mathsf{r})$ and generate $[\![\cdot]\!]$-shares of it in the online phase. Similarly, P_0 truncates r in the preprocessing and generates its $[\![\cdot]\!]$-shares. Parties locally compute $[\![\mathsf{z}^t]\!] = [\![(\mathsf{z} - \mathsf{r})^t]\!] + [\![\mathsf{r}^t]\!]$.

3.2.2.2 Multiplication with constant

Multiplication by a constant in MPC is typically local. Given constant α and $[\![\mathsf{v}]\!]$, the $[\![\cdot]\!]$-shares of the product $\mathsf{y} = \alpha\mathsf{v}$ can be locally computed as per (3.3).

$$\mathsf{m}_\mathsf{y} = \alpha\mathsf{m}_\mathsf{u}, \quad \lambda^1_\mathsf{y} = \alpha\lambda^1_\mathsf{v}, \quad \lambda^2_\mathsf{y} = \alpha\lambda^2_\mathsf{v} \tag{3.3}$$

However, in FPA, we need to perform a truncation on the output. Let $\alpha\mathsf{v} = \beta^1 + \beta^2$ where $\beta^1 = \alpha.\mathsf{m}_\mathsf{v}$ and $\beta^2 = \alpha.(-\lambda^1_\mathsf{v} - \lambda^2_\mathsf{v})$. P_1, P_2 truncate β^1 and generate its arithmetic sharing using Π_{JSh}, while P_0 does the same with β^2.

3.2.3 Reconstruction

Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, \mathsf{v})$ (Fig. 3.3) enables parties in $\mathcal{P}$ to compute v, given its $[\![\cdot]\!]$-share. Note that each party misses one share to reconstruct the output, and the other two parties hold this share. One out of the two parties will send the missing share to the party that lacks it. Reconstruction towards a single party can be viewed as a special case.

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

Input(s): $[\![\mathsf{v}]\!]$, **Output:** v.

1. P_0 sends λ^1_v to P_2; P_0 sends λ^2_v to P_1; P_1 sends m_v to P_0.
2. Compute $\mathsf{v} = \mathsf{m}_\mathsf{v} - \lambda^1_\mathsf{v} - \lambda^2_\mathsf{v}$.

Figure 3.3: Reconstruction of value v among $\mathcal{P}$ in ASTRA.

Lemma 3.3 (Communication) *Protocol* Π_{Rec} *(Fig. 3.3) requires a communication of* 3ℓ *bits and* 1 *round in the online phase.*

3.2.4 Multi-input Multiplication

3-input multiplication To compute $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{abc}$, note that

$$\begin{aligned}\mathsf{z} - \mathsf{r} = \mathsf{abc} - \mathsf{r} &= (\mathsf{m_a} - \lambda_\mathsf{a})(\mathsf{m_b} - \lambda_\mathsf{b})(\mathsf{m_c} - \lambda_\mathsf{c}) - \mathsf{r} \\ &= \mathsf{m_{abc}} - \mathsf{m_{ac}}\lambda_\mathsf{b} - \mathsf{m_{bc}}\lambda_\mathsf{a} - \mathsf{m_{ab}}\lambda_\mathsf{c} + \mathsf{m_a}\gamma_\mathsf{bc} + \mathsf{m_b}\gamma_\mathsf{ac} + \mathsf{m_c}\gamma_\mathsf{ab} - \gamma_\mathsf{abc} - \mathsf{r} \quad \text{(cf. notation 3.1)}\end{aligned} \tag{3.4}$$

Similar to Π_{Mult}, for $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_{abc}}$, let $\mathsf{y} = \mathsf{y}_1 + \mathsf{y}_2$, where y_1 and y_2 can be computed respectively by P_1 and P_2.

$$\begin{aligned}P_1 : \mathsf{y}_1 &= -\lambda^1_\mathsf{a}\mathsf{m_{bc}} - \lambda^1_\mathsf{b}\mathsf{m_{ac}} - \lambda^1_\mathsf{c}\mathsf{m_{ab}} + [\gamma_\mathsf{ab}]_1\,\mathsf{m_c} + [\gamma_\mathsf{ac}]_1\,\mathsf{m_b} + [\gamma_\mathsf{bc}]_1\,\mathsf{m_a} - [\gamma_\mathsf{abc} + \mathsf{r}]_1 \\ P_2 : \mathsf{y}_2 &= -\lambda^2_\mathsf{a}\mathsf{m_{bc}} - \lambda^2_\mathsf{b}\mathsf{m_{ac}} - \lambda^2_\mathsf{c}\mathsf{m_{ab}} + [\gamma_\mathsf{ab}]_1\,\mathsf{m_c} + [\gamma_\mathsf{ac}]_2\,\mathsf{m_b} + [\gamma_\mathsf{bc}]_2\,\mathsf{m_a} - [\gamma_\mathsf{abc} + \mathsf{r}]_2\end{aligned} \tag{3.5}$$

To generate $[\mathsf{x}]$ for $\mathsf{x} \in \{\gamma_\mathsf{ab}, \gamma_\mathsf{bc}, \gamma_\mathsf{ac}\}$, P_0, P_1 non-interactively sample P_1's share. P_0 computes the share of P_2 and communicates to it. The generation of $[\gamma_\mathsf{abc} + \mathsf{r}]$ and the rest of the steps follow similar to that of 2-input multiplication protocol Π_{Mult} in §3.2.2. The formal protocol appears in Fig. 3.4.

Protocol $\Pi_{\mathsf{Mult3}}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!], [\![\mathsf{c}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^\mathsf{t}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{abc}$.

Preprocessing:

1. For each $\mathsf{x} \in \{\gamma_\mathsf{ab}, \gamma_\mathsf{bc}, \gamma_\mathsf{ac}\}$, P_0, P_1 sample $\mathsf{x}^1 \in_R \mathbb{Z}_{2^\ell}$. P_0 computes and sends $\mathsf{x}^2 = \mathsf{x} - \mathsf{x}^1$ to P_2.
2. P_0, P_j sample $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma_\mathsf{abc} + \mathsf{r}$ for $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
3. Party P_0: Computes $\mathsf{r} = \mathsf{u}^1 + \mathsf{u}^2 - \gamma_\mathsf{abc}$. If $\mathsf{isTr} = 1$, sets $\mathsf{q} = \mathsf{r}^\mathsf{t}$, else $\mathsf{q} = \mathsf{r}$.
   Executes $\Pi_{\mathsf{Sh}}(P_0, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_{ab}}$.

1. Locally compute:

$$P_1 : \mathsf{y}_1 = -\lambda_{\mathsf{a}}^1 \mathsf{m}_{\mathsf{bc}} - \lambda_{\mathsf{b}}^1 \mathsf{m}_{\mathsf{ac}} - \lambda_{\mathsf{c}}^1 \mathsf{m}_{\mathsf{ab}} + [\gamma_{\mathsf{ab}}]_1\, \mathsf{m}_{\mathsf{c}} + [\gamma_{\mathsf{ac}}]_1\, \mathsf{m}_{\mathsf{b}} + [\gamma_{\mathsf{bc}}]_1\, \mathsf{m}_{\mathsf{a}} - \mathsf{u}^1,$$
$$P_2 : \mathsf{y}_2 = -\lambda_{\mathsf{a}}^2 \mathsf{m}_{\mathsf{bc}} - \lambda_{\mathsf{b}}^2 \mathsf{m}_{\mathsf{ac}} - \lambda_{\mathsf{c}}^2 \mathsf{m}_{\mathsf{ab}} + [\gamma_{\mathsf{ab}}]_1\, \mathsf{m}_{\mathsf{c}} + [\gamma_{\mathsf{ac}}]_2\, \mathsf{m}_{\mathsf{b}} + [\gamma_{\mathsf{bc}}]_2\, \mathsf{m}_{\mathsf{a}} - \mathsf{u}^2$$

2. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{m}_{\mathsf{ab}}$.

3. P_1, P_2: If $\mathsf{isTr} = 1$, set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.

4. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 3.4: Three-input Multiplication with / without truncation in ASTRA.

Lemma 3.4 (Communication) *Protocol* Π_{Mult3} *(Fig. 3.4) (in ASTRA) requires* 4ℓ *bits of communication in the preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

Proof: During preprocessing, ℓ bits of communication from P_0 to P_2 is required to generate $[\cdot]$-shares of each of $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{bc}}$, and γ_{ac}. The sampling of $\mathsf{u}^1, \mathsf{u}^2$ are performed non-interactively using $\mathcal{F}_{\text{KEY}}$. Another ℓ bits are required for the sharing of q by P_0. During online, P_1, P_2 exchange $\mathsf{y}_1, \mathsf{y}_2$ values in parallel resulting in a communication of 2ℓ bits and 1 round. □

4-input multiplication For the case of 4-input multiplication with $\mathsf{z} = \mathsf{abcd}$, note that

$$\begin{aligned}\mathsf{z} - \mathsf{r} = \mathsf{abcd} - \mathsf{r} &= (\mathsf{m}_{\mathsf{a}} - \lambda_{\mathsf{a}})(\mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}})(\mathsf{m}_{\mathsf{c}} - \lambda_{\mathsf{c}})(\mathsf{m}_{\mathsf{d}} - \lambda_{\mathsf{d}}) - \mathsf{r} \\ &= \mathsf{m}_{\mathsf{abcd}} - \mathsf{m}_{\mathsf{abc}}\lambda_{\mathsf{d}} - \mathsf{m}_{\mathsf{abd}}\lambda_{\mathsf{c}} - \mathsf{m}_{\mathsf{acd}}\lambda_{\mathsf{b}} - \mathsf{m}_{\mathsf{bcd}}\lambda_{\mathsf{a}} + \mathsf{m}_{\mathsf{ab}}\gamma_{\mathsf{cd}} + \mathsf{m}_{\mathsf{ac}}\gamma_{\mathsf{bd}} + \mathsf{m}_{\mathsf{ad}}\gamma_{\mathsf{bc}} + \mathsf{m}_{\mathsf{bc}}\gamma_{\mathsf{ad}} \\ &\quad + \mathsf{m}_{\mathsf{bd}}\gamma_{\mathsf{ac}} + \mathsf{m}_{\mathsf{cd}}\gamma_{\mathsf{ab}} - \mathsf{m}_{\mathsf{a}}\gamma_{\mathsf{bcd}} - \mathsf{m}_{\mathsf{b}}\gamma_{\mathsf{acd}} - \mathsf{m}_{\mathsf{c}}\gamma_{\mathsf{abd}} - \mathsf{m}_{\mathsf{d}}\gamma_{\mathsf{abc}} + \gamma_{\mathsf{abcd}} - \mathsf{r} \quad \text{(cf. notation 3.1)}\end{aligned} \tag{3.6}$$

Here the parties need to generate $[\cdot]$-shares of $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{ac}}, \gamma_{\mathsf{ad}}, \gamma_{\mathsf{bc}}, \gamma_{\mathsf{bd}}, \gamma_{\mathsf{cd}}, \gamma_{\mathsf{abc}}, \gamma_{\mathsf{abd}}, \gamma_{\mathsf{acd}}, \gamma_{\mathsf{bcd}}$ and $\gamma_{\mathsf{abcd}} - \mathsf{r}$. This is computed similarly as in 3-input multiplication and the protocol is denoted as Π_{Mult4}.

Lemma 3.5 (Communication) *Protocol* Π_{Mult4} *(in ASTRA) requires* 11ℓ *bits of communication in the preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

Proof: During preprocessing, ℓ bits of communication from P_0 to P_2 is required to generate $[\cdot]$-shares of each of the ten values $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{ac}}, \gamma_{\mathsf{ad}}, \gamma_{\mathsf{bc}}, \gamma_{\mathsf{bd}}, \gamma_{\mathsf{cd}}, \gamma_{\mathsf{abc}}, \gamma_{\mathsf{abd}}, \gamma_{\mathsf{acd}}, \gamma_{\mathsf{bcd}}$. The sampling of

$\mathsf{u}^1, \mathsf{u}^2$ are performed non-interactively using $\mathcal{F}_{\text{KEY}}$. A communication of ℓ bits is required for the sharing of q by P_0. During online, P_1, P_2 exchange $\mathsf{y}_1, \mathsf{y}_2$ values in parallel resulting in a communication of 2ℓ bits and 1 round. □

N-input multiplication Consider an N-input multiplication gate with inputs $\mathsf{a}_1, \ldots, \mathsf{a}_N$ and output z. Then, we can write

$$\mathsf{z} - \mathsf{r} = \prod_{j=1}^{N} (\mathsf{m}_{\mathsf{a}_j} - \lambda_{\mathsf{a}_j}) - \mathsf{r} = \left(\sum_{I \subseteq \{1,\ldots,N\}} (-1)^{|I|} \prod_{j \in I} \lambda_{\mathsf{a}_j} \prod_{k \notin I} \mathsf{m}_{\mathsf{a}_j} \right) - \mathsf{r} \tag{3.7}$$

Here $I \subseteq \{1, \ldots, N\}$ denotes a subset of indices from 1 to N, while $|I|$ denotes the cardinality of the set.

We note that for an N-Input multiplication gate, we would require a total of $2^N - N - 1$ terms to be processed in the preprocessing, while the online phase still requires a communication of just 2 ring elements. Hence, to maintain a balance between the online communication and the overhead in the preprocessing, we consider $N = 3$ and $N = 4$ in our platform.

3.2.5 Supporting on-demand computations

For on-demand applications where the underlying function to be computed is not known in advance, the preprocessing model is not desirable. We observe that the ASTRA protocol can be modified by executing the preprocessing steps in the online phase itself, keeping the same overall communication cost and online rounds. The formal protocol appears in Fig. 3.5.

Protocol $\Pi^{\text{NoPre}}_{\text{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{ab}$.

Online:

1. P_0, P_j sample $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma_{\mathsf{ab}} - \mathsf{r}$ for $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
2. Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m}_{\mathsf{ab}}$. Compute: $P_1 : \mathsf{y}_1 = -\lambda^1_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} - \lambda^1_{\mathsf{b}}\mathsf{m}_{\mathsf{a}} + \mathsf{u}^1$, $P_2 : \mathsf{y}_2 = -\lambda^2_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} - \lambda^2_{\mathsf{b}}\mathsf{m}_{\mathsf{a}} + \mathsf{u}^2$.
3. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1.
4. Parties proceed as follows:

(a) P_0: $\mathsf{r} = \gamma_{\mathsf{ab}} - \mathsf{u}^1 - \mathsf{u}^2$; $\mathsf{q} = \mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else $\mathsf{q} = \mathsf{r}$. Executes $\Pi_{\mathsf{Sh}}(P_0, \mathsf{q})$.

(b) P_1, P_2: $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2) + \mathsf{m}_{\mathsf{ab}}$; $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$.

5. Locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 3.5: Multiplication for on-demand applications in ASTRA.

Lemma 3.6 (Communication) *Protocol* $\Pi^{\mathsf{NoPre}}_{\mathsf{Mult}}$ *(Fig. 3.2) (in ASTRA) requires* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: Steps 3 and 4 (a) of $\Pi^{\mathsf{NoPre}}_{\mathsf{Mult}}$ can be executed in parallel resulting in 1 round and 3ℓ bits of communication. □

3.3 Garbled World

We propose 2 GC protocols – $\mathsf{ASTRA}_{\mathsf{T}}$ requiring communication of 2 GCs and 1 online round, and $\mathsf{ASTRA}_{\mathsf{C}}$ requiring 1 GC and 2 rounds. The 2 GC variant has two parallel executions, each comprising of 2 garblers and 1 evaluator. P_1, P_2 act as evaluators in two independent executions and the parties in $\Phi_1 = \{P_0, P_2\}$, $\Phi_2 = \{P_0, P_1\}$ act as garblers, respectively. The 1 GC variant comprises of a single execution with Φ_1 acting as garblers and P_1 as the evaluator.

3.3.1 2 GC Variant

Input Phase Given that the function input x is already available as $[\![\mathsf{x}]\!]^{\mathbf{B}}$, the boolean values $\mathsf{m}_{\mathsf{x}}, \lambda_{\mathsf{x}}$ act as the *new* inputs for the garbled computation, and garbled sharing ($[\![\cdot]\!]^{\mathbf{G}}$) is generated for each of these values. The semantics of $[\![\cdot]\!]^{\mathbf{B}}$-sharing ensures that each of these shares $(\mathsf{m}_{\mathsf{x}}, \lambda_{\mathsf{x}})$ is available with at least one garbler in each garbling instance. Thus, the goal of our input phase is to create the compound sharing, $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m}_{\mathsf{x}}]\!]^{\mathbf{G}}, [\![\lambda_{\mathsf{x}}]\!]^{\mathbf{G}})$ for every input x to the function to be evaluated via the GC. We first discuss the semantics for $[\![\cdot]\!]^{\mathbf{G}}$-sharing followed by steps for generating $[\![\cdot]\!]^{\mathbf{C}}$-sharing.

Garbled sharing semantics A value $\mathsf{v} \in \mathbb{Z}_2$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared (garbled shared) amongst $\mathcal{P}$ if P_0 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_0 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$, P_1 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$ and P_2 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{\mathsf{v},2}_{\mathsf{v}})$. Here, $\mathsf{K}^{\mathsf{v},j}_{\mathsf{v}} = \mathsf{K}^{0,j}_{\mathsf{v}} \oplus \mathsf{v}\Delta^j$ for $j \in \{1, 2\}$, and Δ^j, which is known only to the garblers in Φ_j, denotes the global offset with its least significant bit set to 1 and is same for every wire in the circuit. A value $\mathsf{x} \in \mathbb{Z}_2$ is said to be $[\![\cdot]\!]^{\mathbf{C}}$-shared (compound shared) if each value from $(\mathsf{m}_{\mathsf{x}}, \lambda_{\mathsf{x}})$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared. We write $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m}_{\mathsf{x}}]\!]^{\mathbf{G}}, [\![\lambda_{\mathsf{x}}]\!]^{\mathbf{G}})$.

Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ and $[\![\mathsf{x}]\!]^{\mathbf{C}}$ Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(\mathcal{P}, \mathsf{v})$ (Fig. 3.6) enables generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ where two garblers in each garbling instance hold v, and proceeds as follows. Consider the first garbling instance with evaluator P_1. Garblers in Φ_1 generate $\{\mathsf{K}^{\mathsf{b},1}_{\mathsf{v}}\}_{\mathsf{b}\in\{0,1\}}$ which denotes the key for value b on wire v, following the free-XOR technique [82, 84]. $P_s \in \Phi_1$ sends $\mathsf{K}^{\mathsf{v},1}_{\mathsf{v}}$ to evaluator P_1 where $P_s \in \Phi_1$ denotes the garbler that knows v in clear. Similar steps carried out with respect to the second garbling instance, at the end of which, garblers in Φ_2 possess $\{\mathsf{K}^{\mathsf{b},2}_{\mathsf{v}}\}_{\mathsf{b}\in\{0,1\}}$ while the evaluator P_2 holds $\mathsf{K}^{\mathsf{v},2}_{\mathsf{v}}$. Following this, the shares $[\![\mathsf{v}]\!]^{\mathbf{G}}_s$ held by $P_s \in \mathcal{P}$ are defined as $[\![\mathsf{v}]\!]^{\mathbf{G}}_0 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{\mathsf{v},2}_{\mathsf{v}})$. To generate $[\![\mathsf{x}]\!]^{\mathbf{C}}$, $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$ is invoked for each of m_{x} and λ_{x}.

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

Input(s): v, **Output:** $[\![\mathsf{v}]\!]^{\mathbf{G}}$.
Let $P_s \in \Phi_j$ be the garbler that knows v in clear where $j \in \{1, 2\}$.

1. Garblers in Φ_j generate keys $\mathsf{K}^{0,j}_{\mathsf{v}}, \mathsf{K}^{1,j}_{\mathsf{v}}$ for wire v, using free-XOR technique.
2. $P_s \in \Phi_j$ sends $\mathsf{K}^{\mathsf{v},j}_{\mathsf{v}}$ to evaluator P_j for the j^{th} garbling instance.
3. P_0 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_0 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$, P_1 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$ and P_2 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{\mathsf{v},2}_{\mathsf{v}})$.

Figure 3.6: Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ in ASTRA.

Evaluation Let $f(\mathsf{x})$ be the function to be evaluated. At this point, the function input is $[\![\cdot]\!]^{\mathbf{C}}$-shared. This renders $[\![\cdot]\!]^{\mathbf{G}}$-sharing for the input of the GC that corresponds to the function $f'(\mathsf{m}_{\mathsf{x}}, \alpha_{\mathsf{x}})$ which first combines the given boolean-shares to compute the actual input and then applies f on it. Let GC_j denote the garbled circuit to be sent to $P_j \in \{P_1, P_2\}$ by garblers in Φ_j. Sending of GC_j is overlapped with the key transfer (during generation of $[\![\mathsf{x}]\!]^{\mathbf{C}}$), to save rounds, where garbler P_0 sends GC_j to P_j. On receiving the GC, evaluators evaluate their respective GCs and obtain the key corresponding to the output, say z. This generates $[\![\mathsf{z}]\!]^{\mathbf{G}}$.

Output phase The goal of output computation is to compute the output z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$. To reconstruct z towards $P_j \in \{P_1, P_2\}$, P_0 sends the least significant bit p^j of $\mathsf{K}^{0,j}_{\mathsf{z}}$, referred to as the decoding information, to P_j. P_j uses the received p^j to reconstruct z as $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$, where q^j denotes the least significant bit of $\mathsf{K}^{\mathsf{z},j}_{\mathsf{z}}$. To reconstruct z towards P_0, one evaluator, say P_1 sends the least significant bit, q^1, of $\mathsf{K}^{\mathsf{z},1}_{\mathsf{z}}$ to P_0. Reconstruction is lightweight and requires a single round for garblers while reconstruction towards evaluators can be overlapped with key transfer and does not incur extra rounds. The protocol appears in Fig. 3.7.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{Rec}}(\mathcal{P}, [\![\mathsf{z}]\!]^{\mathbf{G}})$

Input(s): $[\![\mathsf{z}]\!]^{\mathbf{G}}$, **Output:** z.

1. For an output wire z, let p^j denote the least significant bit of $\mathsf{K}^{0,j}_{\mathsf{z}}$ and q^j denote the least significant bit of $\mathsf{K}^{\mathsf{z},j}_{\mathsf{z}}$ for $j \in \{1, 2\}$.
2. *Reconstruction towards* $P_j \in \{P_1, P_2\}$: P_0 sends p^j to P_j who reconstructs $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$.
3. *Reconstruction towards* P_0: P_1 (or P_2) sends q^1 to P_0 who reconstructs $\mathsf{z} = \mathsf{p}^1 \oplus \mathsf{q}^1$.

Figure 3.7: Output computation: reconstruction of z in ASTRA.

Optimizations when deployed in mixed framework Working in the preprocessing model enables transfer of the (communication-intensive) GC and generating $[\![\cdot]\!]^{\mathbf{G}}$-shares of the input-independent shares of x (i.e. λ_{x}) in the preprocessing. Thus, the online phase is very light and only requires one round to generate $[\![\cdot]\!]^{\mathbf{G}}$-shares for the input-dependent data (i.e. m_{x}). Since evaluation is local, evaluators obtain $[\![\cdot]\!]^{\mathbf{G}}$-sharing of the GC output at the end of 1 round. Moreover, we require the garbled output to be reconstructed towards both P_1 and P_2 in clear. Thus, the steps for reconstruction towards P_0 can be avoided in $\Pi^{\mathbf{G}}_{\mathsf{Rec}}$ protocol (Fig. 3.7).

3.3.2 1 GC Variant

The garbling scheme here is similar to the 2GC variant except that now there exists only a single garbling instance. Parties in $\Phi_1 = \{P_0, P_2\}$ act as the garblers while P_1 act as the evaluator. Looking ahead, in the mixed protocol framework, the output has to be reconstructed towards P_1, P_2. Reconstruction towards P_1 does not incur additional rounds since sending of decoding information can be overlapped with the key transfer. However, unlike in the 2GC variant, an additional round is required for P_1 to send the output to P_2. This incurs one extra round as opposed to the 2GC variant.

3.4 Security proofs

The simulation for the semi-honest 3PC case is straightforward in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model, where $\mathcal{F}_{\mathsf{setup}}$ (§2.5.1) denotes the ideal functionality for the shared-key setup. 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 $\mathcal{A}$. Since $\mathcal{S}$ is given the input and output of the $\mathcal{A}$,

it can compute all the intermediate values of the circuit Ckt in clear.

For the input sharing of value v, $\mathcal{S}$ receives the $\mathsf{m_v}$ from $\mathcal{A}$ on behalf of the honest parties. Similarly, for the inputs of honest parties, $\mathcal{S}$ interacts with the $\mathcal{A}$ with the inputs set to 0. The simulated view is indistinguishable from the ideal view due to the privacy of the underlying sharing scheme. The linear gates involve no communication, while simulation of the multiplication protocol is straightforward. Moreover, simulation for the joint sharing (Π_{JSh}) instances is similar to that of the sharing protocol. The protocol's design is such that $\mathcal{S}$ will always know the value to be sent as part of the joint sharing protocol. Finally, for the reconstruction towards $\mathcal{A}$, $\mathcal{S}$ calculates the missing share of $\mathcal{A}$ using y and the other shares. The missing share is then communicated to $\mathcal{A}$ as per the reconstruction protocol.

Chapter 4

SWIFT: 3PC Fair and Robust Protocols

This chapter provides details for the Layer I blocks of our 3PC framework SWIFT. Some of the results in this chapter resulted in publications at NDSS'20 [110] and USENIX Security'21 [85]. Comparison of SWIFT with actively secure 3PC PPML framework of ABY3 [101], in terms of the communication for multiplication, is presented in Table 4.1.

Work	#Active Parties	Security	Multiplication		Multiplication with Truncation[a]		Conversions[b]
			Comm_{pre}	Comm_{on}[c]	Comm_{pre}	Comm_{on}	
ABY3 [101]	3	Abort	12ℓ	9ℓ	$100\ell - 44x - 84$	12ℓ	A-B-G
SWIFT	2	GOD	3ℓ	3ℓ	15ℓ	3ℓ	A-B-G

[a] ℓ - size of ring in bits, x - number of bits for the fractional part in FPA semantics.
[b] A, B, G indicate support for arithmetic, boolean, and garbled worlds respectively.
[c] 'Comm' - communication, 'pre' - preprocessing, 'on' - online

Table 4.1: Comparison of malicious 3PC frameworks for PPML

4.1 Preliminaries and Definitions

We consider 3 parties denoted by $\mathcal{P} = \{P_1, P_2, P_3\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, malicious adversary that can corrupt at most 1 party.

4.1.1 Sharing Semantics

For the arithmetic and boolean sharing, we follow a $(3, 1)$ RSS scheme similar to ASTRA, except that a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is split into four shares. Three of the shares $(\lambda_\mathsf{v}^1, \lambda_\mathsf{v}^2, \lambda_\mathsf{v}^3)$ can be generated in the preprocessing phase independent of the value to be shared, and their sum can be interpreted as a mask (λ_v). The fourth share, dependent on v, can be computed in the online phase and can be treated as the masked value $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_\mathsf{v}$.

Similar to ASTRA, we distinguish the three parties into two sets; the *eval* set $\mathcal{E} = \{P_1, P_2\}$ which is assigned the task of carrying out the computation, and is active throughout the online phase. The *helper* set $\mathcal{D} = \{P_3\}$, is used to assist $\mathcal{E}$ in verification, and so it is only active towards the end of the computation. Moreover, the share distribution is done as follows: $P_1 : \{\lambda_\mathsf{v}^1, \lambda_\mathsf{v}^3, \mathsf{m}_\mathsf{v}\}, P_2 : \{\lambda_\mathsf{v}^2, \lambda_\mathsf{v}^3, \mathsf{m}_\mathsf{v}\}$, and $P_3 : \{\lambda_\mathsf{v}^1, \lambda_\mathsf{v}^2, \mathsf{m}_\mathsf{v}\}$.

Sharing Type	P_1	P_2	P_3
$[\cdot]$-sharing	v^1	v^2	–
$\langle\cdot\rangle$-sharing[a]	$(\mathsf{v}^1, \mathsf{v}^3)$	$(\mathsf{v}^2, \mathsf{v}^3)$	$(\mathsf{v}^1, \mathsf{v}^2)$
$[\![\cdot]\!]$-sharing[b]	$(\lambda_\mathsf{v}^1, \lambda_\mathsf{v}^3, \mathsf{m}_\mathsf{v})$	$(\lambda_\mathsf{v}^2, \lambda_\mathsf{v}^3, \mathsf{m}_\mathsf{v})$	$(\lambda_\mathsf{v}^1, \lambda_\mathsf{v}^2, \mathsf{m}_\mathsf{v})$

[a] $\mathsf{v} = \mathsf{v}^1 + \mathsf{v}^2 + \mathsf{v}^3$ [b] $\lambda_\mathsf{v} = \lambda_\mathsf{v}^1 + \lambda_\mathsf{v}^2 + \lambda_\mathsf{v}^3$, $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_\mathsf{v}$

Table 4.2: Semantics for $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ in SWIFT.

The RSS sharing semantics is presented in Table 4.2, denoted by $[\![\cdot]\!]$, in a modular way with the help of two intermediate sharing semantics $[\cdot]$, and $\langle\cdot\rangle$. All the sharings used are linear i.e. given sharings of values $\mathsf{v}_1, \ldots, \mathsf{v}_m$ and public constants $c_1, \ldots, c_m$, sharing of $\sum_{i=1}^m c_i \mathsf{v}_i$ can be computed non-interactively for an integer m.

Notation 4.1 *(a) For the $[\![\cdot]\!]$-shares of n values $\mathsf{a}_1, \ldots, \mathsf{a}_n$, $\gamma_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^n \lambda_{\mathsf{a}_i}$ and $\mathsf{m}_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^n \mathsf{m}_{\mathsf{a}_i}$ (b) We use superscripts $\mathbf{B}$, and $\mathbf{G}$ to denote sharing semantics in boolean, and garbled world, respectively– $[\![\cdot]\!]^\mathbf{B}$, $[\![\cdot]\!]^\mathbf{G}$. We omit the superscript for arithmetic world.*

Sharing semantics for boolean sharing over $\mathbb{Z}_2$ is similar to arithmetic sharing except that addition is replaced with XOR. The semantics for garbled sharing are described in §4.3 with the relevant context.

4.1.1.1 $\mathcal{F}_{\mathsf{zero}}$ - Generating additive shares of zero

In SWIFT, we make use of a functionality $\mathcal{F}_{\mathsf{zero}}$ to enable P_i obtain Z_i for $i \in \{1, 2, 3\}$ such that $Z_1 + Z_2 + Z_3 = 0$. We observe that the functionality can be instantiated non-interactively

using the pre-shared keys (cf. §2.5.1). For this, parties in $\mathcal{P} \setminus \{P_j\}$ sample random value r_j for $j \in \{1,2,3\}$. The shares are then defined as $Z_1 = \mathsf{r}_3 - \mathsf{r}_2, Z_2 = \mathsf{r}_1 - \mathsf{r}_3$ and $Z_3 = \mathsf{r}_2 - \mathsf{r}_1$.

4.1.2 Joint-Send (jsnd) Primitive

The Joint-Send (jsnd) primitive, for the case of security with fairness, allows parties P_i, P_j to relay a message v to a third party P_k ensuring either the delivery of the message or `abort` in case of inconsistency. Towards this, P_i sends v to P_k, while P_j sends a hash of the same ($\mathsf{H}(\mathsf{v})$) to P_k. Party P_k accepts the message if the hash values are consistent and `abort` otherwise. Note that the communication of the hash can be clubbed together for several instances and be deferred to the end of the protocol, amortizing the cost.

Joint-Send (jsnd) for robust protocols The jsnd primitive, for the case of robustness, allows P_i, P_j to relay a common message, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, to recipient P_k, either by ensuring successful delivery of v, or by establishing a Trusted Third Party (TTP). The striking feature of jsnd is that it offers a rate-1 communication, i.e. for a message of ℓ elements, it only incurs a communication of ℓ elements (in an amortized sense). The task of jsnd is captured in an ideal functionality (Fig. 4.1) and the protocol for the same appears in Fig. 4.2. Next, we give an overview.

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

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

Step 1: $\mathcal{F}_{\mathsf{jsnd}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from P_s for $s \in \{i,j\}$, while it receives $(\mathsf{select}, \mathsf{ttp})$ from $\mathcal{S}$. ttp denotes the party that $\mathcal{S}$ wants to choose as the TTP and $P^\star \in \mathcal{P}$ denotes the corrupt party.

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 $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{ttp}$ and go to **Step 5**.

Step 4: TTP is 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 $s \in \{1,2,3\}$.

Figure 4.1: Ideal functionality for robust jsnd primitive in SWIFT

Given two parties P_i, P_j possessing a common value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, protocol Π_{jsnd} proceeds as follows. First, P_i sends v to P_k while P_j sends a hash of v to P_k. The communication of the hash is done once and for all from P_j to P_k. In the simplest case, P_k receives a consistent (value, hash) pair, and the protocol terminates. In all other cases, a TTP is identified as follows without having to communicate v again. Importantly, the following part can be run once and for all instances of Π_{jsnd} with P_i, P_j, P_k in the same roles, invoked in the final 3PC

protocol. Consequently, the cost relevant to this part vanishes in an amortized sense, making the construction rate-1.

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

Input(s): $P_i, P_j : \mathsf{v}$, $P_k : \perp$, **Output:** $P_i, P_j : \perp/\mathsf{TTP}$, $P_k : \mathsf{v}/\mathsf{TTP}$.
Each party P_s for $s \in \{i, j, k\}$ initializes bit $\mathsf{b}_s = 0$.

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

- P_k broadcasts "(accuse,$\mathtt{P_i}$)", 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,$\mathtt{P_k}$)" and $\mathsf{TTP} = P_j$. Analogously for P_j. If both P_i, P_j accuse P_k, then $\mathsf{TTP} = P_i$. Otherwise, P_s for $s \in \{i, j\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.
- P_i, P_j exchange their bits to each other. If P_i does not receive b_j from P_j, it broadcasts "(accuse,$\mathtt{P_j}$)" 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 4.2: Joint-Send for robust protocols in SWIFT

Each P_s for $s \in \{i, j, k\}$ maintains a bit b_s initialized to 0, as an indicator for inconsistency. When P_k receives an inconsistent (value, hash) pair, it sets $\mathsf{b}_k = 1$ and sends the bit to both P_i, P_j. Parties P_i, P_j cross-check with each other by exchanging the bit and turning on their inconsistency bit if the bit received from either P_k or its fellow sender is turned on. A party broadcasts a hash of its value when its inconsistency bit is on;[1] P_k's value is the one it receives from P_i. There are a bunch of possible cases at this stage, and a detailed analysis determines an eligible TTP in each case.

When P_k is silent, the protocol is understood to be complete. This is fine irrespective of

[1] hash can be computed on a combined message across many calls of jsnd.

the status of P_k– an honest P_k never skips this broadcast with inconsistency bit on, and a corrupt P_k implies honest senders. If either P_i or P_j is silent, then P_k is picked as TTP which is surely honest. A corrupt P_k could not make one of $\{P_i, P_j\}$ speak, as the senders (honest in this case) agree on their inconsistency bit (due to their mutual exchange of inconsistency bit). When all of them speak and (i) the senders' hashes do not match, P_k is picked as TTP; (ii) one of the senders conflicts with P_k, the other sender is picked as TTP; and lastly (iii) if there is no conflict, P_i is picked as TTP. The first two cases are self-explanatory. In the last case, either P_j or P_k is corrupt. If not, a corrupt P_i can have honest P_k speak (and hence turn on its inconsistency bit) by sending a v' whose hash is not the same as that of v and so inevitably, the hashes of honest P_j and P_k will conflict, contradicting (iii). As a final touch, we ensure that, in each step, a party raises a public alarm (via broadcast) accusing a silent party when it is not supposed to be. Then the protocol terminates immediately by labelling the party as TTP who is neither the complainer nor the accused.

Using jsnd *in protocols.* As mentioned earlier, the jsnd protocol needs to be viewed as consisting of two phases (*send, verify*), where *send* phase consists of P_i sending v to P_k and the rest goes to *verify* phase. Looking ahead, most of our protocols use jsnd, and consequently, our final construction, either of general MPC or any PPML task, will have several calls to jsnd. To leverage amortization, the *send* phase will be executed in all protocols invoking jsnd on the flow, while the *verify* for a fixed ordered pair of senders will be executed once and for all in the end. The *verify* phase will determine if all the sends were correct. If not, a TTP is identified, as explained, and the computation completes with the help of TTP, just as in the ideal world.

Lemma 4.1 (Communication) *Protocol* Π_{jsnd} *(Fig. 4.2) requires* 1 *round and an amortized communication of* ℓ *bits overall.*

Proof: Party P_i sends value v to P_k while P_j sends hash of the same to P_k. This accounts for one round and communication of ℓ bits. P_k then sends back its inconsistency bit to P_i, P_j, who then exchange it; this takes another two rounds. This is followed by parties broadcasting hashes on their values and selecting a TTP based on it, which takes one more round. All except the first round can be combined for several instances of Π_{jsnd} protocol and hence the cost gets amortized. □

Note that the appropriate instantiation of jsnd is used depending on the security guarantee. For simplicity, protocols where the fair and robust variants only differ in the instantiation of jsnd used, we give a common construction for both.

Notation 4.2 *Protocol* Π_{jsnd} *denotes the instantiation of Joint-Send (*jsnd*) primitive. We say that* P_i, P_j jsnd v *to* P_k *when they invoke* $\Pi_{\mathsf{jsnd}}(P_i, P_j, \mathsf{v}, P_k)$.

4.2 Arithmetic / Boolean 3PC

This section covers the details of our 3PC protocol SWIFT over an arithmetic ring $\mathbb{Z}_{2^\ell}$. We begin by explaining the sharing protocol in §4.2.1, multiplication with abort in §4.2.2, and the reconstruction in §4.2.3. Lastly, the details on elevating the security to fairness are presented in §4.2.3.1 and to robustness in §4.2.4.

4.2.1 Sharing

Protocol Π_{Sh} (Fig. 4.3) enables P_i to generate $[\![\cdot]\!]$-share of a value v. During the preprocessing phase, λ-shares are sampled non-interactively using the pre-shared keys (cf. §2.5.1) in a way that P_i will get the entire mask λ_{v}. During the online phase, P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1. Parties P_i, P_1 then communicates m_{v} to P_2 and P_3 using jsnd primitive.

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

Input(s): $P_i : \mathsf{v}$, **Output:** $[\![\mathsf{v}]\!]$.

Preprocessing: Sample as follows: $P_i, P_1, P_3 : \lambda_{\mathsf{v}}^1$, $P_i, P_2, P_3 : \lambda_{\mathsf{v}}^2$, $P_i, P_1, P_2 : \lambda_{\mathsf{v}}^3$.

Online:

1. P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_j. Here $P_j = P_1$ if $P_i \neq P_1$, else $P_j = P_2$.
2. P_i, P_j jsnd m_{v} to P_2 and P_3.

Figure 4.3: $[\![\cdot]\!]$-sharing of a value v by party P_i in SWIFT.

For the case when sharing happens in the preprocessing, the communication can be optimized to ℓ bits. For this, parties set $\mathsf{m}_{\mathsf{v}} = 0$ and the λ_{v}-shares are computed as follows:

- $P_i = P_1$: $\mathcal{P} \setminus \{P_2\} \leftarrow_R \lambda_{\mathsf{v}}^1$; $\mathcal{P} \leftarrow_R \lambda_{\mathsf{v}}^2$; P_1 sends $\lambda_{\mathsf{v}}^3 = -(\mathsf{v} + \lambda_{\mathsf{v}}^1 + \lambda_{\mathsf{v}}^2)$ to P_2.
- $P_i = P_2$: $\mathcal{P} \setminus \{P_1\} \leftarrow_R \lambda_{\mathsf{v}}^2$; $\mathcal{P} \leftarrow_R \lambda_{\mathsf{v}}^1$; P_2 sends $\lambda_{\mathsf{v}}^3 = -(\mathsf{v} + \lambda_{\mathsf{v}}^1 + \lambda_{\mathsf{v}}^2)$ to P_1.
- $P_i = P_3$: $\mathcal{P} \setminus \{P_1\} \leftarrow_R \lambda_{\mathsf{v}}^2$; $\mathcal{P} \leftarrow_R \lambda_{\mathsf{v}}^3$; P_3 sends $\lambda_{\mathsf{v}}^1 = -(\mathsf{v} + \lambda_{\mathsf{v}}^2 + \lambda_{\mathsf{v}}^3)$ to P_1.

Lemma 4.2 (Communication) *Protocol* Π_{Sh} *(Fig. 4.3) requires an amortized communication of at most* 2ℓ *bits and* 2 *rounds in the online phase.*

Proof: The preprocessing of Π_{Sh} is non-interactive as the parties sample non interactively using key setup $\mathcal{F}_{\text{SETUP}}$ (§2.5.1). In the online phase, P_i sends m_{v} to P_1 resulting in 1 round and communication of ℓ bits. The next round consists of one instance of Π_{jsnd} protocol and the cost follows from Lemma 4.1. □

4.2.1.1 Joint Sharing

Protocol Π_{JSh} enables parties P_i, P_j to generate $[\![\cdot]\!]$-share of a value v. The protocol is similar to Π_{Sh} except that P_j ensures the correctness of the sharing performed by P_i. During the preprocessing, λ-shares are sampled such that both P_i, P_j will get the entire mask λ_{v}. During the online phase, P_i, P_j compute and $\mathsf{jsnd}\ \mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ to parties P_1, P_2, P_3.

When the value v is available to both P_i, P_j in the preprocessing, protocol Π_{JSh} can be made non-interactive by setting the shares as given in Table 4.3.

(P_i, P_j)	λ_{v}^1	λ_{v}^2	λ_{v}^3	m_{v}
(P_1, P_2)	0	0	$-\mathsf{v}$	0
(P_1, P_3)	$-\mathsf{v}$	0	0	0
(P_2, P_3)	0	$-\mathsf{v}$	0	0

Table 4.3: Shares for Π_{JSh} in the preprocessing in SWIFT.

Lemma 4.3 (Communication) *Protocol Π_{JSh} is non-interactive in the preprocessing and requires an amortized communication of ℓ bits and 1 round in the online phase.*

Proof: The protocol involves one invocation of Π_{jsnd} protocol in the online and the cost follows from Lemma 4.1. □

4.2.2 Multiplication

Given the shares of a, b, the goal of the multiplication protocol is to generate shares of $\mathsf{z} = \mathsf{ab}$. The protocol is designed such that parties P_1, P_2 obtain a masked version of the output z, say $\mathsf{z} - \mathsf{r}$ in the online phase. Moreover, parties obtain the $[\![\cdot]\!]$-sharing of the mask r in the preprocessing. P_1, P_2 then generate $[\![\cdot]\!]$-sharing of $(\mathsf{z} - \mathsf{r})$ by executing Π_{JSh}. Parties locally compute the final output as $[\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$.

Online Similar to ASTRA, we have,

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= \mathsf{ab} - \mathsf{r} = (\mathsf{m}_{\mathsf{a}} - \lambda_{\mathsf{a}})(\mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}) - \mathsf{r} \\ &= \mathsf{m}_{\mathsf{ab}} - \mathsf{m}_{\mathsf{a}}\lambda_{\mathsf{b}} - \mathsf{m}_{\mathsf{b}}\lambda_{\mathsf{a}} + \gamma_{\mathsf{ab}} - \mathsf{r} \quad \text{(cf. notation 4.1)} \end{aligned} \tag{4.1}$$

In Eq 4.1, all the parties can compute m_{ab} locally, and hence we are interested in computing $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m}_{\mathsf{ab}}$. Let $\mathsf{y} = \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3$, where $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ can be computed respectively by the

pairs $(P_1, P_3), (P_2, P_3)$ and (P_1, P_2). Given a preprocessing that enables parties to obtain a $\langle\cdot\rangle$-sharing of $(\gamma_{\mathsf{ab}} - \mathsf{r})$, parties locally compute the additive shares of y according to (4.2).

$$\begin{aligned} P_1, P_3 : \mathsf{y}_1 &= -\lambda_{\mathsf{a}}^1 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^1 \mathsf{m}_{\mathsf{a}} + (\gamma_{\mathsf{ab}} - \mathsf{r})^1 \\ P_2, P_3 : \mathsf{y}_2 &= -\lambda_{\mathsf{a}}^2 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^2 \mathsf{m}_{\mathsf{a}} + (\gamma_{\mathsf{ab}} - \mathsf{r})^2 \\ P_1, P_2 : \mathsf{y}_3 &= -\lambda_{\mathsf{a}}^3 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^3 \mathsf{m}_{\mathsf{a}} + (\gamma_{\mathsf{ab}} - \mathsf{r})^3 \end{aligned} \tag{4.2}$$

Once the shares are computed, P_1, P_3 jsnd y_1 to P_2 and P_2, P_3 jsnd y_2 to P_1. Parties P_1, P_2 reconstruct y using the shares received and subsequently $\mathsf{z} - \mathsf{r}$.

Protocol $\Pi_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{ab}$.

Preprocessing:

1. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on $\langle\lambda_{\mathsf{a}}\rangle$ and $\langle\lambda_{\mathsf{b}}\rangle$ to obtain $\langle\gamma_{\mathsf{ab}}\rangle$.
2. If $\mathsf{isTr} = 0$:
   (a) Local computation of $\langle\mathsf{r}\rangle$: $\mathcal{P} \setminus \{P_2\} \leftarrow_R \mathsf{r}^1$; $\mathcal{P} \setminus \{P_1\} \leftarrow_R \mathsf{r}^2$; $\mathcal{P} \setminus \{P_3\} \leftarrow_R \mathsf{r}^3$.
   (b) Local computation of $[\![\mathsf{r}]\!]$: $\lambda_{\mathsf{r}}^1 = -\mathsf{r}^1$, $\lambda_{\mathsf{r}}^2 = -\mathsf{r}^2$, $\lambda_{\mathsf{r}}^3 = -\mathsf{r}^3$, $\mathsf{m}_{\mathsf{r}} = 0$. Set $[\![\mathsf{q}]\!] = [\![\mathsf{r}]\!]$.
3. If $\mathsf{isTr} = 1$, invoke Π_{trgen} (Fig. 8.4) to generate $(\langle\mathsf{r}\rangle, [\![\mathsf{r}^{\mathsf{t}}]\!])$. Set $[\![\mathsf{q}]\!] = [\![\mathsf{r}^{\mathsf{t}}]\!]$.
4. Locally compute $\langle(\gamma_{\mathsf{ab}} - \mathsf{r})\rangle = \langle\gamma_{\mathsf{ab}}\rangle - \langle\mathsf{r}\rangle$.

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

1. Parties locally compute the following:

$$\begin{aligned} P_1, P_3 : \mathsf{y}_1 &= -\lambda_{\mathsf{a}}^1 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^1 \mathsf{m}_{\mathsf{a}} + (\gamma_{\mathsf{ab}} - \mathsf{r})^1 \\ P_2, P_3 : \mathsf{y}_2 &= -\lambda_{\mathsf{a}}^2 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^2 \mathsf{m}_{\mathsf{a}} + (\gamma_{\mathsf{ab}} - \mathsf{r})^2 \\ P_1, P_2 : \mathsf{y}_3 &= -\lambda_{\mathsf{a}}^3 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^3 \mathsf{m}_{\mathsf{a}} + (\gamma_{\mathsf{ab}} - \mathsf{r})^3 \end{aligned}$$

2. P_1, P_3 jsnd y_1 to P_2, while P_2, P_3 jsnd y_2 to P_1. They locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{ab}}$.

3. P_1, P_2: If $\mathsf{isTr} = 1$, set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.

4. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 4.4: Multiplication with / without truncation in SWIFT.

Verification To leverage amortization, the *send* phase of jsnd alone is executed on the fly and *verify* is performed once for multiple instances of jsnd. Further, observe that P_1, P_2 possess the required shares in the online phase to compute the entire circuit. Hence, P_3 can come in only during *verify* of jsnd towards P_1, P_2, which can be deferred towards the end. Hence, the jsnd to P_3 (as part of Π_{JSh} by P_1, P_2 during the online) can be performed once, towards the end, thereby requiring a single round for multiple instances of Π_{JSh}. Following this, the *verify* of jsnd towards P_3 is performed first, followed by performing the *verify* of jsnd towards P_1, P_2 in parallel.

Preprocessing As mentioned above, parties should obtain a $\langle\cdot\rangle$-sharing of $(\gamma_{\mathsf{ab}} - \mathsf{r})$ from the preprocessing. The $\langle\cdot\rangle$-shares for a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ can be generated non-interactively using the key setup $\mathcal{F}_{\text{Setup}}$ (§2.5.1). To compute $\langle\gamma_{\mathsf{ab}}\rangle$, we rely on a 3-party multiplication protocol, say Π_{MultPre}, abstracted in a functionality $\mathcal{F}_{\mathsf{MultPre}}$ (Fig. 4.5). The security of Π_{MultPre} depends on the security required in our framework. For instance, instantiating $\mathcal{F}_{\mathsf{MultPre}}$ with the protocols of [24] and [2] will result in abort or fairness guarantees whereas using the robust 3 party protocol of [27] will result in a multiplication protocol with robustness. In SWIFT, we use the protocol of [27] in a black-box manner resulting in a communication of 3ℓ bits (amortized) for Π_{MultPre}. This leaves room for further improvements in the overall efficiency of our multiplication, which can be obtained by instantiating the black-box with efficient protocols.

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

$\mathcal{F}_{\mathsf{MultPre}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{MultPre}}$ receives $\langle\cdot\rangle$-shares of d, e from the parties. Let $P^\star$ denotes the party corrupted by $\mathcal{S}$. $\mathcal{F}_{\mathsf{MultPre}}$ receives $(\mathsf{f}_i, \mathsf{f}_j)$ from $\mathcal{S}$ as its share for $\langle\mathsf{f}\rangle$ where $\mathsf{f} = \mathsf{de}$. $\mathcal{F}_{\mathsf{MultPre}}$ proceeds as follows:

1. Reconstructs d, e using the shares received from honest parties and compute $\mathsf{f} = \mathsf{de}$.

2. Computes the third share $\mathsf{f}_k = \mathsf{f} - \mathsf{f}_i - \mathsf{f}_j$ and sets $\langle\mathsf{f}\rangle_1 = (\mathsf{f}_1, \mathsf{f}_3), \langle\mathsf{f}\rangle_2 = (\mathsf{f}_2, \mathsf{f}_3), \langle\mathsf{f}\rangle_3 = (\mathsf{f}_1, \mathsf{f}_2)$.

3. Send $(\mathsf{Output}, \langle\mathsf{f}\rangle_s)$ to $P_s \in \mathcal{P}$.

Figure 4.5: Ideal functionality for Π_{MultPre} in SWIFT.

Lemma 4.4 (Communication) *Protocol* Π_{Mult} *(Fig. 4.4) without truncation (in SWIFT) requires* 3ℓ *bits of communication in the preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: During preprocessing, sampling of the shares for $\langle \mathsf{r} \rangle$ is performed non-interactively using $\mathcal{F}_{\text{SETUP}}$. The Π_{MultPre} protocol, instantiated using the protocol of [27] requires a communication of 3ℓ bits in the preprocessing. During online, two instances of Π_{jsnd} are executed in parallel resulting in a communication of 2ℓ bits and 1 round. This is followed by a joint sharing by P_1, P_2 for which an additional communication of ℓ bits are required. However, in joint sharing, the communication is from P_1 to P_3 and the same can be deferred till the verification stage. Thus the online round is retained as 1 in an amortized sense. □

4.2.2.1 Truncation

To incorporate truncation, the multiplication protocol is modified such that P_1, P_2 execute joint sharing on the truncated value of $(\mathsf{z} - \mathsf{r})$ in the online phase. To complete the protocol, the $[\![\cdot]\!]$-shares of the truncated r, denoted by r^t, is needed. For this, we use Π_{trgen} (Fig. 8.4) protocol in the preprocessing that generates a pair of the form $(\langle \mathsf{r} \rangle, [\![\mathsf{r}^t]\!])$. More details on Π_{trgen} are provided in §8.1.5. Parties locally compute $[\![\mathsf{z}^t]\!] = [\![(\mathsf{z} - \mathsf{r})^t]\!] + [\![\mathsf{r}^t]\!]$.

4.2.2.2 Multiplication with constant

Multiplication by a constant in MPC is typically local. Given constant α and $[\![\mathsf{v}]\!]$, the $[\![\cdot]\!]$-shares of the product $\mathsf{y} = \alpha\mathsf{v}$ can be locally computed as per (4.3).

$$\mathsf{m}_{\mathsf{y}} = \alpha \mathsf{m}_{\mathsf{u}}, \quad \lambda^1_{\mathsf{y}} = \alpha\lambda^1_{\mathsf{v}}, \quad \lambda^2_{\mathsf{y}} = \alpha\lambda^2_{\mathsf{v}}, \quad \lambda^3_{\mathsf{y}} = \alpha\lambda^3_{\mathsf{v}} \tag{4.3}$$

In FPA, parties should obtain truncated y as both α and v are decimal values. For this, parties invoke Π_{trgen} (Fig. 8.4) in the preprocessing to generate $(\langle \mathsf{r} \rangle, [\![\mathsf{r}^t]\!])$ for a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. The $[\![\cdot]\!]$-shares of r are locally computed from $\langle \mathsf{r} \rangle$ locally similar to Π_{Mult} (Fig. 4.4). During online, parties locally compute $[\![\mathsf{v} - \mathsf{r}]\!]$ and reconstructs $\mathsf{z} = \mathsf{v} - \mathsf{r}$ using Π_{Rec} (Fig. 4.6). Parties locally compute $[\![\mathsf{z}^t]\!]$ by setting $\mathsf{m}_{\mathsf{z}^t} = \mathsf{z}^t$ and $\lambda^1_{\mathsf{z}^t} = \lambda^2_{\mathsf{z}^t} = \lambda^3_{\mathsf{z}^t} = 0$. Lastly, parties locally compute $[\![\mathsf{v}^t]\!] - [\![\mathsf{z}^t]\!] + [\![\mathsf{r}^t]\!]$.

4.2.3 Reconstruction

Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, \mathsf{v})$ (Fig. 4.6) enables parties to compute v, given its $[\![\cdot]\!]$-share and achieves security with abort. Note that each party misses one share to reconstruct the output, and the

other two parties hold this share. They will jsnd (abort variant) the missing share to the party that lacks it. Reconstruction towards a single party can be viewed as a special case.

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

Input(s): $[\![\mathsf{v}]\!]$, **Output:** v.

1. P_1, P_3 jsnd λ_{v}^1 to P_2; P_2, P_3 jsnd λ_{v}^2 to P_1; P_1, P_2 jsnd λ_{v}^3 to P_3.
2. Parties compute $\mathsf{v} = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}^1 - \lambda_{\mathsf{v}}^2 - \lambda_{\mathsf{v}}^3$.

Figure 4.6: Reconstruction (with abort security) of value v among $\mathcal{P}$ in SWIFT.

Lemma 4.5 (Communication) *Protocol* Π_{Rec} *(abort security, Fig. 4.6) requires an amortized communication of* 3ℓ *bits and* 1 *round.*

Proof: The protocol involves three invocations of Π_{jsnd} protocol and the cost follows from Lemma 4.1. □

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

Input(s): $[\![\mathsf{v}]\!]$, **Output:** v.

Preprocessing:

1. Parties locally compute the commitments on the λ_{v} shares as:

$$P_1, P_3 : \mathsf{Com}(\lambda_{\mathsf{v}}^1), \quad P_2, P_3 : \mathsf{Com}(\lambda_{\mathsf{v}}^2), \quad P_1, P_2 : \mathsf{Com}(\lambda_{\mathsf{v}}^3)$$

2. P_1, P_3 jsnd $\mathsf{Com}(\lambda_{\mathsf{v}}^1)$ to P_2; P_2, P_3 jsnd $\mathsf{Com}(\lambda_{\mathsf{v}}^2)$ to P_1; P_1, P_2 jsnd $\mathsf{Com}(\lambda_{\mathsf{v}}^3)$ to P_3

Online: Parties set their aliveness bit $\mathsf{b} = \texttt{continue}$, if the verification phase is successful. Else $\mathsf{b} = \texttt{abort}$.

1. Party $P_s \in \mathcal{P}$ broadcasts b_s and parties accept the value that forms the majority.
2. If the accepted value is $\texttt{abort}$, parties abort. Else P_1, P_3 open $\mathsf{Com}(\lambda_{\mathsf{v}}^1)$ towards P_2; P_2, P_3 open $\mathsf{Com}(\lambda_{\mathsf{v}}^2)$ towards P_1; P_1, P_2 open $\mathsf{Com}(\lambda_{\mathsf{v}}^3)$ towards P_3. Parties use the correct opening to obtain their missing share.
3. Parties compute $\mathsf{v} = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}^1 - \lambda_{\mathsf{v}}^2 - \lambda_{\mathsf{v}}^3$.

Figure 4.7: Fair Reconstruction of value v among $\mathcal{P}$ in SWIFT.

4.2.3.1 Achieving Fairness

Here, we show how to extend the security of SWIFT from abort to fairness by modifying the reconstruction protocol. During preprocessing, each pair of parties together prepare a commitment on the λ_v share missing at the third party. The commitments are then communicated via jsnd (abort variant), and the privacy is guaranteed by the hiding property of the underlying commitment scheme (cf. §2.5.3). Before proceeding with the output reconstruction in the online phase, we need to ensure that all the honest parties are alive after the verification phase. For this, all the parties maintain an *aliveness* bit, say b, which is initialized to `continue`. If the verification phase is not successful for a party, it sets $\mathsf{b} =$ `abort`. In the first round of reconstruction, the parties broadcast their b bit and accept the value that forms the majority. If $\mathsf{b} =$ `continue`, then a pair of parties open the commitment (communicated in the preprocessing) towards the third party. This method is fair because at least one honest party would have provided the correct opening to allow the third party to obtain its missing share. The formal protocol appears in Fig. 4.7.

4.2.4 Achieving Robustness

To elevate the security of SWIFT to robustness, we use the robust variant of jsnd in all the protocols. Moreover, for reconstruction, we use the fair reconstruction protocol in Fig. 4.7 except that the aliveness check (Online, Step 1) is no longer required. This is because the verification in robust jsnd guarantees identification of a TTP in case of any inconsistency, and the parties wouldn't have executed the reconstruction protocol.

4.2.5 Multi-input Multiplication

3-input multiplication To compute $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{abc}$, note that

$$\begin{aligned}\mathsf{z} - \mathsf{r} = \mathsf{abc} - \mathsf{r} &= (\mathsf{m_a} - \lambda_\mathsf{a})(\mathsf{m_b} - \lambda_\mathsf{b})(\mathsf{m_c} - \lambda_\mathsf{c}) - \mathsf{r} \\ &= \mathsf{m_{abc}} - \mathsf{m_{ac}}\lambda_\mathsf{b} - \mathsf{m_{bc}}\lambda_\mathsf{a} - \mathsf{m_{ab}}\lambda_\mathsf{c} + \mathsf{m_a}\gamma_\mathsf{bc} + \mathsf{m_b}\gamma_\mathsf{ac} + \mathsf{m_c}\gamma_\mathsf{ab} - \gamma_\mathsf{abc} - \mathsf{r} \quad \text{(cf. notation 4.1)}\end{aligned} \tag{4.4}$$

Similar to Π_{Mult}, for $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_{abc}}$, let $\mathsf{y} = \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3$.

$$\begin{aligned}P_1, P_3 : \mathsf{y}_1 &= -\lambda_\mathsf{a}^1\mathsf{m_{bc}} - \lambda_\mathsf{b}^1\mathsf{m_{ac}} - \lambda_\mathsf{c}^1\mathsf{m_{ab}} + \gamma_\mathsf{ab}^1\mathsf{m_c} + \gamma_\mathsf{ac}^1\mathsf{m_b} + \gamma_\mathsf{bc}^1\mathsf{m_a} - (\gamma_\mathsf{abc} + \mathsf{r})^1 \\ P_2, P_3 : \mathsf{y}_2 &= -\lambda_\mathsf{a}^2\mathsf{m_{bc}} - \lambda_\mathsf{b}^2\mathsf{m_{ac}} - \lambda_\mathsf{c}^2\mathsf{m_{ab}} + \gamma_\mathsf{ab}^2\mathsf{m_c} + \gamma_\mathsf{ac}^2\mathsf{m_b} + \gamma_\mathsf{bc}^2\mathsf{m_a} - (\gamma_\mathsf{abc} + \mathsf{r})^2 \\ P_1, P_2 : \mathsf{y}_3 &= -\lambda_\mathsf{a}^3\mathsf{m_{bc}} - \lambda_\mathsf{b}^3\mathsf{m_{ac}} - \lambda_\mathsf{c}^3\mathsf{m_{ab}} + \gamma_\mathsf{ab}^3\mathsf{m_c} + \gamma_\mathsf{ac}^3\mathsf{m_b} + \gamma_\mathsf{bc}^3\mathsf{m_a} - (\gamma_\mathsf{abc} + \mathsf{r})^3\end{aligned} \tag{4.5}$$

To generate $\langle \mathsf{x} \rangle$ for $\mathsf{x} \in \{\gamma_{\mathsf{ab}}, \gamma_{\mathsf{ac}}, \gamma_{\mathsf{bc}}\}$, parties rely on Π_{MultPre} protocol. Parties then use another instance of Π_{MultPre} on the inputs γ_{ab} and λ_{c} to generate $\langle \gamma_{\mathsf{abc}} \rangle$. The generation of $\langle \gamma_{\mathsf{abc}} + \mathsf{r} \rangle$ and the rest of the steps follow similar to that of 2-input multiplication protocol Π_{Mult} in §4.2.2. The formal protocol appears in Fig. 4.8.

Protocol $\Pi_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!], [\![\mathsf{c}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{abc}$.

Preprocessing:

1. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on the $\langle \cdot \rangle$-shares of $(\lambda_{\mathsf{a}}, \lambda_{\mathsf{b}})$, $(\lambda_{\mathsf{a}}, \lambda_{\mathsf{c}})$, and $(\lambda_{\mathsf{b}}, \lambda_{\mathsf{c}})$ to obtain $\langle \gamma_{\mathsf{ab}} \rangle, \langle \gamma_{\mathsf{ac}} \rangle, \langle \gamma_{\mathsf{bc}} \rangle$ respectively.
2. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on the $\langle \cdot \rangle$-shares of γ_{ab} and λ_{c} to obtain $\langle \gamma_{\mathsf{abc}} \rangle$.
3. If $\mathsf{isTr} = 0$:
   (a) Local computation of $\langle \mathsf{r} \rangle$: $\mathcal{P} \setminus \{P_2\} \leftarrow_R \mathsf{r}^1$; $\mathcal{P} \setminus \{P_1\} \leftarrow_R \mathsf{r}^2$; $\mathcal{P} \setminus \{P_3\} \leftarrow_R \mathsf{r}^3$.
   (b) Local computation of $[\![\mathsf{r}]\!]$: $\lambda_{\mathsf{r}}^1 = -\mathsf{r}^1$, $\lambda_{\mathsf{r}}^2 = -\mathsf{r}^2$, $\lambda_{\mathsf{r}}^3 = -\mathsf{r}^3$, $\mathsf{m}_{\mathsf{r}} = 0$. Set $[\![\mathsf{q}]\!] = [\![\mathsf{r}]\!]$.
4. If $\mathsf{isTr} = 1$, invoke Π_{trgen} (Fig. 8.4) to generate $(\langle \mathsf{r} \rangle, [\![\mathsf{r}^{\mathsf{t}}]\!])$. Set $[\![\mathsf{q}]\!] = [\![\mathsf{r}^{\mathsf{t}}]\!]$.
5. Locally compute $\langle (\gamma_{\mathsf{abc}} + \mathsf{r}) \rangle = \langle \gamma_{\mathsf{abc}} \rangle + \langle \mathsf{r} \rangle$.

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

1. Parties locally compute the following:

$$P_1, P_3 : \mathsf{y}_1 = -\lambda_{\mathsf{a}}^1 \mathsf{m}_{\mathsf{bc}} - \lambda_{\mathsf{b}}^1 \mathsf{m}_{\mathsf{ac}} - \lambda_{\mathsf{c}}^1 \mathsf{m}_{\mathsf{ab}} + \gamma_{\mathsf{ab}}^1 \mathsf{m}_{\mathsf{c}} + \gamma_{\mathsf{ac}}^1 \mathsf{m}_{\mathsf{b}} + \gamma_{\mathsf{bc}}^1 \mathsf{m}_{\mathsf{a}} - (\gamma_{\mathsf{abc}} + \mathsf{r})^1$$
$$P_2, P_3 : \mathsf{y}_2 = -\lambda_{\mathsf{a}}^2 \mathsf{m}_{\mathsf{bc}} - \lambda_{\mathsf{b}}^2 \mathsf{m}_{\mathsf{ac}} - \lambda_{\mathsf{c}}^2 \mathsf{m}_{\mathsf{ab}} + \gamma_{\mathsf{ab}}^2 \mathsf{m}_{\mathsf{c}} + \gamma_{\mathsf{ac}}^2 \mathsf{m}_{\mathsf{b}} + \gamma_{\mathsf{bc}}^2 \mathsf{m}_{\mathsf{a}} - (\gamma_{\mathsf{abc}} + \mathsf{r})^2$$
$$P_1, P_2 : \mathsf{y}_3 = -\lambda_{\mathsf{a}}^3 \mathsf{m}_{\mathsf{bc}} - \lambda_{\mathsf{b}}^3 \mathsf{m}_{\mathsf{ac}} - \lambda_{\mathsf{c}}^3 \mathsf{m}_{\mathsf{ab}} + \gamma_{\mathsf{ab}}^3 \mathsf{m}_{\mathsf{c}} + \gamma_{\mathsf{ac}}^3 \mathsf{m}_{\mathsf{b}} + \gamma_{\mathsf{bc}}^3 \mathsf{m}_{\mathsf{a}} - (\gamma_{\mathsf{abc}} + \mathsf{r})^3$$

2. P_1, P_3 jsnd y_1 to P_2, while P_2, P_3 jsnd y_2 to P_1. They locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{abc}}$.
3. P_1, P_2: If $\mathsf{isTr} = 1$, set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.
4. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 4.8: Three-input Multiplication with / without truncation in SWIFT.

Lemma 4.6 (Communication) *Protocol* Π_{Mult3} *(Fig. 4.8) (in SWIFT) requires* 12ℓ *bits of communication in the preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: During preprocessing, sampling of the shares for $\langle \mathsf{r} \rangle$ is performed non-interactively using $\mathcal{F}_{\text{SETUP}}$. Also, four instances of Π_{MultPre} protocol are executed in the preprocessing. Instantiating Π_{MultPre} using [27] requires a communication of 3ℓ bits for each of the instances. The online phase is similar to that of Π_{Mult} and the costs follow from Lemma 4.4. □

4-input multiplication For the case of 4-input multiplication with $\mathsf{z} = \mathsf{abcd}$, note that

$$\begin{aligned}\mathsf{z} - \mathsf{r} = \mathsf{abcd} - \mathsf{r} &= (\mathsf{m_a} - \lambda_\mathsf{a})(\mathsf{m_b} - \lambda_\mathsf{b})(\mathsf{m_c} - \lambda_\mathsf{c})(\mathsf{m_d} - \lambda_\mathsf{d}) - \mathsf{r} \\ &= \mathsf{m_{abcd}} - \mathsf{m_{abc}}\lambda_\mathsf{d} - \mathsf{m_{abd}}\lambda_\mathsf{c} - \mathsf{m_{acd}}\lambda_\mathsf{b} - \mathsf{m_{bcd}}\lambda_\mathsf{a} + \mathsf{m_{ab}}\gamma_\mathsf{cd} + \mathsf{m_{ac}}\gamma_\mathsf{bd} + \mathsf{m_{ad}}\gamma_\mathsf{bc} + \mathsf{m_{bc}}\gamma_\mathsf{ad} \\ &\quad + \mathsf{m_{bd}}\gamma_\mathsf{ac} + \mathsf{m_{cd}}\gamma_\mathsf{ab} - \mathsf{m_a}\gamma_\mathsf{bcd} - \mathsf{m_b}\gamma_\mathsf{acd} - \mathsf{m_c}\gamma_\mathsf{abd} - \mathsf{m_d}\gamma_\mathsf{abc} + \gamma_\mathsf{abcd} - \mathsf{r} \quad \text{(cf. notation 4.1)}\end{aligned} \tag{4.6}$$

Here the parties first generate $\langle \cdot \rangle$-shares of $\gamma_\mathsf{ab}, \gamma_\mathsf{ac}, \gamma_\mathsf{ad}, \gamma_\mathsf{bc}, \gamma_\mathsf{bd}$, and γ_cd using Π_{MultPre} on the respective inputs. In the next round, parties make use of these shares and Π_{MultPre} to generate $\gamma_\mathsf{abc}, \gamma_\mathsf{abd}, \gamma_\mathsf{acd}, \gamma_\mathsf{bcd}$ and γ_abcd. Thus, the preprocessing involves a total of eleven instances of Π_{MultPre} protocol.

Lemma 4.7 (Communication) *Protocol* Π_{Mult4} *(in SWIFT) requires* 33ℓ *bits of communication in the preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: During preprocessing, 11 instances of Π_{MultPre} protocol are executed. This results in communication of 33ℓ bits when the Π_{MultPre} protocol is instantiated with [27]. The online phase is similar to that of Π_{Mult}, and the costs follow from Lemma 4.4. □

4.3 Garbled World

Similar to ASTRA, we have two variants – $\mathsf{SWIFT_T}$ requiring communication of 2 GCs and one online round, and $\mathsf{SWIFT_C}$ requiring 1 GC and two rounds. The 2 GC variant has two parallel executions, each comprising of 2 garblers and 1 evaluator. P_1, P_2 act as evaluators in two independent executions and the parties in $\Phi_1 = \{P_2, P_3\}$, $\Phi_2 = \{P_1, P_3\}$ act as garblers, respectively. The 1 GC variant comprises of a single execution with Φ_1 acting as garblers and P_1 as the evaluator.

4.3.1 2 GC Variant

Input Phase Here, the boolean values $(\mathsf{m}_\mathsf{x}, \lambda^1_\mathsf{x}, \lambda^2_\mathsf{x}, \lambda^3_\mathsf{x})$ act as the *new* inputs for the garbled computation. The semantics of $[\![\cdot]\!]^{\mathbf{B}}$-sharing ensures that each of these shares is available with at least two parties (including evalator) in each garbling instance. Thus, the goal of our input phase is to create the compound sharing, $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m}_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^1_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^2_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^2_\mathsf{x}]\!]^{\mathbf{G}})$ for every input x to the function to be evaluated via the GC. Consider the garbling instance with P_1 as the evaluator. The number of input keys for this instance can be further reduced by treating $(\mathsf{m}_\mathsf{x} \oplus \lambda^2_\mathsf{x})$ as a single input. For the case of arithmetic values, the input changes to $(\mathsf{m}_\mathsf{x} - \lambda^2_\mathsf{x})$. Similarly, the other instance with P_2 as evaluator uses $(\mathsf{m}_\mathsf{x} \oplus \lambda^1_\mathsf{x})$ as input. We first discuss the semantics for $[\![\cdot]\!]^{\mathbf{G}}$-sharing followed by steps for generating $[\![\cdot]\!]^{\mathbf{C}}$-sharing.

Garbled sharing semantics A value $\mathsf{v} \in \mathbb{Z}_2$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared (garbled shared) amongst $\mathcal{P}$ if P_3 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_i = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, P_1 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$ and P_2 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$. Here, $\mathsf{K}^{\mathsf{v},j}_\mathsf{v} = \mathsf{K}^{0,j}_\mathsf{v} \oplus \mathsf{v}\Delta^j$ for $j \in \{1, 2\}$, and Δ^j, which is known only to the garblers in Φ_j, denotes the global offset with its least significant bit set to 1 and is same for every wire in the circuit. A value $\mathsf{x} \in \mathbb{Z}_2$ is said to be $[\![\cdot]\!]^{\mathbf{C}}$-shared (compound shared) if each value from $(\mathsf{m}_\mathsf{x}, \lambda^1_\mathsf{x}, \lambda^2_\mathsf{x}, \lambda^3_\mathsf{x})$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared. We write $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m}_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^1_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^2_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^2_\mathsf{x}]\!]^{\mathbf{G}})$.

Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ and $[\![\mathsf{x}]\!]^{\mathbf{C}}$ Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(\mathcal{P}, \mathsf{v})$ (Fig. 4.9) enables generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ where two garblers in each garbling instance hold v, and proceeds as follows. Consider the first garbling instance with evaluator P_1 and garblers P_s, P_t. Garblers in Φ_1 generate $\{\mathsf{K}^{\mathsf{b},1}_\mathsf{v}\}_{\mathsf{b}\in\{0,1\}}$ which denotes the key for value b on wire v, following the free-XOR technique [82, 84]. If the value v is known to both P_s, P_t, they jsnd the respective key to P_1. Else, w.l.o.g. let $P_s \in \Phi_j$ be the garbler that knows v. To ensure the correct key delivery towards P_1, we make garblers P_s, P_t commit to both the keys to P_1 via jsnd. P_s then sends the opening for commitment of $\mathsf{K}^{\mathsf{v},j}_\mathsf{v}$ to P_1. If the decommitment fails, P_1 abort for the case of secuirty with abort or fairness. For robustness it accuses P_s and P_t is chosen as the TTP.

Similar steps carried out with respect to the second garbling instance, at the end of which, garblers in Φ_2 possess $\{\mathsf{K}^{\mathsf{b},2}_\mathsf{v}\}_{\mathsf{b}\in\{0,1\}}$ while the evaluator P_2 holds $\mathsf{K}^{\mathsf{v},2}_\mathsf{v}$. Following this, the shares $[\![\mathsf{v}]\!]^{\mathbf{G}}_s$ held by $P_s \in \mathcal{P}$ are defined as $[\![\mathsf{v}]\!]^{\mathbf{G}}_0 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$. To generate $[\![\mathsf{x}]\!]^{\mathbf{C}}$, $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$ is invoked for each of $\mathsf{m}_\mathsf{x}, \lambda^1_\mathsf{x}, \lambda^2_\mathsf{x}$, and λ^3_x .

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

Input(s): v, **Output:** $[\![\mathsf{v}]\!]^{\mathbf{G}}$.

Let $P_s \in \Phi_j$ be the garbler that knows v in clear where $j \in \{1, 2\}$ and P_t be the co-garbler in Φ_j.

1. Garblers in Φ_j generate keys $\mathsf{K}^{0,j}_{\mathsf{v}}, \mathsf{K}^{1,j}_{\mathsf{v}}$ for wire v, using free-XOR technique.
2. If both P_s and P_t know v in clear, P_s, P_t jsnd $\mathsf{K}^{\mathsf{v},j}_{\mathsf{v}}$ to evaluator P_j.
3. Else, parties proceed as follows:
   (a) P_s, P_t prepare commitment on both the keys as $\mathsf{Com}(\mathsf{K}^{0,j}_{\mathsf{v}}), \mathsf{Com}(\mathsf{K}^{1,j}_{\mathsf{v}})$ and communicates to evaluator P_j in a random permuted order using jsnd.
   (b) P_s sends the opening for commitment of $\mathsf{K}^{\mathsf{v},j}_{\mathsf{v}}$ to P_j.
   (c) If the decommitment using the opening fails, P_j `abort` for the case of security with abort or fairness. For robustness, P_j broadcasts "`(accuse,`P_s`)`" and P_t is chosen as the TTP.
4. P_0 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_0 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$, P_1 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_{\mathsf{v}}, \mathsf{K}^{0,2}_{\mathsf{v}})$ and P_2 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_{\mathsf{v}}, \mathsf{K}^{\mathsf{v},2}_{\mathsf{v}})$.

Figure 4.9: Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ in SWIFT.

Evaluation Let $f(\mathsf{x})$ be the function to be evaluated. At this point, the function input is $[\![\cdot]\!]^{\mathbf{C}}$-shared. This renders $[\![\cdot]\!]^{\mathbf{G}}$-sharing for the input of the GC that corresponds to the function $f'\big(\mathsf{m}_{\mathsf{x}}, \lambda^1_{\mathsf{x}}, \lambda^2_{\mathsf{x}}, \lambda^3_{\mathsf{x}}\big)$ which first combines the given boolean-shares to compute the actual input and then applies f on it. Let GC_j denote the garbled circuit to be sent to $P_j \in \{P_1, P_2\}$ by garblers in Φ_j. Sending of GC_j is overlapped with the key transfer (during generation of $[\![\mathsf{x}]\!]^{\mathbf{C}}$), to save rounds, where garblers jsnd GC_j to P_j. On receiving the GC, evaluators evaluate their respective GCs and obtain the key corresponding to the output, say z. This generates $[\![\mathsf{z}]\!]^{\mathbf{G}}$.

Output phase The goal of output computation is to compute the output z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$. To reconstruct z towards $P_j \in \{P_1, P_2\}$, garblers in Φ_j jsnd the least significant bit p^j of $\mathsf{K}^{0,j}_{\mathsf{z}}$, referred to as the decoding information, to P_j. P_j uses the received p^j to reconstruct z as $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$, where q^j denotes the least significant bit of $\mathsf{K}^{\mathsf{z},j}_{\mathsf{z}}$. P_1, P_2 then jsnd z to P_3 completing the protocol. Reconstruction is lightweight and requires a single round towards P_3 while reconstruction towards P_1, P_2 can be overlapped with key transfer and does not incur extra rounds. The protocol appears in Fig. 4.10.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{Rec}}(\mathcal{P}, [\![\mathsf{z}]\!]^{\mathbf{G}})$

Input(s): $[\![\mathsf{z}]\!]^{\mathbf{G}}$, **Output:** z.

1. For an output wire z, let p^j denote the least significant bit of $\mathsf{K}^{0,j}_{\mathsf{z}}$ and q^j denote the least significant bit of $\mathsf{K}^{\mathsf{z},j}_{\mathsf{z}}$ for $j \in \{1, 2\}$.
2. *Reconstruction towards* P_j: Parties in Φ_j jsnd p^j to P_j who reconstructs $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$.
3. *Reconstruction towards* P_3: P_1, P_2 jsnd z to P_3.

Figure 4.10: Output computation: reconstruction of z in SWIFT.

Optimizations when deployed in mixed framework Working in the preprocessing model enables transfer of the (communication-intensive) GC and generating $[\![\cdot]\!]^{\mathbf{G}}$-shares of the input-independent shares of x (i.e. λ_{x}) in the preprocessing. Thus, the online phase is very light and only requires one round to generate $[\![\cdot]\!]^{\mathbf{G}}$-shares for the input-dependent data (i.e. m_{x}). Since evaluation is local, evaluators obtain $[\![\cdot]\!]^{\mathbf{G}}$-sharing of the GC output at the end of 1 round. Moreover, we require the garbled output to be reconstructed towards both P_1 and P_2 in clear. Thus, the steps for reconstruction towards P_3 can be avoided in $\Pi^{\mathbf{G}}_{\mathsf{Rec}}$ protocol (Fig. 4.10).

4.3.2 1 GC Variant

The garbling scheme here is similar to the 2GC variant except that now there exists only a single garbling instance. Parties in $\Phi_1 = \{P_2, P_3\}$ act as the garblers while P_1 act as the evaluator. Looking ahead, in the mixed protocol framework, the output has to be reconstructed towards P_1, P_2. Reconstruction towards P_1 does not incur additional rounds since sending of decoding information can be overlapped with the key transfer. To reconstruct towards P_2, P_1 sends the least significant bit of $\mathsf{K}^{\mathsf{z},1}_{\mathsf{z}}$, denoted by q^1, along with a hash of $\mathsf{K}^{\mathsf{z},j}_{\mathsf{z}}$ to P_2. Party P_2 accepts q^1 if the hash is consistent with the respective key. This is fine since a corrupt P_1 cannot send an incorrect key due to the *authenticity* of the garbling scheme [15]. On the other hand, if the hash is inconsistent, P_2 aborts for the case of security with abort or fairness. For robustness it accuses P_1 and P_3 is chosen as the TTP.

4.4 Security proofs

Without loss of generality, we prove the security of our robust framework. The case for fairness follows similarly, and we omit its details. We provide proofs in the $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{MultPre}}, \mathcal{F}_{\mathsf{jsnd}}\}$-hybrid

model, where $\mathcal{F}_{\mathsf{setup}}$ (§2.5.1), $\mathcal{F}_{\mathsf{MultPre}}$ (Fig. 4.5) and $\mathcal{F}_{\mathsf{jsnd}}$ (Fig. 5.18) denote the ideal functionality for the shared-key setup, preprocessing of multiplication (Π_{MultPre}) and jsnd, respectively.

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 $\mathcal{S}$ computes the input of $\mathcal{A}$, using the known keys, and sets the honest parties' inputs to be used in the simulation to 0. $\mathcal{S}$ invokes the ideal functionality $\mathcal{F}_{\text{GOD}}$ on behalf of $\mathcal{A}$ using the extracted input and obtains the output y. $\mathcal{S}$ now knows the inputs of $\mathcal{A}$ and can compute all the intermediate values for each building block. $\mathcal{S}$ proceeds with simulating each of the building blocks in the topological order. We provide the simulation for the case for corrupt P_1 and P_3. The case for corrupt P_2 is similar to that of P_1.

For modularity, we provide the simulation steps for each building block separately. Carrying out these blocks in the topological order yields the simulation for the entire computation. If a TTP is identified during the simulation, the simulator stops and returns the function output to the adversary on behalf of the TTP as per $\mathcal{F}_{\mathsf{jsnd}}$.

Ideal jsnd Functionality The ideal jsnd functionality for fairness security appears in Fig. 4.11 and that for the robust setting appears in Fig. 4.1.

Functionality $\mathcal{F}_{\mathsf{jsnd}}$ (for fair security)

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

Step 1: $\mathcal{F}_{\mathsf{jsnd}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from senders P_s for $s \in \{i, j\}$, $(\mathsf{Input}, \perp)$ from receiver P_k. While sending the inputs, the adversary is also allowed to send a special `abort` command.

Step 2: Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_l = \perp$.

Step 3: If $\mathsf{v}_i = \mathsf{v}_j$, set $\mathsf{msg}_k = \mathsf{v}_i$. Else, set $\mathsf{msg}_k =$ `abort`.

Step 4: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{1, 2, 3\}$.

Figure 4.11: Ideal functionality for jsnd in SWIFT

Sharing Protocol (Π_{Sh}, Fig. 4.3) During the preprocessing, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the respective keys to $\mathcal{A}$. The values commonly held with $\mathcal{A}$ are sampled using the respective keys, while others are sampled randomly. The details for the online phase are provided next. We omit the simulation for corrupt P_3 as it is similar to that of P_1.

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$

Online:

– If dealer is $\mathcal{A}$, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ receives $\mathsf{m_v}$ from $\mathcal{A}$ on behalf of P_2. $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ computes $\mathcal{A}$'s input v as $\mathsf{v} = \mathsf{m_v} - [\lambda_\mathsf{v}]_1 - [\lambda_\mathsf{v}]_2 - [\lambda_\mathsf{v}]_3$. It invokes $\mathcal{F}_{\mathrm{GOD}}$ on input $(\mathsf{Input}, \mathsf{v})$ to obtain the function output y.

– If dealer is P_2 or P_3, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ sets $\mathsf{v} = 0$ and performs the protocol steps honestly.

Figure 4.12: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ for corrupt P_1

Shares unknown to $\mathcal{A}$ are sampled randomly in the simulation, whereas in the real protocol, they are sampled using the pseudorandom function (PRF). The indistinguishability of the simulation thus follows by a reduction to the security of the PRF. The same holds for the rest of the blocks.

The simulation for the joint sharing protocol (Π_{JSh}) is similar to that of the sharing protocol. The protocol's design is such that the simulator will always know the value to be sent as part of the joint sharing protocol. The communication is constituted by jsnd calls and is emulated according to the simulation of $\mathcal{F}_{\mathsf{jsnd}}$.

Multiplication Protocol (Π_{Mult} Fig. 4.4)

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Mult}}}$ emulates $\mathcal{F}_{\mathsf{MultPre}}$ for a corrupt P_1 and obtains $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab}.

Online:

– Computes $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ honestly.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ – i) $\mathcal{A}$ as sender to send y_1 to P_2, and ii) $\mathcal{A}$ as receiver to obtain y_2 from P_2.

– Simulates joint sharing for a corrupt sender as discussed earlier.

Figure 4.13: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_1

Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Mult}}}$ emulates $\mathcal{F}_{\mathsf{MultPre}}$ for a corrupt P_3 and obtains $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab}.

Online:

– Computes $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ honestly.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as sender to send y_1 to P_2 and y_2 to P_1.

– Simulates joint sharing for a corrupt receiver as discussed earlier.

Figure 4.14: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_3

Reconstruction Protocol (Π_{Rec}, Fig. 4.6) Using the input of $\mathcal{A}$ obtained during simulation of sharing protocol, $\mathcal{S}_{\Pi_{\mathsf{Rec}}}$ invokes $\mathcal{F}_{\mathrm{GOD}}$ on behalf of $\mathcal{A}$ and obtains the function output y in clear. $\mathcal{S}_{\Pi_{\mathsf{Rec}}}$ calculates the missing share of $\mathcal{A}$ using y and the other shares. The missing share is then communicated to $\mathcal{A}$ by emulating the $\mathcal{F}_{\mathsf{jsnd}}$ functionality.

Chapter 5

Tetrad: 4PC Fair and Robust Protocols

This chapter provides details for the Layer I blocks of our 4PC framework Tetrad. Some of the results in this chapter resulted in publications at NDSS'20 [38] and NDSS'22 [87]. Depending on the sensitivity of the application and the underlying data, we may want different levels of security. For this, we propose multiple variants of the framework, covering fairness (**Tetrad**) and robustness (**Tetrad-R$^{\text{I}}$**, **Tetrad-R$^{\text{II}}$**) guarantees. Comparison of Tetrad with actively secure 4PC PPML frameworks, in terms of the communication for multiplication, is presented in Table 5.1.

Work	#Active Parties	Security	Multiplication		Multiplication with Truncation[a]		Conversions[b]
			Comm_{pre}	Comm_{on}[c]	Comm_{pre}	Comm_{on}	
Mazloom et al. [98]	4	Abort	2ℓ	4ℓ	2ℓ	4ℓ	A-B
Trident [38]	3	Fair	3ℓ	3ℓ	6ℓ	3ℓ	A-B-G
Tetrad	2	Fair	2ℓ	3ℓ	2ℓ	3ℓ	A-B-G
SWIFT (4PC) [85]	2	GOD	3ℓ	3ℓ	4ℓ	3ℓ	A-B
Fantastic Four [46]	3	GOD	-	$6(\ell+\kappa)$	$76(\ell+\kappa)+54x+12$	$9\ell+6\kappa$	A-B
Tetrad-R$^{\text{I}}$	2	GOD	2ℓ	3ℓ	2ℓ	3ℓ	A-B-G
Tetrad-R$^{\text{II}}$	2	GOD	3ℓ	3ℓ	3ℓ	3ℓ	A-B-G

[a] ℓ - size of ring in bits, x - number of bits for the fractional part in FPA semantics.
[b] A, B, G indicate support for arithmetic, boolean, and garbled worlds respectively.
[c] 'Comm' - communication, 'pre' - preprocessing, 'on' - online

Table 5.1: Comparison of malicious 4PC frameworks for PPML

5.1 Preliminaries and Definitions

We consider 4 parties denoted by $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, active adversary that can corrupt at most 1 party.

5.1.1 Sharing Semantics

For the arithmetic and boolean sharing, we follow a $(4, 1)$ replicated secret sharing (RSS) [38], where a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is split into four shares. To leverage the benefits of the preprocessing paradigm, we associate meaning to the shares and demarcate the parties in terms of their roles. Three of the shares of a $(4, 1)$ RSS can be generated in the preprocessing phase independent of the value to be shared, and their sum can be interpreted as a mask. The fourth share, dependent on v, can be computed in the online phase and can be treated as the masked value. We denote the three preprocessed shares as $\lambda^1_\mathsf{v}, \lambda^2_\mathsf{v}, \lambda^3_\mathsf{v}$ and the mask as $\lambda_\mathsf{v} = \lambda^1_\mathsf{v} + \lambda^2_\mathsf{v} + \lambda^3_\mathsf{v}$. The masked value is denoted as m_v, and $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_\mathsf{v}$.

Type	P_0	P_1	P_2	P_3
$[\cdot]$-sharing	–	v^1	v^2	–
$(\!(\cdot)\!)$-sharing	–	v^1	v^2	v^3
$\langle\cdot\rangle$-sharing[a]	–	$(\mathsf{v}^1, \mathsf{v}^3)$	$(\mathsf{v}^2, \mathsf{v}^3)$	$(\mathsf{v}^1, \mathsf{v}^2)$
$[\![\cdot]\!]$-sharing[b]	$(\lambda^1_\mathsf{v}, \lambda^2_\mathsf{v}, \lambda^3_\mathsf{v})$	$(\mathsf{m}_\mathsf{v}, \lambda^1_\mathsf{v}, \lambda^3_\mathsf{v})$	$(\mathsf{m}_\mathsf{v}, \lambda^2_\mathsf{v}, \lambda^3_\mathsf{v})$	$(\mathsf{m}_\mathsf{v}, \lambda^1_\mathsf{v}, \lambda^2_\mathsf{v})$

[a] $\mathsf{v} = \mathsf{v}^1 + \mathsf{v}^2 + \mathsf{v}^3$ [b] $\lambda_\mathsf{v} = \lambda^1_\mathsf{v} + \lambda^2_\mathsf{v} + \lambda^3_\mathsf{v}$, $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_\mathsf{v}$

Table 5.2: Sharing semantics for a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ in Tetrad.

Next, we distinguish the four parties into two sets; the *eval* set $\mathcal{E} = \{P_1, P_2\}$ which is assigned the task of carrying out the computation, and is active throughout the online phase. The *helper* set $\mathcal{D} = \{P_0, P_3\}$, is used to assist $\mathcal{E}$ in verification, and so it is only active towards the end of the computation. Complying with the roles and RSS format, the distribution is done as follows: $P_0 : \{\lambda^1_\mathsf{v}, \lambda^2_\mathsf{v}, \lambda^3_\mathsf{v}\}, P_1 : \{\lambda^1_\mathsf{v}, \lambda^3_\mathsf{v}, \mathsf{m}_\mathsf{v}\}, P_2 : \{\lambda^2_\mathsf{v}, \lambda^3_\mathsf{v}, \mathsf{m}_\mathsf{v}\}$, and $P_3 : \{\lambda^1_\mathsf{v}, \lambda^2_\mathsf{v}, \mathsf{m}_\mathsf{v}\}$. The shares are distributed among $\mathcal{D}$ such that P_3 gets m_v whereas P_0 gets all the shares of λ_v. In the preprocessing phase, P_0 computes a part of the data needed for verification (cf. Fig. 5.3) using its input independent shares, which is communicated to P_3. This enables a verification in the online, without P_0, for the fair protocols.

The RSS sharing semantics is presented in Table 5.2, denoted by $[\![\cdot]\!]$, in a modular way with the help of three intermediate sharing semantics $[\cdot]$, $(\!(\cdot)\!)$ and $\langle\cdot\rangle$. All the sharings used are linear i.e. given sharings of values $\mathsf{v}_1, \ldots, \mathsf{v}_m$ and public constants $c_1, \ldots, c_m$, sharing of $\sum_{i=1}^m c_i \mathsf{v}_i$ can be computed non-interactively for an integer m.

Notation 5.1 *(a) For the $[\![\cdot]\!]$-shares of n values $\mathsf{a}_1, \ldots, \mathsf{a}_n$, $\gamma_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^{n} \lambda_{\mathsf{a}_i}$ and $\mathsf{m}_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^{n} \mathsf{m}_{\mathsf{a}_i}$ (b) We use superscripts $\mathbf{B}$, and $\mathbf{G}$ to denote sharing semantics in boolean, and garbled world, respectively– $[\![\cdot]\!]^{\mathbf{B}}$, $[\![\cdot]\!]^{\mathbf{G}}$. We omit the superscript for arithmetic world.*

Sharing semantics for boolean sharing over $\mathbb{Z}_2$ is similar to arithmetic sharing except that addition is replaced with XOR. The semantics for garbled sharing are described in §5.3 with the relevant context.

5.1.1.1 $\mathcal{F}_{\mathsf{zero}}$ - Generating additive shares of zero

In Tetrad, we make use of a functionality $\mathcal{F}_{\mathsf{zero}}$ to enable parties P_0, P_i obtain Z_i for $i \in \{1, 2, 3\}$ such that $Z_1 + Z_2 + Z_3 = 0$. We observe that the functionality can be instantiated non-interactively using the pre-shared keys (cf. §2.5.1). For this, parties in $\mathcal{P} \setminus \{P_j\}$ sample random value r_j for $j \in \{1, 2, 3\}$. The shares are then defined as $Z_1 = \mathsf{r}_3 - \mathsf{r}_2$, $Z_2 = \mathsf{r}_1 - \mathsf{r}_3$ and $Z_3 = \mathsf{r}_2 - \mathsf{r}_1$.

5.1.2 Joint-Send (jsnd) Primitive

The Joint-Send (jsnd) primitive, for the case of security with fairness, allows to parties P_i, P_j to relay a message v to a third party P_k ensuring either the delivery of the message or `abort` in case of inconsistency. Towards this, P_i sends v to P_k, while P_j sends a hash of the same ($\mathsf{H}(\mathsf{v})$) to P_k. Party P_k accepts the message if the hash values are consistent and `abort` otherwise. Note that the communication of the hash can be clubbed together for several instances and be deferred to the end of the protocol, amortizing the cost.

Joint-Send (jsnd) for robust protocols The jsnd primitive (Fig. 5.1), for the case of robustness, allows two senders P_i, P_j to relay a common message, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, to recipient P_k, either by ensuring successful delivery of v, or by establishing a Trusted Third Party (TTP) among the parties. The instantiation of jsnd can be viewed as consisting of two phases (*send, verify*), where the *send* phase consists of P_i sending v to P_k and the rest of the protocol steps go to *verify* phase (which ensures correct *send* or TTP identification). This requires 1 round of interaction and ℓ bits of communication. To leverage amortization, *verify* will be executed only once, at the end the computation, requiring 2 rounds.

Note that the appropriate instantiation of jsnd is used depending on the security guarantee. For simplicity, protocols where the fair and robust variants only differ in the instantiation of jsnd used, we give a common construction for both.

Notation 5.2 *Protocol* Π_{jsnd} *denotes the instantiation of Joint-Send (*jsnd*) primitive. We say that* P_i, P_j jsnd v *to* P_k *when they invoke* $\Pi_{\mathsf{jsnd}}(P_i, P_j, \mathsf{v}, P_k)$.

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

Input(s): $P_i, P_j : \mathsf{v}$, $P_k : \perp$, **Output:** $P_i, P_j : \perp/\mathsf{TTP}$, $P_k : \mathsf{v}/\mathsf{TTP}$.

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

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

Figure 5.1: Joint-Send for robust protocols in Tetrad.

Lemma 5.1 (Communication) *Protocol* Π_{jsnd} *(Fig. 5.1) requires an amortized communication of* ℓ *bits and* 1 *round.*

Proof: In the protocol $\Pi_{\mathsf{jsnd}}(P_i, P_j, \mathsf{v}, P_k)$ for the fair variant, P_i communicates v to P_k requiring communication of ℓ bits and one round. The hash value communication from P_j to P_k can be clubbed for multiple instances with the same set of parties and hence the cost gets amortized. The analysis is similar for the robust case as well. Here, though the verification consists of multiple steps, the cost gets amortized over multiple instances. □

5.2 Arithmetic / Boolean 4PC

This section covers the details of our 4PC protocol Tetrad over an arithmetic ring $\mathbb{Z}_{2^\ell}$. We begin by explaining the sharing protocol in §5.2.1, multiplication with abort in §5.2.2, and the reconstruction in §5.2.3. Lastly, the details on elevating the security to fairness are presented in §5.2.3.1 and to robustness in §5.2.4.

5.2.1 Sharing

Protocol Π_{Sh} (Fig. 5.2) enables P_i to generate $[\![\cdot]\!]$-share of a value v. During the preprocessing phase, λ-shares are sampled non-interactively using the pre-shared keys (cf. §2.5.1) in a way

that P_i will get the entire mask λ_{v}. During the online phase, P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2, P_3, which exchange the hash values to check for consistency. Parties abort in the fair protocol in case of inconsistency, whereas for robust security, parties proceed with a default value.

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

Input(s): $P_i : \mathsf{v}$, **Output:** $[\![\mathsf{v}]\!]$.

Preprocessing: Sample as follows: $P_i, P_0, P_1, P_3 : \lambda_{\mathsf{v}}^1 \mid P_i, P_0, P_2, P_3 : \lambda_{\mathsf{v}}^2 \mid P_i, P_0, P_1, P_2 : \lambda_{\mathsf{v}}^3$

Online:

1. P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2, P_3.
2. P_1, P_2, P_3 mutually exchange $\mathsf{H}(\mathsf{m}_{\mathsf{v}})$ and accept the sharing if there exists a majority. Else parties `abort` for the case of fairness and accepts a default value for the case of robust security.

Figure 5.2: $[\![\cdot]\!]$-sharing of a value v by party P_i in Tetrad.

Lemma 5.2 (Communication) *Protocol* Π_{Sh} *(Fig. 5.2) requires an amortized communication of at most* 3ℓ *bits and* 1 *round in the online phase.*

Proof: The preprocessing of Π_{Sh} is non-interactive as the parties sample non interactively using key setup $\mathcal{F}_{\text{SETUP}}$ (§2.5.1). in the online phase, P_i sends m_{v} to P_1, P_2, P_3 resulting in 1 round and communication of at most 3ℓ bits ($P_i = P_0$). The next round of hash exchange can be clubbed for several instances and the cost gets amortized over multiple instances. □

5.2.1.1 Joint Sharing

Protocol Π_{JSh} enables parties P_i, P_j to generate $[\![\cdot]\!]$-share of a value v. The protocol is similar to Π_{Sh} except that P_j ensures the correctness of the sharing performed by P_i. During the preprocessing, λ-shares are sampled such that both P_i, P_j will get the entire mask λ_{v}. During the online phase, P_i, P_j compute and $\mathsf{jsnd}\ \mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ to parties P_1, P_2, P_3.

For joint-sharing a value v possessed by P_0 along with another party in the preprocessing, the communication can be optimized further. The protocol steps based on the (P_i, P_j) pair are summarised below:

- $(P_0, P_1) : \mathcal{P} \setminus \{P_2\}$ sample $\lambda_{\mathsf{v}}^1 \in_R \mathbb{Z}_{2^\ell}$; Parties set $\lambda_{\mathsf{v}}^2 = \mathsf{m}_{\mathsf{v}} = 0$; P_0, P_1 jsnd $\lambda_{\mathsf{v}}^3 = -\mathsf{v} - \lambda_{\mathsf{v}}^1$ to P_2.
- $(P_0, P_2) : \mathcal{P} \setminus \{P_3\}$ sample $\lambda_{\mathsf{v}}^3 \in_R \mathbb{Z}_{2^\ell}$; Parties set $\lambda_{\mathsf{v}}^1 = \mathsf{m}_{\mathsf{v}} = 0$; P_0, P_2 jsnd $\lambda_{\mathsf{v}}^2 = -\mathsf{v} - \lambda_{\mathsf{v}}^3$ to P_3.
- $(P_0, P_3) : \mathcal{P} \setminus \{P_1\}$ sample $\lambda_{\mathsf{v}}^2 \in_R \mathbb{Z}_{2^\ell}$; Parties set $\lambda_{\mathsf{v}}^3 = \mathsf{m}_{\mathsf{v}} = 0$; P_0, P_3 jsnd $\lambda_{\mathsf{v}}^1 = -\mathsf{v} - \lambda_{\mathsf{v}}^1$ to P_1.

5.2.2 Multiplication

Given the shares of a, b, the goal of the multiplication protocol is to generate shares of $\mathsf{z} = \mathsf{ab}$. The protocol is designed such that parties P_1, P_2 obtain a masked version of the output z, say $\mathsf{z} - \mathsf{r}$ in the online phase, and P_0, P_3 obtain the mask r in the preprocessing phase. Parties then generate $[\![\cdot]\!]$-sharing of these values by executing Π_{JSh}, and locally compute $[\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$ to obtain the final output.

Online Note that,

$$\begin{aligned} \mathsf{z} - \mathsf{r} = \mathsf{ab} - \mathsf{r} &= (\mathsf{m}_\mathsf{a} - \lambda_\mathsf{a})(\mathsf{m}_\mathsf{b} - \lambda_\mathsf{b}) - \mathsf{r} \\ &= \mathsf{m}_\mathsf{ab} - \mathsf{m}_\mathsf{a}\lambda_\mathsf{b} - \mathsf{m}_\mathsf{b}\lambda_\mathsf{a} + \gamma_\mathsf{ab} - \mathsf{r} \quad \text{(cf. notation 5.1)} \end{aligned} \tag{5.1}$$

In Eq 5.1, P_1, P_2 can compute m_ab locally, and hence we are interested in computing $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m}_\mathsf{ab}$. Let us view y as $\mathsf{y} = \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3$, where y_1 and y_2 can be computed respectively by P_1 and P_2, and y_3 consists of terms that can be computed by both P_1, P_2.

$$\begin{aligned} P_1 : \mathsf{y}_1 &= -\lambda_\mathsf{a}^1\mathsf{m}_\mathsf{b} - \lambda_\mathsf{b}^1\mathsf{m}_\mathsf{a} + [\gamma_\mathsf{ab} - \mathsf{r}]_1 \\ P_2 : \mathsf{y}_2 &= -\lambda_\mathsf{a}^2\mathsf{m}_\mathsf{b} - \lambda_\mathsf{b}^2\mathsf{m}_\mathsf{a} + [\gamma_\mathsf{ab} - \mathsf{r}]_2 \\ P_1, P_2 : \mathsf{y}_3 &= -\lambda_\mathsf{a}^3\mathsf{m}_\mathsf{b} - \lambda_\mathsf{b}^3\mathsf{m}_\mathsf{a} \end{aligned} \tag{5.2}$$

The preprocessing is set up such that P_1, P_2 receive an additive sharing ($[\cdot]$) of $\gamma_\mathsf{ab} - \mathsf{r}$. Parties P_1, P_2 mutually exchange the missing share to reconstruct y and subsequently $\mathsf{z} - \mathsf{r}$.

Protocol $\Pi_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^\mathsf{t}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{ab}$.

Preprocessing:

1. Locally compute the following:

$$\begin{aligned} P_0, P_1 : \gamma_\mathsf{ab}^1 &= \lambda_\mathsf{a}^1\lambda_\mathsf{b}^3 + \lambda_\mathsf{a}^3\lambda_\mathsf{b}^1 + \lambda_\mathsf{a}^3\lambda_\mathsf{b}^3 \\ P_0, P_2 : \gamma_\mathsf{ab}^2 &= \lambda_\mathsf{a}^2\lambda_\mathsf{b}^3 + \lambda_\mathsf{a}^3\lambda_\mathsf{b}^2 + \lambda_\mathsf{a}^2\lambda_\mathsf{b}^2 \\ P_0, P_3 : \gamma_\mathsf{ab}^3 &= \lambda_\mathsf{a}^1\lambda_\mathsf{b}^2 + \lambda_\mathsf{a}^2\lambda_\mathsf{b}^1 + \lambda_\mathsf{a}^1\lambda_\mathsf{b}^1 \end{aligned}$$

2. P_0, P_3 and P_j sample random $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma_\mathsf{ab}^3 - \mathsf{r}$ for a random

$\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.

3. P_0, P_3 compute $\mathsf{r} = \gamma^3_{\mathsf{ab}} - \mathsf{u}^1 - \mathsf{u}^2$ and set $\mathsf{q} = \mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else set $\mathsf{q} = \mathsf{r}$. P_0, P_3 execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

4. P_0, P_1, P_2 sample random $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$ and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$[a]. P_0 sends $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}$ to P_3.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_a}\mathsf{m_b}$.

1. Locally compute the following:

$$P_1 : \mathsf{y}_1 = -\lambda^1_{\mathsf{a}}\mathsf{m_b} - \lambda^1_{\mathsf{b}}\mathsf{m_a} + \gamma^1_{\mathsf{ab}} + \mathsf{u}^1$$
$$P_2 : \mathsf{y}_2 = -\lambda^2_{\mathsf{a}}\mathsf{m_b} - \lambda^2_{\mathsf{b}}\mathsf{m_a} + \gamma^2_{\mathsf{ab}} + \mathsf{u}^2$$
$$P_1, P_2 : \mathsf{y}_3 = -\lambda^3_{\mathsf{a}}\mathsf{m_b} - \lambda^3_{\mathsf{b}}\mathsf{m_a}$$

2. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m_a}\mathsf{m_b}$.

3. If $\mathsf{isTr} = 1$, P_1, P_2 set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. P_1, P_2 execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.

4. Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

5. *Verification:* P_3 computes $\mathsf{v} = -(\lambda^1_{\mathsf{a}} + \lambda^2_{\mathsf{a}})\mathsf{m_b} - (\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{b}})\mathsf{m_a} + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ and sends $\mathsf{H}(\mathsf{v})$ to P_1 and P_2. Parties P_1, P_2 `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s})$.

[a] For the fair protocol, it is enough for P_0, P_1, P_2 to sample s directly.

Figure 5.3: Multiplication with / without truncation in Tetrad.

Verification To ensure the correctness of the values exchanged, we use the assistance of P_3. Concretely, P_3 obtains $\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s}$, where s is a random mask known to P_0, P_1, P_2. For this P_3 needs $\gamma_{\mathsf{ab}} + \mathsf{s}$, which it obtains from the preprocessing phase. The mask s is used to prevent the leakage from γ_{ab} to P_3. P_3 computes a hash of $\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s}$ and sends it to P_1, P_2, which `abort` if it is inconsistent.

Preprocessing Parties should obtain the following values from the preprocessing phase:

$$\text{i)}\ \ P_1, P_2 : [\gamma_{\mathsf{ab}} - \mathsf{r}] \quad \Big| \quad \text{ii)}\ \ P_0, P_3 : \mathsf{r} \quad \Big| \quad \text{iii)}\ \ P_3 : \gamma_{\mathsf{ab}} + \mathsf{s}$$

For i) and ii), let $\gamma_{\mathsf{ab}} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \gamma^3_{\mathsf{ab}}$, where P_0 along with P_i can compute γ^i_{ab} for $i \in \{1, 2, 3\}$. For P_1, P_2, to form an additive sharing of $(\gamma_{\mathsf{ab}} - r)$, it suffices for them to define their share as $\gamma^i_{\mathsf{ab}} + [\gamma^3_{\mathsf{ab}} - r]$. Instead of sampling a random r, P_0, P_3, along with P_i, sample the share for

$\gamma^3_{\mathsf{ab}} - \mathsf{r}$ as u^i for $i \in \{1,2\}$. P_0, P_3 compute r as $\gamma^3_{\mathsf{ab}} - \mathsf{u}^1 - \mathsf{u}^2$.

For iii), P_3 needs $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}$. To tackle this, P_0, P_1, P_2 sample $\mathsf{s}_1, \mathsf{s}_2$, and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$. P_0, P_i, for $i \in \{1,2\}$, $\mathsf{jsnd}\ \gamma^i_{\mathsf{ab}} + \mathsf{s}_i$ to P_3. This requires a communication of 2 elements. As an optimization, P_0 sends w to P_3. If P_0 is malicious, it might send a wrong value to P_3. However, in this case, every party in the online phase would be honest. And since P_1, P_2 do not use w in their computation, any error in w is bound to get caught in the verification phase.

Lemma 5.3 (Communication) *Protocol* Π_{Mult} *(Fig. 5.3) (in* Tetrad*) requires* 2ℓ *bits of communication in the preprocessing phase, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: During preprocessing, sampling of values $\mathsf{u}^1, \mathsf{u}^2$ are performed non-interactively using $\mathcal{F}_{\text{Setup}}$. A communication of ℓ bits is required for the joint sharing of q by P_0, P_3 as explained in §5.2.1.1. In addition, P_0 communicates w to P_3 requiring additional ℓ bits. During online, two instances of Π_{jsnd} are executed in parallel resulting in a communication of 2ℓ bits and 1 round. This is followed by a joint sharing by P_1, P_2 for which an additional communication of ℓ bits are required. However, in joint sharing, the communication is from P_1 to P_3 and the same can be deferred till the verification stage. Thus the online round is retained as 1 in an amortized sense. □

5.2.2.1 Truncation

To accommodate truncation, the multiplication protocol is modified as follows. P_1, P_2 locally truncate $(\mathsf{z} - \mathsf{r})$ and generate $[\![\cdot]\!]$-shares of it in the online phase. Similarly, P_0, P_3 truncate r in the preprocessing phase and generate its $[\![\cdot]\!]$-shares. Parties locally compute $[\![\mathsf{z}^{\mathsf{t}}]\!] = [\![(\mathsf{z} - \mathsf{r})^{\mathsf{t}}]\!] + [\![\mathsf{r}^{\mathsf{t}}]\!]$.

5.2.2.2 Multiplication with constant

Multiplication by a constant in MPC is typically local. Given constant α and $[\![\mathsf{v}]\!]$, the $[\![\cdot]\!]$-shares of the product $\mathsf{y} = \alpha\mathsf{v}$ can be locally computed as per (5.3).

$$\mathsf{m}_{\mathsf{y}} = \alpha\mathsf{m}_{\mathsf{u}}, \quad \lambda^1_{\mathsf{y}} = \alpha\lambda^1_{\mathsf{v}}, \quad \lambda^2_{\mathsf{y}} = \alpha\lambda^2_{\mathsf{v}}, \quad \lambda^3_{\mathsf{y}} = \alpha\lambda^3_{\mathsf{v}} \tag{5.3}$$

However, in FPA, we need to perform a truncation on the output. For this, note that the product can be written as $\alpha\mathsf{v} = \beta^1 + \beta^2$ where $\beta^1 = \alpha.(\mathsf{m}_{\mathsf{v}} - \lambda^3_{\mathsf{v}})$ and $\beta^2 = \alpha.(-\lambda^1_{\mathsf{v}} - \lambda^2_{\mathsf{v}})$. P_1, P_2 locally truncate β^1 and execute Π_{JSh}, while P_0, P_3 do the same with β^2.

5.2.2.3 Special multiplication protocol Π_{MultS}

Given the $\langle\cdot\rangle$-shares of values a, b with P_0 knowing the entire shares of both $\langle\mathsf{a}\rangle$ and $\langle\mathsf{b}\rangle$, protocol Π_{MultS} (Fig. 5.4) computes $\langle\mathsf{z}\rangle$ for $\mathsf{z} = \mathsf{ab}$.

Protocol $\Pi_{\mathsf{MultS}}(\langle\mathsf{a}\rangle, \langle\mathsf{b}\rangle)$

Input(s): $\langle\mathsf{a}\rangle, \langle\mathsf{b}\rangle$, **Output:** $\langle\mathsf{z}\rangle$ where $\mathsf{z} = \mathsf{ab}$.

1. Parties invoke $\mathcal{F}_{\mathsf{zero}}$ (§5.1.1.1) to enable P_0, P_j obtain Z_j for $j \in \{1,2,3\}$ such that $Z_1+Z_2+Z_3 = 0$. Then,

$$P_0, P_1 \text{ jsnd } (\mathsf{ab})^1 = \mathsf{a}^1\mathsf{b}^3 + \mathsf{a}^3\mathsf{b}^1 + \mathsf{a}^3\mathsf{b}^3 + Z_1 \text{ to } P_2.$$
$$P_0, P_2 \text{ jsnd } (\mathsf{ab})^2 = \mathsf{a}^2\mathsf{b}^3 + \mathsf{a}^3\mathsf{b}^2 + \mathsf{a}^2\mathsf{b}^2 + Z_2 \text{ to } P_3.$$
$$P_0, P_3 \text{ jsnd } (\mathsf{ab})^3 = \mathsf{a}^1\mathsf{b}^2 + \mathsf{a}^2\mathsf{b}^1 + \mathsf{a}^1\mathsf{b}^1 + Z_3 \text{ to } P_1.$$

– Set $\langle\mathsf{z}\rangle$ as $\langle\mathsf{z}\rangle^1 = (\mathsf{ab})^3$, $\langle\mathsf{z}\rangle^2 = (\mathsf{ab})^2$, $\langle\mathsf{z}\rangle^3 = (\mathsf{ab})^1$.

Figure 5.4: Special multiplication of $\langle\cdot\rangle$-shares in Tetrad.

5.2.3 Reconstruction

Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}, \mathsf{v})$ (Fig. 5.5) enables parties in $\mathcal{P}$ to compute v, given its $[\![\cdot]\!]$-share and achieves security with abort. Note that each party misses one share to reconstruct the output, and the other 3 parties hold this share. 2 out of the 3 parties will jsnd the missing share to the party that lacks it. Reconstruction towards a single party can be viewed as a special case.

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

Input(s): $[\![\mathsf{v}]\!]$, **Output:** v.

1. P_1, P_0 jsnd λ_{v}^1 to P_2; P_2, P_0 jsnd λ_{v}^3 to P_3;
   P_3, P_0 jsnd λ_{v}^2 to P_1; P_1, P_2 jsnd m_{v} to P_0.
2. Parties compute $\mathsf{v} = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}^1 - \lambda_{\mathsf{v}}^2 - \lambda_{\mathsf{v}}^3$.

Figure 5.5: Reconstruction (with abort security) of value v among $\mathcal{P}$ in Tetrad.

Lemma 5.4 (Communication) *Protocol* Π_{Rec} *(Fig. 5.5) requires an amortized communication of* 4ℓ *bits and* 1 *round in the online phase.*

Proof: The protocol involves 4 invocations of Π_{jsnd} protocol and the communication follows from Lemma 5.1. □

5.2.3.1 Achieving Fairness

Here, we show how to extend the security of Tetrad from abort to fairness. Before proceeding with the output reconstruction, we must ensure that all the honest parties are alive after the verification phase. For this, all the parties maintain an *aliveness* bit, say b, which is initialized to `continue`. If the verification phase is not successful for a party, it sets b = `abort`. In the first round of reconstruction, the parties mutually exchange their b bit and accept the value that forms the majority. Since we have only one corruption, it is guaranteed that all the honest parties will agree on b. If b = `continue`, the parties exchange their missing shares and accept the majority. As per the sharing semantics, every missing share is possessed by three parties, out of which there can be at most one corruption. As an optimization, for instances where many values are reconstructed, two out of the three parties can send the share while the third can send a hash of the same.

Looking ahead, a similar reconstruction will be used for the robust variants as well. However, there is no need to perform an explicit aliveness check as it will be taken care of by the verification of jsnd instances.

5.2.4 Achieving Robustness

In this section, we show how to extend the security of Tetrad to robustness. We provide two variants with different trade-offs in the communication for multiplication – i) Tetrad-R$^{\text{I}}$: It has the same amortized communication complexity as that of Tetrad but requires verification in the preprocessing phase, and ii) Tetrad-R$^{\text{II}}$: It avoids the verification in Tetrad-R$^{\text{I}}$ but incurs a communication overhead of 1 element in the preprocessing phase over Tetrad.

5.2.4.1 Tetrad-R$^{\text{I}}$

On a high level, we make two modifications to the multiplication protocol Π_{Mult} (Fig. 5.3). In the preprocessing, communication comes from a Π_{JSh} in step 3 of the protocol, and the value w sent by P_0 to P_3, in step 4. To get robustness, the robust variant of Π_{JSh} is used. To ensure the correctness of w, we introduce Π_{VrfyP0} (Fig. 5.7). If Π_{VrfyP0} fails, parties identify a TTP in the preprocessing phase itself. The second modification is in the online phase, which proceeds as that of Π_{Mult}. If any `abort` happens, P_0 is assigned as the TTP. Since P_0 does not participate in the online phase of the multiplication, and its communication in the preprocessing has been verified via Π_{VrfyP0}, this assignment is safe.

Verifying the communication by P_0: In Π_{Mult} (Fig. 5.3) protocol, P_0 computes and sends $\mathsf{w} = \gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}} + \mathsf{s}_1 + \mathsf{s}_2$ to P_3 with P_0, P_1, P_2 knowing $\mathsf{s}_1, \mathsf{s}_2$ in clear. Note that $\mathsf{w} = \mathsf{w}^1 + \mathsf{w}^2$ for $\mathsf{w}^1 = \gamma^1_{\mathsf{ab}} + \mathsf{s}_1$ and $\mathsf{w}^2 = \gamma^2_{\mathsf{ab}} + \mathsf{s}_2$. Also, P_0 along with P_1, P_2 and P_3 possess the values $\mathsf{w}^1, \mathsf{w}^2$ and w respectively. Checking the correctness of w reduces to verifying $\mathsf{w} = \mathsf{w}^1 + \mathsf{w}^2$.

To verify this relation for all M multiplication gates in the circuit, i.e. $\{\mathsf{w}_j \stackrel{?}{=} \mathsf{w}^1_j + \mathsf{w}^2_j\}_{j\in[M]}$, one approach is to compute a random linear combination and verify the relation on the sum. While working over a field $\mathbb{F}_p$, this solution has an error probability $1/|\mathbb{F}_p|$, where $|\mathbb{F}_p|$ denotes the size of $\mathbb{F}_p$. However, this solution does not work naively over rings since not every element in the ring has an inverse, unlike fields. Concretely, the check can still pass with a probability of at most $1/2$ [1, 27]. To reduce the cheating probability, the check is repeated κ times, thereby bounding the cheating probability by $1/2^\kappa$. As an optimization, it is sufficient to choose the random combiners from $\{0, 1\}$. Thus, parties need to sample only a binary string of M bits using the shared key for one check. The formal verification protocol appears in Fig. 5.6.

Protocol $\Pi_{\mathsf{VrfyP0}}(\{[\mathsf{w}_j]\}_{j=1}^M)$

Input(s): $P_0, P_1 : \mathsf{w}^1_j \;\big|\; P_0, P_2 : \mathsf{w}^2_j \;\big|\; P_0, P_3 : \mathsf{w}_j \;\big|$, for $j = 1, \ldots, M$.
Output: Whether $\mathsf{w}_j = \mathsf{w}^1_j + \mathsf{w}^2_j$ or not, for $j = 1, \ldots, M$.

Repeat the following κ times, in parallel.

1. Sample random values $\tau_1, \ldots, \tau_M \in \mathbb{Z}_{2^\ell}$.
2. Locally compute: $P_0, P_1 : \mathsf{e}^1 = \sum_{j=1}^M \tau_j \mathsf{w}^1_j$; $P_0, P_2 : \mathsf{e}^2 = \sum_{j=1}^M \tau_j \mathsf{w}^2_j$; $P_0, P_3 : \mathsf{e} = \sum_{j=1}^M \tau_j \mathsf{w}_j$.
3. (P_0, P_1), (P_0, P_2) and (P_0, P_3) generate $[\![\cdot]\!]$-shares of $\mathsf{e}^1, \mathsf{e}^2$ and e respectively using Π_{JSh}.
4. Locally compute $[\![\mathsf{g}]\!] = [\![\mathsf{e}]\!] - [\![\mathsf{e}^1]\!] - [\![\mathsf{e}^2]\!]$.
5. Robustly reconstruct g and check if $\mathsf{g} \stackrel{?}{=} 0$.

If for all κ repetitions, $\mathsf{g} = 0$, then continue with rest of the computation. Else, P_0 is identified to be corrupt and $\mathsf{TTP} = P_1$.

Figure 5.6: Verifying P_0's communication in the multiplication protocol of Tetrad-R$^{\mathsf{l}}$: Approach 1

Another approach, that avoids the repetition in the Π_{VrfyP0} protocol above, is to perform a similar check over a Galois ring [1, 27]. To carry out the verification, the extended ring $\mathbb{Z}_{2^\ell}/f(x)$ is used, which is the ring of all polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ modulo an irreducible polynomial f of degree d over $\mathbb{Z}_2$. Here, each element in $\mathbb{Z}_{2^\ell}$ is lifted to a d-degree polynomial in $\mathbb{Z}_{2^\ell}[x]/f(x)$ (which results in blowing up the communication by a factor d). Given this, to verify the M values, further packing is performed. More concretely, assume that d divides M

and $M = d \cdot q$. For $j = 1, \ldots, q$, public polynomial g_j and shared polynomials g_j^1 and g_j^2 are defined for each set of d values $\{\mathsf{w}, \mathsf{w}^1, \mathsf{w}^2\}$, all of which are then combined to check whether $\{\mathsf{w}_j \stackrel{?}{=} \mathsf{w}_j^1 + \mathsf{w}_j^2\}_{j \in [M]}$. We describe the polynomial with respect to $j = 1$ below.

$$
\begin{aligned}
g_1 &= \mathsf{w}_1 + X \cdot \mathsf{w}_2 + \ldots + X^{d-1} \cdot \mathsf{w}_d \\
g_1^1 &= \mathsf{w}_1^1 + X \cdot \mathsf{w}_2^1 + \ldots + X^{d-1} \cdot \mathsf{w}_d^1 \\
g_1^2 &= \mathsf{w}_1^2 + X \cdot \mathsf{w}_2^2 + \ldots + X^{d-1} \cdot \mathsf{w}_d^2
\end{aligned}
$$

Now, parties sample random values $\mathsf{r}_1, \ldots, \mathsf{r}_q \in \mathbb{Z}_{2^\ell}/f(x)$ and compute $g = \sum_{j=1}^q \mathsf{r}_j g_j$, $g^1 = \sum_{j=1}^q \mathsf{r}_j g_j^1$ and $g^2 = \sum_{j=1}^q \mathsf{r}_j g_j^2$. This is followed by robustly reconstructing $g - g^1 - g^2$ and verifying if this value is 0. If not, P_0 is identified to be a corrupt and computation is carried out by a TTP. The formal verification protocol appears in Fig. 5.7.

Protocol $\Pi_{\mathsf{VrfyP0}}(\{[\mathsf{w}_j]\}_{j=1}^M)$

Input(s): $P_0, P_1 : \mathsf{w}_j^1 \;\big|\; P_0, P_2 : \mathsf{w}_j^2 \;\big|\; P_0, P_3 : \mathsf{w}_j \;\big|$, for $j = 1, \ldots, M$.
Output: Whether $\mathsf{w}_j = \mathsf{w}_j^1 + \mathsf{w}_j^2$ or not, for $j = 1, \ldots, M$.

1. Define the following polynomials over $\mathbb{Z}_{2^\ell}/f(x)$ for $j \in [q]$.

$$
\begin{aligned}
g_j &= \mathsf{w}_{1+(j-1)d} + X \cdot \mathsf{w}_{2+(j-1)d} + \ldots + X^{d-1} \cdot \mathsf{w}_{d+(j-1)d} \\
g_j^1 &= \mathsf{w}_{1+(j-1)d}^1 + X \cdot \mathsf{w}_{2+(j-1)d}^1 + \ldots + X^{d-1} \cdot \mathsf{w}_{d+(j-1)d}^1 \\
g_j^2 &= \mathsf{w}_{1+(j-1)d}^2 + X \cdot \mathsf{w}_{2+(j-1)d}^2 + \ldots + X^{d-1} \cdot \mathsf{w}_{d+(j-1)d}^2
\end{aligned}
$$

2. Parties generate random values $\mathsf{r}_1, \ldots, \mathsf{r}_q \in \mathbb{Z}_{2^\ell}/f(x)$, and compute $g = \sum_{j=1}^q \mathsf{r}_j g_j$, $g^1 = \sum_{j=1}^q \mathsf{r}_j g_j^1$ and $g^2 = \sum_{j=1}^q \mathsf{r}_j g_j^2$.
3. Parties execute $\Pi_{\mathsf{JSh}}(P_0, P_1, g^1)$, $\Pi_{\mathsf{JSh}}(P_0, P_2, g^2)$ and $\Pi_{\mathsf{JSh}}(P_0, P_3, g)$ to generate $[\![g^1]\!], [\![g^2]\!]$ and $[\![g]\!]$, respectively.
4. Parties robustly reconstruct $g - g^1 - g^2$ and check equality to 0. If it is 0, then parties continue with rest of the computation. Else, P_0 is identified to be corrupt and $\mathsf{TTP} = P_1$.

Figure 5.7: Verifying P_0's communication in the multiplication protocol of Tetrad-R$^{\mathsf{I}}$: Approach 2

5.2.4.2 Tetrad-R$^{\mathsf{II}}$

This variant (Fig. 5.8) avoids the verification of P_0 at the cost of communicating 1 extra ring element in the preprocessing, compared to Tetrad-R$^{\mathsf{I}}$. Note that the communication cost of this

protocol is similar to that of the one in SWIFT [85]. We were unable to extend the latter's efficiently to support multi-input multiplication. Hence, we design Tetrad-R$^{\mathsf{II}}$ that has the same communication complexity as SWIFT but also supports multi-input multiplication, as well as truncation without any overhead. In order to get rid of Π_{VrfyP0}, the communication of w from P_0 to P_3 is split into 2 parts. (P_0, P_1) and (P_0, P_2) compute w in parts, and send them to P_3 using jsnd. This modification allows P_3 to compute $\mathsf{y}_1 + \mathsf{s}_1$ and $\mathsf{y}_2 + \mathsf{s}_2$ separately in the online phase. In addition, to enable P_2 to obtain y_1, P_1, P_3 can jsnd $\mathsf{y}_1 + \mathsf{s}_1$ to P_2. P_1 obtains $\mathsf{y}_2 + \mathsf{s}_2$ similarly.

The formal protocol for the robust multiplication in Tetrad-R$^{\mathsf{II}}$, $\Pi^{\mathsf{R}}_{\mathsf{Mult}}$, appears in Fig. 5.8. The primary difference from the fair counterpart is that the communication of w from P_0 to P_3 in the preprocessing is now split into two parts. $(P_0, P_1), (P_0, P_2)$ communicates $\mathsf{w}_1, \mathsf{w}_2$ respectively to P_3 via jsnd.

Protocol $\Pi^{\mathsf{R}}_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).

Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$.

Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{ab}$.

Preprocessing:

1. Parties locally compute the following:

$$P_0, P_1 : \gamma^1_{\mathsf{ab}} = \lambda^1_{\mathsf{a}}\lambda^3_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^3_{\mathsf{b}}$$
$$P_0, P_2 : \gamma^2_{\mathsf{ab}} = \lambda^2_{\mathsf{a}}\lambda^3_{\mathsf{b}} + \lambda^3_{\mathsf{a}}\lambda^2_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^2_{\mathsf{b}}$$
$$P_0, P_3 : \gamma^3_{\mathsf{ab}} = \lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^1_{\mathsf{a}}\lambda^1_{\mathsf{b}}$$

2. P_0, P_3 and P_j sample random $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma^3_{\mathsf{ab}} - \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.

3. P_0, P_3 compute $\mathsf{r} = \gamma^3_{\mathsf{ab}} - \mathsf{u}^1 - \mathsf{u}^2$ and set $\mathsf{q} = \mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else set $\mathsf{q} = \mathsf{r}$. P_0, P_3 execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

4. P_0, P_1, P_2 sample random $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$. P_0, P_j jsnd $\mathsf{w}_j = \gamma^j_{\mathsf{ab}} + \mathsf{s}_j$ to P_3 for $j \in \{1, 2\}$.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} + \mathsf{s}_1 + \mathsf{s}_2$.

1. Parties locally compute the following:

$$P_1, P_3 : \mathsf{y}_1 + \mathsf{s}_1 = -\lambda_{\mathsf{a}}^1 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^1 \mathsf{m}_{\mathsf{a}} + \mathsf{u}^1 + \mathsf{w}_1$$
$$P_2, P_3 : \mathsf{y}_2 + \mathsf{s}_2 = -\lambda_{\mathsf{a}}^2 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^2 \mathsf{m}_{\mathsf{a}} + \mathsf{u}^2 + \mathsf{w}_2$$
$$P_1, P_2 : \mathsf{y}_3 = -\lambda_{\mathsf{a}}^3 \mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}^3 \mathsf{m}_{\mathsf{a}}$$

2. P_1, P_3 jsnd $\mathsf{y}_1 + \mathsf{s}_1$ to P_2, while P_1, P_3 jsnd $\mathsf{y}_2 + \mathsf{s}_2$ to P_1.

3. P_1, P_2 locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}} - \mathsf{s}_1 - \mathsf{s}_2$.

4. If $\mathsf{isTr} = 1$, P_1, P_2 locally set $\mathsf{p} = (\mathsf{z}-\mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z}-\mathsf{r}$. P_1, P_2 execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.

5. Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 5.8: Robust multiplication in $\mathsf{Tetrad\text{-}R^{II}}$.

Lemma 5.5 (Communication) *Protocol* $\Pi^{\mathsf{R}}_{\mathsf{Mult}}$ *(Fig. 5.8) (in* $\mathsf{Tetrad\text{-}R^{II}}$*) requires* 3ℓ *bits of communication in the preprocessing phase, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: During preprocessing, the sampling of values $\mathsf{u}^1, \mathsf{u}^2$ are performed non-interactively using $\mathcal{F}_{\text{SETUP}}$. A communication of ℓ bits is required for the joint sharing of q by P_0, P_3 as explained in §5.2.1.1. In addition, P_0, P_j for $j \in \{1, 2\}$ communicates w_j to P_3 via jsnd requiring additional 2ℓ bits. The online phase is similar to the fair multiplication protocol (Π_{Mult}) and the costs follow from Lemma 5.3. □

5.2.5 Multi-input Multiplication

The goal of 3-input multiplication is to generate $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{abc}$ given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!], [\![\mathsf{c}]\!]$. For this parties proceed similar to the multiplication protocol (see §5.2.2), where they compute $[\![\mathsf{z}]\!] = [\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$. Observe that

$$\begin{aligned}\mathsf{z} - \mathsf{r} &= \mathsf{abc} - \mathsf{r} = (\mathsf{m}_{\mathsf{a}} - \lambda_{\mathsf{a}})(\mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}})(\mathsf{m}_{\mathsf{c}} - \lambda_{\mathsf{c}}) - \mathsf{r} \\ &= \mathsf{m}_{\mathsf{abc}} - \mathsf{m}_{\mathsf{ac}}\lambda_{\mathsf{b}} - \mathsf{m}_{\mathsf{bc}}\lambda_{\mathsf{a}} - \mathsf{m}_{\mathsf{ab}}\lambda_{\mathsf{c}} + \mathsf{m}_{\mathsf{a}}\gamma_{\mathsf{bc}} + \mathsf{m}_{\mathsf{b}}\gamma_{\mathsf{ac}} + \mathsf{m}_{\mathsf{c}}\gamma_{\mathsf{ab}} - \gamma_{\mathsf{abc}} - \mathsf{r}\end{aligned}$$

Similar to the 2-input fair multiplication Π_{Mult} (Fig. 5.3), the goal of the preprocessing phase is to generate additive shares of $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{ac}}, \gamma_{\mathsf{bc}}, \gamma_{\mathsf{abc}} + \mathsf{r}$ among P_1, P_2.

Informally, the terms that P_1, P_2 cannot compute locally for the aforementioned γ values, can be computed by P_0, P_3, as evident from our sharing semantics. P_0, P_3 compute the missing terms and share them among P_1, P_2 in the preprocessing phase. P_1, P_2 proceed with online phase similar to Π_{Mult}, to compute $\mathsf{z} - \mathsf{r}$. Thus the online complexity is retained as that of Π_{Mult} while the preprocessing communication is increased to 9 elements. The protocol appears in Fig. 5.9.

Protocol $\Pi_{\mathsf{Mult3}}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!], [\![\mathsf{c}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{abc}$.

Preprocessing:

1. Computation for γ_{ab}: Invoke Π_{MultS} (Fig. 5.4) on $\langle \lambda^{\mathsf{R}}_{\mathsf{a}} \rangle$ and $\langle \lambda^{\mathsf{R}}_{\mathsf{b}} \rangle$ to generate $\langle \gamma_{\mathsf{ab}} \rangle$.

2. Computation for γ_{ac}:
   - Parties locally compute the following:

$$P_0, P_1 : \gamma^1_{\mathsf{ac}} = \lambda^1_{\mathsf{a}}\lambda^3_{\mathsf{c}} + \lambda^3_{\mathsf{a}}\lambda^1_{\mathsf{c}} + \lambda^3_{\mathsf{a}}\lambda^3_{\mathsf{c}}$$
$$P_0, P_2 : \gamma^2_{\mathsf{ac}} = \lambda^2_{\mathsf{a}}\lambda^3_{\mathsf{c}} + \lambda^3_{\mathsf{a}}\lambda^2_{\mathsf{c}} + \lambda^2_{\mathsf{a}}\lambda^2_{\mathsf{c}}$$
$$P_0, P_3 : \gamma^3_{\mathsf{ac}} = \lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{c}} + \lambda^2_{\mathsf{a}}\lambda^1_{\mathsf{c}} + \lambda^1_{\mathsf{a}}\lambda^1_{\mathsf{c}}$$

   - P_0, P_3 and P_1 sample random $\mathsf{u}^1_{\mathsf{ac}} \in_R \mathbb{Z}_{2^\ell}$. P_0, P_3 compute and $\mathsf{jsnd}\ \mathsf{u}^2_{\mathsf{ac}} = \gamma^3_{\mathsf{ac}} - \mathsf{u}^1_{\mathsf{ac}}$ to P_2.
   - P_0, P_1, P_2 sample random $\mathsf{s}_{\mathsf{ac}} \in_R \mathbb{Z}_{2^\ell}$. P_0 sends $\mathsf{w}_{\mathsf{ac}} = \gamma^1_{\mathsf{ac}} + \gamma^2_{\mathsf{ac}} + \mathsf{s}_{\mathsf{ac}}$ to P_3.

3. Computation for γ_{bc}: Similar to Step 2 (for γ_{ac}). P_1, P_2 obtain $\mathsf{u}^1_{\mathsf{bc}}, \mathsf{u}^2_{\mathsf{bc}}$ respectively such that $\mathsf{u}^1_{\mathsf{bc}} + \mathsf{u}^2_{\mathsf{bc}} = \gamma^3_{\mathsf{bc}}$. P_3 obtains $\mathsf{w}_{\mathsf{bc}} = \gamma^1_{\mathsf{bc}} + \gamma^2_{\mathsf{bc}} + \mathsf{s}_{\mathsf{bc}}$.

4. Computation for γ_{abc}:
   - Using γ_{ab} (Step 1), λ_{c}, compute the following:

$$P_0, P_1 : \gamma^1_{\mathsf{abc}} = \gamma^1_{\mathsf{ab}}\lambda^3_{\mathsf{c}} + \gamma^3_{\mathsf{ab}}\lambda^1_{\mathsf{c}} + \gamma^3_{\mathsf{ab}}\lambda^3_{\mathsf{c}}$$
$$P_0, P_2 : \gamma^2_{\mathsf{abc}} = \gamma^2_{\mathsf{ab}}\lambda^3_{\mathsf{c}} + \gamma^3_{\mathsf{ab}}\lambda^2_{\mathsf{c}} + \gamma^2_{\mathsf{ab}}\lambda^2_{\mathsf{c}}$$
$$P_0, P_3 : \gamma^3_{\mathsf{abc}} = \gamma^1_{\mathsf{ab}}\lambda^2_{\mathsf{c}} + \gamma^2_{\mathsf{ab}}\lambda^1_{\mathsf{c}} + \gamma^1_{\mathsf{ab}}\lambda^1_{\mathsf{c}}$$

   - P_0, P_3 and P_j sample random $\mathsf{u}^j_{\mathsf{abc}} \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1_{\mathsf{abc}} + \mathsf{u}^2_{\mathsf{abc}} = \gamma^3_{\mathsf{abc}} + \mathsf{r}$ for $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.
   - P_0, P_1, P_2 sample random $\mathsf{s} \in_R \mathbb{Z}_{2^\ell}$. P_0 sends $\mathsf{w}_{\mathsf{abc}} = \gamma^1_{\mathsf{abc}} + \gamma^2_{\mathsf{abc}} + \mathsf{s}$ to P_3.

5. P_0, P_3 compute $\mathsf{r} = \mathsf{u}^1_{\mathsf{abc}} + \mathsf{u}^2_{\mathsf{abc}} - \gamma^3_{\mathsf{abc}}$ and set $\mathsf{q} = \mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else set $\mathsf{q} = \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

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

1. Parties locally compute the following:

$$\begin{aligned} P_1 : \mathsf{y}_1 &= -\lambda^1_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}} - \lambda^1_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}} - \lambda^1_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}} + \gamma^1_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}} + (\gamma^1_{\mathsf{ac}} + \mathsf{u}^1_{\mathsf{ac}})\mathsf{m}_{\mathsf{b}} + (\gamma^1_{\mathsf{bc}} + \mathsf{u}^1_{\mathsf{bc}})\mathsf{m}_{\mathsf{a}} - (\gamma^1_{\mathsf{abc}} + \mathsf{u}^1_{\mathsf{abc}}) \\ P_2 : \mathsf{y}_2 &= -\lambda^2_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}} - \lambda^2_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}} - \lambda^2_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}} + (\gamma^2_{\mathsf{ac}} + \mathsf{u}^2_{\mathsf{ac}})\mathsf{m}_{\mathsf{b}} - (\gamma^2_{\mathsf{bc}} + \mathsf{u}^2_{\mathsf{bc}})\mathsf{m}_{\mathsf{a}} - (\gamma^2_{\mathsf{abc}} + \mathsf{u}^2_{\mathsf{abc}}) \\ P_1, P_2 : \mathsf{y}_3 &= -\lambda^3_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}} - \lambda^3_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}} - \lambda^3_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}} + \gamma^3_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}} \end{aligned}$$

2. P_1 sends y_2 to P_2, while P_2 sends y_1 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m}_{\mathsf{abc}}$.
3. If $\mathsf{isTr} = 1$, P_1, P_2 locally set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.
4. Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.
5. *Verification:*
   - Parties locally compute the following:

$$\begin{aligned} P_3 : \mathsf{v} = &-(\lambda^1_{\mathsf{a}} + \lambda^2_{\mathsf{a}})\mathsf{m}_{\mathsf{bc}} - (\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{b}})\mathsf{m}_{\mathsf{ac}} - (\lambda^1_{\mathsf{c}} + \lambda^2_{\mathsf{c}})\mathsf{m}_{\mathsf{ab}} + (\gamma^1_{\mathsf{ab}} + \gamma^2_{\mathsf{ab}})\mathsf{m}_{\mathsf{c}} + (\mathsf{w}_{\mathsf{ac}} + \gamma^3_{\mathsf{ac}})\mathsf{m}_{\mathsf{b}} \\ &+ (\mathsf{w}_{\mathsf{bc}} + \gamma^3_{\mathsf{bc}})\mathsf{m}_{\mathsf{a}} - (\mathsf{w}_{\mathsf{abc}} + \gamma^3_{\mathsf{abc}} + \mathsf{r}) \\ P_1, P_2 : \mathsf{v}' = &\ \mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s}_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}} + \mathsf{s}_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}} - \mathsf{s} \end{aligned}$$

   - P_3 sends $\mathsf{H}(\mathsf{v})$ to P_1, P_2, who `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{v}')$.

Figure 5.9: 3-input fair multiplication in Tetrad.

Lemma 5.6 (Communication) *Protocol Π_{Mult3} (Fig. 5.9) (in Tetrad) requires 9ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Proof: In the preprocessing, computation of γ_{ab} involves three instances of jsnd. Each of the computation of $\gamma_{\mathsf{ac}}, \gamma_{\mathsf{bc}}$ involves one instance of jsnd and a communication from P_0 to P_3. The computation of γ_{abc} is similar to the preprocessing of fair multiplication protocol (Fig. 5.3). The communication pattern of the online phase is similar to that of the fair multiplication protocol. The costs follow from Lemma 5.3 and Lemma 5.1. □

Analogously, $\Pi^{\mathsf{R}}_{\mathsf{Mult3}}$ can be extended to support 3-input multiplication while costing 12 elements communication in preprocessing. The protocol appears in Fig. 5.10.

Protocol $\Pi^{\mathsf{R}}_{\mathsf{Mult3}}(\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr}=1$) or not ($\mathsf{isTr}=0$).
Input(s): $[\![\mathsf{a}]\!],[\![\mathsf{b}]\!],[\![\mathsf{c}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o}=\mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr}=1$ and $\mathsf{o}=\mathsf{z}$ if $\mathsf{isTr}=0$ and $\mathsf{z}=\mathsf{abc}$.

Preprocessing:

1. Computation for γ_{ab}: Invoke Π_{MultS} (Fig. 5.4) on $\langle\lambda^{\mathsf{R}}_{\mathsf{a}}\rangle$ and $\langle\lambda^{\mathsf{R}}_{\mathsf{b}}\rangle$ to generate $\langle\gamma_{\mathsf{ab}}\rangle$.
2. Computation for $\gamma_{\mathsf{ac}},\gamma_{\mathsf{bc}}$: Similar to Step 1 (for γ_{ab}).
3. Computation for γ_{abc}:
   - Using γ_{ab} (Step 1), λ_{c}, compute the following:

$$\begin{aligned}
P_0,P_1 &: \gamma^1_{\mathsf{abc}}=\gamma^1_{\mathsf{ab}}\lambda^3_{\mathsf{c}}+\gamma^3_{\mathsf{ab}}\lambda^1_{\mathsf{c}}+\gamma^3_{\mathsf{ab}}\lambda^3_{\mathsf{c}}\\
P_0,P_2 &: \gamma^2_{\mathsf{abc}}=\gamma^2_{\mathsf{ab}}\lambda^3_{\mathsf{c}}+\gamma^3_{\mathsf{ab}}\lambda^2_{\mathsf{c}}+\gamma^2_{\mathsf{ab}}\lambda^2_{\mathsf{c}}\\
P_0,P_3 &: \gamma^3_{\mathsf{abc}}=\gamma^1_{\mathsf{ab}}\lambda^2_{\mathsf{c}}+\gamma^2_{\mathsf{ab}}\lambda^1_{\mathsf{c}}+\gamma^1_{\mathsf{ab}}\lambda^1_{\mathsf{c}}
\end{aligned}$$

   - P_0,P_3 and P_j sample random $\mathsf{u}^j_{\mathsf{abc}}\in_R\mathbb{Z}_{2^\ell}$ for $j\in\{1,2\}$. Let $\mathsf{u}^1_{\mathsf{abc}}+\mathsf{u}^2_{\mathsf{abc}}=\gamma^3_{\mathsf{abc}}+\mathsf{r}$ for $\mathsf{r}\in_R\mathbb{Z}_{2^\ell}$.
   - P_0,P_1,P_2 sample random $\mathsf{s}_1,\mathsf{s}_2\in_R\mathbb{Z}_{2^\ell}$. P_0,P_j jsnd $\mathsf{w}^j=\gamma^j_{\mathsf{abc}}+\mathsf{s}_j$ to P_3 for $j\in\{1,2\}$.
4. P_0,P_3 compute $\mathsf{r}=\mathsf{u}^1_{\mathsf{abc}}+\mathsf{u}^2_{\mathsf{abc}}-\gamma^3_{\mathsf{abc}}$ and set $\mathsf{q}=\mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr}=1$, else set $\mathsf{q}=\mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_0,P_3,\mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

Online: Let $\mathsf{y}=(\mathsf{z}-\mathsf{r})-\mathsf{m}_{\mathsf{abc}}-\mathsf{s}_1-\mathsf{s}_2$.

1. Parties locally compute the following:

$$\begin{aligned}
P_0,P_1 &: \mathsf{y}_1=-\lambda^1_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}}-\lambda^1_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}}-\lambda^1_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}}+\gamma^1_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}}+\gamma^1_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}}+\gamma^1_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}}-(\mathsf{u}^1_{\mathsf{abc}}+\mathsf{w}^1)\\
P_0,P_2 &: \mathsf{y}_2=-\lambda^2_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}}-\lambda^2_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}}-\lambda^2_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}}+\gamma^2_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}}+\gamma^2_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}}+\gamma^2_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}}-(\mathsf{u}^2_{\mathsf{abc}}+\mathsf{w}^2)\\
P_1,P_2 &: \mathsf{y}_3=-\lambda^3_{\mathsf{a}}\mathsf{m}_{\mathsf{bc}}-\lambda^3_{\mathsf{b}}\mathsf{m}_{\mathsf{ac}}-\lambda^3_{\mathsf{c}}\mathsf{m}_{\mathsf{ab}}+\gamma^3_{\mathsf{ab}}\mathsf{m}_{\mathsf{c}}+\gamma^3_{\mathsf{ac}}\mathsf{m}_{\mathsf{b}}+\gamma^3_{\mathsf{bc}}\mathsf{m}_{\mathsf{a}}
\end{aligned}$$

2. P_1,P_3 jsnd y_1 to P_2, while P_2,P_3 jsnd y_2 to P_1. P_1,P_2 locally compute $\mathsf{z}-\mathsf{r}=(\mathsf{y}_1+\mathsf{y}_2+\mathsf{y}_3)+\mathsf{m}_{\mathsf{abc}}+\mathsf{s}_1+\mathsf{s}_2$.
3. If $\mathsf{isTr}=1$, P_1,P_2 set $\mathsf{p}=(\mathsf{z}-\mathsf{r})^{\mathsf{t}}$, else $\mathsf{p}=\mathsf{z}-\mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1,P_2,\mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.
4. Parties locally compute $[\![\mathsf{o}]\!]=[\![\mathsf{p}]\!]+[\![\mathsf{q}]\!]$. Here $\mathsf{o}=\mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr}=1$ and z otherwise.

Figure 5.10: 3-input robust multiplication in $\mathsf{Tetrad\text{-}R^{II}}$.

Lemma 5.7 (Communication) *Protocol* $\Pi^{\mathsf{R}}_{\mathsf{Mult3}}$ *(Fig. 5.10) (in* Tetrad-R^{II}*) requires* 12ℓ *bits of communication in preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: In the preprocessing, computation of each of $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{ac}}, \gamma_{\mathsf{bc}}$ involves three instances of jsnd. The computation of γ_{abc} is similar to the preprocessing of robust multiplication protocol (Fig. 5.8). The communication pattern of the online phase is similar to that of the robust multiplication protocol. The costs follow from Lemma 5.5 and Lemma 5.1. □

To obtain $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{abcd}$ given the $[\![\cdot]\!]$-sharing of $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$, we can write $\mathsf{z} - \mathsf{r}$ as

$$\begin{aligned}\mathsf{z} - \mathsf{r} = \mathsf{abcd} - \mathsf{r} &= (\mathsf{m_a} - \lambda_{\mathsf{a}})(\mathsf{m_b} - \lambda_{\mathsf{b}})(\mathsf{m_c} - \lambda_{\mathsf{c}})(\mathsf{m_d} - \lambda_{\mathsf{d}}) - \mathsf{r} \\ &= \mathsf{m_{abcd}} - \mathsf{m_{abc}}\lambda_{\mathsf{d}} - \mathsf{m_{abd}}\lambda_{\mathsf{c}} - \mathsf{m_{acd}}\lambda_{\mathsf{b}} - \mathsf{m_{bcd}}\lambda_{\mathsf{a}} + \mathsf{m_{ab}}\gamma_{\mathsf{cd}} + \mathsf{m_{ac}}\gamma_{\mathsf{bd}} + \mathsf{m_{ad}}\gamma_{\mathsf{bc}} + \mathsf{m_{bc}}\gamma_{\mathsf{ad}} \\ &\quad + \mathsf{m_{bd}}\gamma_{\mathsf{ac}} + \mathsf{m_{cd}}\gamma_{\mathsf{ab}} - \mathsf{m_a}\gamma_{\mathsf{bcd}} - \mathsf{m_b}\gamma_{\mathsf{acd}} - \mathsf{m_c}\gamma_{\mathsf{abd}} - \mathsf{m_d}\gamma_{\mathsf{abc}} + \gamma_{\mathsf{abcd}} - \mathsf{r} \quad \text{(cf. notation 5.1)}\end{aligned} \tag{5.4}$$

While the online phase proceeds similarly to the 2 and 3-input multiplication, in the preprocessing phase, the parties need to generate the additive shares of γ_{ab},γ_{ac},γ_{ad},γ_{bc},γ_{bd},γ_{cd},γ_{abc},γ_{abd},γ_{acd},γ_{bcd} and $\gamma_{\mathsf{abcd}} - \mathsf{r}$. This is computed similarly as in the case of 3-input multiplication as follows. Parties generate shares of $\gamma_{\mathsf{ac}}, \gamma_{\mathsf{ad}}, \gamma_{\mathsf{bc}}, \gamma_{\mathsf{bd}}$ similar to the generation of shares of γ_{ac} in the 3-input multiplication. For $\gamma_{\mathsf{ab}}, \gamma_{\mathsf{cd}}$, parties proceed similar to generation of shares of γ_{ab} in the 3-input multiplication, where the respective $\langle\cdot\rangle$-shares are generated. This is followed by generation of shares of $\gamma_{\mathsf{abc}}, \gamma_{\mathsf{abd}}, \gamma_{\mathsf{acd}}, \gamma_{\mathsf{bcd}}, \gamma_{\mathsf{abcd}}$ following steps similar to the ones involved in generating γ_{abcc} in the 3-input multiplication. Since the protocol is very similar to the 3-input protocol, we omit the formal details.

5.2.6 Supporting on-demand computations

For on-demand applications where the underlying function to be computed is not known in advance, the preprocessing model is not desirable. We observe that the Tetrad protocol can be modified by executing the preprocessing phase in the online phase itself, keeping the same overall communication cost. The formal protocol appears in Fig. 5.11.

We provide the fair multiplication, $\Pi^{\mathsf{NoPre}}_{\mathsf{Mult}}$, for *function-independent* preprocessing in Fig. 5.11. The protocol incurs no overhead over the fair multiplication (Π_{Mult}) in Tetrad. This is due to the design of Π_{Mult} where values $\mathsf{u}^1, \mathsf{u}^2$ are sampled non-interactively in the preprocessing. Thus the joint-sharing by P_0, P_3 (Step 5 (a) in Fig. 5.11) can be performed along with the communication among P_1, P_2 (Step 4 in Fig. 5.11) in the online. Moreover, the rest of the communication can

be deferred till the verification stage and thus, the online round complexity is retained. The protocol for robust setting is similar.

Protocol $\Pi_{\mathsf{Mult}}^{\mathsf{NoPre}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{ab}$.

Online:

1. Parties locally compute the following:

$$\begin{aligned} P_0, P_1 &: \gamma_{\mathsf{ab}}^1 = \lambda_{\mathsf{a}}^1\lambda_{\mathsf{b}}^3 + \lambda_{\mathsf{a}}^3\lambda_{\mathsf{b}}^1 + \lambda_{\mathsf{a}}^3\lambda_{\mathsf{b}}^3 \\ P_0, P_2 &: \gamma_{\mathsf{ab}}^2 = \lambda_{\mathsf{a}}^2\lambda_{\mathsf{b}}^3 + \lambda_{\mathsf{a}}^3\lambda_{\mathsf{b}}^2 + \lambda_{\mathsf{a}}^2\lambda_{\mathsf{b}}^2 \\ P_0, P_3 &: \gamma_{\mathsf{ab}}^3 = \lambda_{\mathsf{a}}^1\lambda_{\mathsf{b}}^2 + \lambda_{\mathsf{a}}^2\lambda_{\mathsf{b}}^1 + \lambda_{\mathsf{a}}^1\lambda_{\mathsf{b}}^1 \end{aligned}$$

2. P_0, P_3 and P_j sample random $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma_{\mathsf{ab}}^3 - \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.

3. Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \mathsf{m_a}\mathsf{m_b}$. Parties locally compute the following:

$$\begin{aligned} P_1 &: \mathsf{y}_1 = -\lambda_{\mathsf{a}}^1\mathsf{m_b} - \lambda_{\mathsf{b}}^1\mathsf{m_a} + \gamma_{\mathsf{ab}}^1 + \mathsf{u}^1 \\ P_2 &: \mathsf{y}_2 = -\lambda_{\mathsf{a}}^2\mathsf{m_b} - \lambda_{\mathsf{b}}^2\mathsf{m_a} + \gamma_{\mathsf{ab}}^2 + \mathsf{u}^2 \\ P_1, P_2 &: \mathsf{y}_3 = -\lambda_{\mathsf{a}}^3\mathsf{m_b} - \lambda_{\mathsf{b}}^3\mathsf{m_a} \end{aligned}$$

4. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1.

5. Parties proceed as follows:

   (a) P_0, P_3: $\mathsf{r} = \gamma_{\mathsf{ab}}^3 - \mathsf{u}^1 - \mathsf{u}^2$; $\mathsf{q} = \mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else $\mathsf{q} = \mathsf{r}$; Execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$.

   (b) P_1, P_2: $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \mathsf{m_a}\mathsf{m_b}$; $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$; Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$.

6. Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Verification:

1. P_0, P_1, P_2 sample random $\mathsf{s} \in_R \mathbb{Z}_{2^\ell}$. P_0 sends $\mathsf{w} = \gamma_{\mathsf{ab}}^1 + \gamma_{\mathsf{ab}}^2 + \mathsf{s}$ to P_3.

2. P_3 computes $\mathsf{v} = -(\lambda_{\mathsf{a}}^1 + \lambda_{\mathsf{a}}^2)\mathsf{m}_{\mathsf{b}} - (\lambda_{\mathsf{b}}^1 + \lambda_{\mathsf{b}}^2)\mathsf{m}_{\mathsf{a}} + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ and sends $\mathsf{H}(\mathsf{v})$ to P_1 and P_2. Parties P_1, P_2 `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s})$.

Figure 5.11: Fair multiplication without preprocessing in Tetrad.

5.2.6.1 Comparison with Fantastic Four [46]

We analyse the performance of Fantastic Four [46] where execution proceeds in segments (cf. §6.4, [46]). Elaborately, computation is carried out optimistically for each segment, followed by a verification phase before proceeding to the next segment. If verification fails, the current segment is recomputed via an active 3PC protocol. Subsequent segments also proceed with a 3PC execution until the verification fails again. In this case, a semi-honest 2PC with a helper is carried out for the current and rest of the segments. For analysis, we consider their best and worst-case execution cost.

Work	Dot Product w/ Truncation		#Active Parties
	Preprocessing	Online	
Fantastic Four: Case I	ℓ	9ℓ	4
Fantastic Four: Case II	$76(\ell+\kappa)+54x+12$	$9\ell+6\kappa$	3
Tetrad-R$^{\mathsf{I}}$(on-demand)	-	5ℓ	3
Tetrad-R$^{\mathsf{II}}$(on-demand)	-	6ℓ	3

Table 5.3: Comparison with Fantastic Four [46]

Observe that the best case happens when the verification is always successful, which we call as *Case I*. In this case, the communication cost is that of the 4PC execution. Note that an adversary can *always* make the verification fail in the first segment itself. This results in executing the entire protocol (all segments) with their active 3PC, which accounts for their worst-case cost. We denote this as *Case II*. Their 3PC protocols are designed to work over the extended ring of size $\ell+\kappa$ bits. As evident from Tables 2, 3 of their paper, their 3PC is at least $10\times$ more expensive than their 4PC in terms of both runtime and communication. Thus, the higher cost of 3PC defeats the purpose of having an additional honest party in the system.

Observe that their protocols are designed to work with a function-independent preprocessing. Thus, for a fair comparison, we compare both cases against the on-demand variants of our robust protocols (Tetrad-R$^{\mathsf{I}}$, Tetrad-R$^{\mathsf{II}}$). The results are summarised in Table 5.3. We remark that the values for their cases are obtained from Table 1 of their paper [46].

5.3 Garbled World

In the applications we consider, the garbled circuit is used as an intermediary to evaluate certain functions where the input to the function as well as the output are in $[\![\cdot]\!]$-shared (or $[\![\cdot]\!]^{\mathbf{B}}$-shared) form.

Instantiating the garbled world using existing 4PC GC-based protocols [72, 30] turn out to be overkill, as they are standalone protocols. For instance, [72] provides robust protocols by communicating 12 GCs while [30] requires generating and exchanging commitments on the inputs to ensure input consistency. On the other hand, the inputs to our protocol are consistent (due to $[\![\cdot]\!]$-sharing), and we do not need an explicit reconstruction, making it lighter overall.

Towards this, we propose 2 GC protocols – $\mathsf{Tetrad}_{\mathsf{T}}$ requiring communication of 2 GC evaluations and 1 online round, and $\mathsf{Tetrad}_{\mathsf{C}}$ requiring 1 GC and 2 rounds. Moreover, these protocols leverage the benefit of amortization which comes from using jsnd. The 2 GC variant has two parallel executions, each comprising of 3 garblers and 1 evaluator. P_1, P_2 act as evaluators in two independent executions and the parties in $\Phi_1 = \{P_0, P_2, P_3\}$, $\Phi_2 = \{P_0, P_1, P_3\}$ act as garblers, respectively. The 1 GC variant comprises of a single execution with Φ_1 acting as garblers and P_1 as the evaluator. Leveraging an honest majority among the garblers and using jsnd, we only need semi-honest GC computation to get active security.

5.3.1 2 GC Variant

Input Phase. Given that the function input x is already available as $[\![\mathsf{x}]\!]^{\mathbf{B}}$, the boolean values $\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}$, where $\alpha_\mathsf{x} = \lambda^1_\mathsf{x} \oplus \lambda^2_\mathsf{x}$ and $\mathsf{x} = \mathsf{m}_\mathsf{x} \oplus \alpha_\mathsf{x} \oplus \lambda^3_\mathsf{x}$, act as the *new* inputs for the garbled computation, and garbled sharing ($[\![\cdot]\!]^{\mathbf{G}}$) is generated for each of these values. The semantics of $[\![\cdot]\!]^{\mathbf{B}}$-sharing ensures that each of these shares $(\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x})$ is available with two garblers in each garbling instance. The keys for the shares can either be sent (using jsnd) correctly to the evaluators or the inconsistency is detected. This key delivery essentially generates $[\![\cdot]\!]^{\mathbf{G}}$-sharing for each of these three values which enables GC evaluation. Thus, the goal of our input phase is to create the compound sharing, $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m}_\mathsf{x}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{x}]\!]^{\mathbf{G}})$ for every input x to the function to be evaluated via the GC. We first discuss the semantics for $[\![\cdot]\!]^{\mathbf{G}}$-sharing followed by steps for generating $[\![\cdot]\!]^{\mathbf{C}}$-sharing.

Garbled sharing semantics. A value $\mathsf{v} \in \mathbb{Z}_2$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared (garbled shared) amongst $\mathcal{P}$ if $P_i \in \{P_0, P_3\}$ holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_i = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, P_1 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$ and P_2 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$. Here, $\mathsf{K}^{\mathsf{v},j}_\mathsf{v} = \mathsf{K}^{0,j}_\mathsf{v} \oplus \mathsf{v}\Delta^j$ for $j \in \{1, 2\}$, and Δ^j, which is known only to the garblers

in Φ_j, denotes the global offset with its least significant bit set to 1 and is same for every wire in the circuit. A value $\mathsf{x} \in \mathbb{Z}_2$ is said to be $[\![\cdot]\!]^{\mathbf{C}}$-shared (compound shared) if each value from $(\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x})$, which are as defined above, is $[\![\cdot]\!]^{\mathbf{G}}$-shared. We write $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\mathsf{m}_\mathsf{x}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{x}]\!]^{\mathbf{G}})$.

Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ and $[\![\mathsf{x}]\!]^{\mathbf{C}}$ Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(\mathcal{P}, \mathsf{v})$ (Fig. 5.12) enables generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ where two garblers in each garbling instance hold v, and proceeds as follows. Consider the first garbling instance with evaluator P_1 where garblers P_k, P_l hold v. Garblers in Φ_1 generate $\{\mathsf{K}^{\mathsf{b},1}_\mathsf{v}\}_{\mathsf{b}\in\{0,1\}}$ which denotes the key for value b on wire v, following the free-XOR technique [82, 84]. P_k, P_l jsnd $\mathsf{K}^{\mathsf{v},1}_\mathsf{v}$ to evaluator P_1. Similar steps carried out with respect to the second garbling instance, at the end of which, garblers in Φ_2 possess $\{\mathsf{K}^{\mathsf{b},2}_\mathsf{v}\}_{\mathsf{b}\in\{0,1\}}$ while the evaluator P_2 holds $\mathsf{K}^{\mathsf{v},2}_\mathsf{v}$. Following this, the shares $[\![\mathsf{v}]\!]^{\mathbf{G}}_s$ held by $P_s \in \mathcal{P}$ are defined as $[\![\mathsf{v}]\!]^{\mathbf{G}}_0 = [\![\mathsf{v}]\!]^{\mathbf{G}}_3 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$.

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

1. Garblers in Φ_j for $j \in \{1, 2\}$ generate keys $\mathsf{K}^{0,j}_\mathsf{v}, \mathsf{K}^{1,j}_\mathsf{v}$ for wire v, using free-XOR technique.
2. Let P^j_k, P^j_l denote the garblers in the j^{th} garbling instance, for $j \in \{1, 2\}$, who hold $\mathsf{v} \in \mathbb{Z}_2$. P^j_k, P^j_l jsnd $\mathsf{K}^{\mathsf{v},j}_\mathsf{v}$ to evaluator P_j.
3. $P_i \in \{P_0, P_3\}$ sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_i = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$, P_1 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = (\mathsf{K}^{\mathsf{v},1}_\mathsf{v}, \mathsf{K}^{0,2}_\mathsf{v})$ and P_2 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = (\mathsf{K}^{0,1}_\mathsf{v}, \mathsf{K}^{\mathsf{v},2}_\mathsf{v})$.

Figure 5.12: Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$

To generate $[\![\mathsf{x}]\!]^{\mathbf{C}}$, we need a way to generate $([\![\mathsf{m}_\mathsf{x}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{x}]\!]^{\mathbf{G}})$, given $[\![\mathsf{x}]\!]^{\mathbf{B}}$. For this, $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$ is invoked for each of $\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}$.

Evaluation. Let $f(\mathsf{x})$ be the function to be evaluated. At this point, the function input is $[\![\cdot]\!]^{\mathbf{C}}$-shared. This renders $[\![\cdot]\!]^{\mathbf{G}}$-sharing for the input of the GC that corresponds to the function $f'(\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x})$ which first combines the given boolean-shares to compute the actual input and then applies f on it. Let GC_j denote the garbled circuit to be sent to $P_j \in \{P_1, P_2\}$ by garblers in Φ_j. Sending of GC_j is overlapped with the key transfer (during generation of $[\![\mathsf{x}]\!]^{\mathbf{C}}$), to save rounds, where garblers in $\{P_0, P_3\}$ jsnd GC_j to P_j. On receiving the GC, evaluators evaluate their respective GCs and obtain the key corresponding to the output, say z. This generates $[\![\mathsf{z}]\!]^{\mathbf{G}}$.

Output phase. The goal of output computation is to compute the output z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$. To reconstruct z towards $P_j \in \{P_1, P_2\}$, two garblers in Φ_j send the least significant bit p^j of $\mathsf{K}^{0,j}_\mathsf{z}$, referred to as the decoding information, to P_j. If the received values are consistent, P_j uses the received p^j to reconstruct z as $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$, where q^j denotes the least significant bit

of $\mathsf{K}_{\mathsf{z}}^{\mathsf{z},j}$; else P_j aborts. To reconstruct z towards the garblers $P_g \in \{P_0, P_3\}$, one evaluator, say P_1 sends the least significant bit, q^1, of $\mathsf{K}_{\mathsf{z}}^{\mathsf{z},1}$ along with $\mathcal{H} = \mathsf{H}(\mathsf{K}_{\mathsf{z}}^{\mathsf{z},1})$ to P_g, where H is a collision-resistant hash function. If a garbler received a consistent $(\mathsf{q}^1, \mathcal{H})$ pair from P_1 such that there exists a $K \in \{\mathsf{K}_{\mathsf{z}}^{0,1}, \mathsf{K}_{\mathsf{z}}^{1,1}\}$ whose least significant bit is q^1 and $\mathsf{H}(K) = \mathcal{H}$, then it uses q^1 for reconstructing z; else the garbler aborts the computation. Note that a corrupt evaluator P_1 cannot create confusion among garblers in $\{P_0, P_3\}$ by sending the key that was not output by the GC owing to the authenticity of the garbling scheme. Reconstruction is lightweight and requires a single round for garblers while reconstruction towards evaluators can be overlapped with key transfer and does not incur extra rounds. The protocol appears in Fig. 5.13.

Protocol $\Pi_{\mathsf{Rec}}^{\mathbf{G}}(\mathcal{P}, [\![\mathsf{z}]\!]^{\mathbf{G}})$

- For an output wire z, let p^j denote the least significant bit of $\mathsf{K}_{\mathsf{z}}^{0,j}$ and q^j denote the least significant bit of $\mathsf{K}_{\mathsf{z}}^{\mathsf{z},j}$ for $j \in \{1, 2\}$.

- *Reconstruction towards* $P_j \in \{P_1, P_2\}$: Garblers P_0, P_3 in Φ_j jsnd p^j to P_j. If P_j received consistent values from P_0, P_3, it reconstructs z as $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$.

- *Reconstruction towards* $P_g \in \{P_0, P_3\}$: P_1 sends q^1 and $\mathcal{H} = \mathsf{H}(\mathsf{K}_{\mathsf{z}}^{\mathsf{z},1})$ to P_g, where H is a collision-resistant hash function. P_g uses the q^1 received from P_1 for reconstructing z as $\mathsf{z} = \mathsf{p}^1 \oplus \mathsf{q}^1$ if there exists a $K \in \{\mathsf{K}_{\mathsf{z}}^{0,1}, \mathsf{K}_{\mathsf{z}}^{1,1}\}$ whose least significant bit is q^1 and $\mathsf{H}(K) = \mathcal{H}$.

Figure 5.13: Output computation: reconstruction of z

Optimizations when deployed in mixed framework. Working in the preprocessing model enables transfer of the (communication-intensive) GC and generating $[\![\cdot]\!]^{\mathbf{G}}$-shares of the input-independent shares of x (i.e. $\alpha_{\mathsf{x}}, \lambda_{\mathsf{x}}^3$) in the preprocessing phase. Thus, the online phase is very light and only requires one round to generate $[\![\cdot]\!]^{\mathbf{G}}$-shares for the input-dependent data (i.e. m_{x}). Since evaluation is local, evaluators obtain $[\![\cdot]\!]^{\mathbf{G}}$-sharing of the GC output at the end of 1 round.

Achieving fairness and robustness. To ensure fairness, we require a fair reconstruction protocol that proceeds as follows. As described in §5.2.3.1, parties first ensure that all parties are alive. If so, they proceed similar to the protocol in Fig. 5.13, except with the following differences. For reconstruction towards evaluators, *all* three respective garblers send it the decoding information. The evaluator selects the value appearing in the majority for reconstruction. For reconstruction towards garblers P_0, P_3, *both* the evaluators send the least significant bit of the output key together with its hash to the garbler. The presence of at least one honest evaluator guarantees that both garblers will be on the same page. The protocol appears in Fig. 5.14.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{fRec}}(\mathcal{P}, [\![\mathsf{z}]\!]^{\mathbf{G}})$

- Parties perform a bit exchange as described in §5.2.3.1 to ensure that all parties are alive. If all parties are alive, they proceed as follows.

- For an output wire z, let p^j denote the least significant bit of $\mathsf{K}^{0,j}_{\mathsf{z}}$ and q^j denote the least significant bit of $\mathsf{K}^{\mathsf{z},j}_{\mathsf{z}}$ for $j \in \{1, 2\}$.

- *Reconstruction towards* $P_j \in \{P_1, P_2\}$: Garblers in Φ_j send p^j to P_j. P_j selects the value forming majority among these and reconstructs z as $\mathsf{z} = \mathsf{p}^j \oplus \mathsf{q}^j$.

- *Reconstruction towards* $P_g \in \{P_0, P_3\}$: $P_j \in \{P_1, P_2\}$ sends q^j and $\mathcal{H}^j = \mathsf{H}(\mathsf{K}^{\mathsf{z},j}_{\mathsf{z}})$ to P_g, where H is a collision-resistant hash function. P_g uses the q^1 received from P_1 for reconstructing z as $\mathsf{z} = \mathsf{p}^1 \oplus \mathsf{q}^1$ if there exists a $K \in \{\mathsf{K}^{0,1}_{\mathsf{z}}, \mathsf{K}^{1,1}_{\mathsf{z}}\}$ whose least significant bit is q^1 and $\mathsf{H}(K) = \mathcal{H}^1$. Else, it computes $\mathsf{z} = \mathsf{p}^2 \oplus \mathsf{q}^2$.

Figure 5.14: Fair output computation: fair reconstruction of z

The main difference from its fair counterpart is the use of a robust jsnd primitive to achieve robustness. This guarantees that a TTP is identified if misbehaviour is detected, taking the computation to completion and delivering the output to all.

5.3.2 1 GC Variant

The input $\mathsf{x} = \mathsf{x}_1 \oplus \mathsf{x}_2$ for this variant consists of two shares, $\mathsf{x}_1 = \mathsf{m}_{\mathsf{x}} \oplus \lambda^2_{\mathsf{x}}$ and $\mathsf{x}_2 = \lambda^1_{\mathsf{x}} \oplus \lambda^3_{\mathsf{x}}$, where $\mathsf{m}_{\mathsf{x}}, \lambda^1_{\mathsf{x}}, \lambda^2_{\mathsf{x}}, \lambda^3_{\mathsf{x}}$ are as defined in $[\![\mathsf{x}]\!]^{\mathbf{B}}$. To ensure correct key transfer for the value x_2 held by garbler P_0 and evaluator P_1, garblers P_0, P_3 commit to both keys for x_2 towards P_1, while P_0 sends the opening to the key for x_2. Then, P_1 verifies the consistency of the received commitments and the opening, as it possesses x_2. The protocol appears in Fig. 5.15.

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

1. Garblers in Φ_1 generate keys $\mathsf{K}^0_{\mathsf{v}}, \mathsf{K}^1_{\mathsf{v}}$ using free-XOR technique.

2. If $(P_i, P_j) = (P_2, P_3)$: P_i, P_j jsnd $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$ to P_1.

3. If $(P_i, P_j) = (P_0, P_1)$:
   - P_0, P_3 compute commitments on $\mathsf{K}^0_{\mathsf{v}}, \mathsf{K}^1_{\mathsf{v}}$, and jsnd the commitment to P_1.
   - P_0 sends the opening of the commitment for $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$ to P_1.
   - P_1 verifies if the received opening information correctly decommits the commitment on $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$, where v is held by P_1. Else it `aborts`.

4. Party $P_s \in \Phi_1$ sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_s = \mathsf{K}^0_{\mathsf{v}}$, while P_1 sets $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = \mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$.

Figure 5.15: Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$

The evaluation and output phases are similar to the 2GC variant, except there is only a single garbling instance now. Looking ahead, in the mixed protocol framework, the output has to be reconstructed towards P_1, P_2. Reconstruction towards P_1 does not incur additional rounds since sending of decoding information can be overlapped with the key transfer. However, unlike in the 2GC variant where reconstruction towards P_2 can be done similar to reconstruction towards P_1, in the 1GC variant, an additional round is required as P_2 is no longer an evaluator. This incurs one extra round as opposed to the 2GC variant.

Achieving fairness. To ensure fair reconstruction, as in §5.2.3.1, parties first perform an aliveness check. Following this, they proceed towards a fair reconstruction of z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$ as follows. First, reconstruction of z is carried out towards the garblers $P_g \in \Phi_1$. For this, P_1 sends q (least significant bit of $\mathsf{K}^{\mathsf{z}}_{\mathsf{z}}$) and $\mathcal{H} = \mathsf{H}(\mathsf{K}^{\mathsf{z}}_{\mathsf{z}})$ to P_g as before. Now, if a garbler received a consistent $(\mathsf{q}, \mathcal{H})$ pair from P_1 such that there exists a $K \in \{\mathsf{K}^0_{\mathsf{z}}, \mathsf{K}^1_{\mathsf{z}}\}$ whose least significant bit is q and $\mathsf{H}(K) = \mathcal{H}$, then it uses q for reconstructing z, and sends z to its co-garblers. Else, a garbler accepts a z received from a co-garbler as the output. Thus, further dissemination of the output by garblers ensures that all parties are on the same page. If garblers receive the output, reconstruction of z is carried out towards P_1. For this, all garblers (who received the output) send the decoding information to P_1, who selects the majority value to reconstruct z.

Protocol $\Pi^{\mathbf{G}}_{\mathsf{fRec}}([\![\mathsf{z}]\!]^{\mathbf{G}})$

- Parties perform a bit exchange as described in §5.2.3.1 to ensure that all parties are alive. If all parties are alive, they proceed as follows.
- Let p, q denote the least significant bit of $\mathsf{K}^0_{\mathsf{z}}, \mathsf{K}^{\mathsf{z}}_{\mathsf{z}}$, respectively.
- *Reconstruction towards garblers* $P_g \in \Phi_1$: P_1 sends q and $\mathcal{H} = \mathsf{H}(\mathsf{K}^{\mathsf{z}}_{\mathsf{z}})$ to $P_g \in \Phi_1$, where H is a collision-resistant hash function. P_g does the following to reconstruct z.
  - If P_g received $(\mathsf{q}, \mathcal{H})$ from P_1 such that there exists a $K \in \{\mathsf{K}^0_{\mathsf{z}}, \mathsf{K}^1_{\mathsf{z}}\}$ whose least significant bit is q and $\mathsf{H}(K) = \mathcal{H}$, set $\mathsf{z} = \mathsf{p} \oplus \mathsf{q}$. P_g sends z to its co-garblers.
  - Else, if P_g did not receive a consistent $(\mathsf{q}, \mathcal{H})$-pair from P_1 but received a z from its co-garbler in the following round, then accept z as the output.
- *Reconstruction towards* P_1: If garblers obtained the output, then they send p to P_1. P_1 selects the value forming majority among these and reconstructs z as $\mathsf{z} = \mathsf{p} \oplus \mathsf{q}$.

Figure 5.16: Fair reconstruction of z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$

Achieving robustness. To attain robustness, we list below the differences from the fair protocol that must be carried out. The first difference is the use of a robust variant of jsnd. Second, in input sharing protocol, where x_1 is held by only garbler P_0, a corrupt P_0 may refrain

from providing P_1 with the correct key (sent as the opening information for the commitment). To ensure robustness, if P_1 fails to receive the correct key from P_0, we let P_1 complain to all parties about this inconsistency by sending an inconsistency bit. All parties exchange this inconsistency bit among themselves and agree on the majority value. If all parties agree on the presence of inconsistency, then P_0, P_1 are identified to be in conflict, and $\mathsf{TTP} = P_2$ is set to carry out the rest of the computation. Finally, to ensure a robust reconstruction, the following approach is taken. Observe that the fair reconstruction provides robustness as long as evaluator P_1 is honest. When none of the garblers obtains the output in the fair protocol, it is guaranteed that evaluator P_1 is corrupt. Thus, in such a scenario, all parties take P_1 to be corrupt and proceed with P_0 as the TTP.

5.4 Security proofs

Without loss of generality, we prove the security of our robust framework. The case for fairness follows similarly, and we omit its details. We provide proofs in the $\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jsnd}}$-hybrid model, where $\mathcal{F}_{\mathsf{setup}}$ (§2.5.1), $\mathcal{F}_{\mathsf{jsnd}}$ (Fig. 5.18) denote the ideal functionality for the shared-key setup and jsnd, respectively.

The strategy for simulating the computation of function f (represented by a circuit Ckt) is as follows: 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 $\mathcal{S}$ computes the input of $\mathcal{A}$, using the known keys, and sets the honest parties' inputs to be used in the simulation to 0. $\mathcal{S}$ invokes the ideal functionality $\mathcal{F}_{\text{GOD}}$ on behalf of $\mathcal{A}$ using the extracted input and obtains the output y. $\mathcal{S}$ now knows the inputs of $\mathcal{A}$ and can compute all the intermediate values for each building block. $\mathcal{S}$ proceeds with simulating each of the building blocks in the topological order. We provide the simulation for the case for corrupt P_0, P_1 and P_3. The case for corrupt P_2 is similar to that of P_1.

For modularity, we provide the simulation steps for each building block separately. Carrying out these blocks in the topological order yields the simulation for the entire computation. If a TTP is identified during the simulation, the simulator stops and returns the function output to the adversary on behalf of the TTP as per $\mathcal{F}_{\mathsf{jsnd}}$.

Ideal jsnd Functionality The ideal jsnd functionality for fairness security appears in Fig. 5.17 and that for the robust setting appears in Fig. 5.18.

Functionality $\mathcal{F}_{\mathsf{jsnd}}$ (for fair security)

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

Step 1: $\mathcal{F}_{\mathsf{jsnd}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from senders P_s for $s \in \{i, j\}$, $(\mathsf{Input}, \perp)$ from receiver P_k and fourth party P_l. While sending the inputs, the adversary is also allowed to send a special `abort` command.

Step 2: Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_l = \perp$.

Step 3: If $\mathsf{v}_i = \mathsf{v}_j$, set $\mathsf{msg}_k = \mathsf{v}_i$. Else, set $\mathsf{msg}_k = \texttt{abort}$.

Step 4: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{0, 1, 2, 3\}$.

Figure 5.17: Ideal functionality for jsnd in Tetrad

Functionality $\mathcal{F}_{\mathsf{jsnd}}$ (for robust security)

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

Step 1: $\mathcal{F}_{\mathsf{jsnd}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from senders P_s for $s \in \{i, j\}$, $(\mathsf{Input}, \perp)$ from receiver P_k and fourth party P_l, while it receives $(\mathsf{select}, \mathsf{ttp})$ from $\mathcal{S}$. Here ttp is a boolean value, with a 1 indicating that $\mathsf{TTP} = P_l$ should be established.

Step 2: If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{ttp} = 0$, or if $\mathcal{S}$ has corrupted P_l[a], set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_l = \perp, \mathsf{msg}_k = \mathsf{v}_i$ and go to **Step 4**.

Step 3: Else, set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{msg}_l = P_l$.

Step 4: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s for $s \in \{0, 1, 2, 3\}$.

[a] This condition is used to capture the fact that a corrupt P_l cannot create an inconsistency in $\mathcal{F}_{\mathsf{jsnd}}$ since the parties actively involved in $\mathcal{F}_{\mathsf{jsnd}}$ would be honest

Figure 5.18: Ideal functionality for robust jsnd in Tetrad.

Sharing Protocol (Π_{Sh}, Fig. 5.2) During the preprocessing, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the respective keys to $\mathcal{A}$. The values commonly held with $\mathcal{A}$ are sampled using the respective keys, while others are sampled randomly. The details for the online phase are provided next. We omit the simulation for corrupt P_3 as it is similar to that of P_1, P_2.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$

Online:

– If dealer is $\mathcal{A}$, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ receives $\mathsf{m_v}$ from $\mathcal{A}$ on behalf of P_1, P_2, P_3. If the received values are consistent, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ computes $\mathcal{A}$'s input v as $\mathsf{v} = \mathsf{m_v} - [\lambda_\mathsf{v}]_1 - [\lambda_\mathsf{v}]_2 - [\lambda_\mathsf{v}]_3$, else sets v as the default value. It invokes $\mathcal{F}_{\mathrm{GOD}}$ on input $(\mathsf{Input}, \mathsf{v})$ to obtain the function output y.

– If dealer is P_1, P_2 or P_3, nothing to simulate as P_0 doesn't receive any value during the protocol.

Figure 5.19: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ for corrupt P_0

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$

Online:

– If dealer is $\mathcal{A}$, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ receives $\mathsf{m_v}$ from $\mathcal{A}$ on behalf of P_2, P_3. If the received values are consistent, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ computes $\mathcal{A}$'s input v as $\mathsf{v} = \mathsf{m_v} - [\lambda_\mathsf{v}]_1 - [\lambda_\mathsf{v}]_2 - [\lambda_\mathsf{v}]_3$, else sets v as the default value. It invokes $\mathcal{F}_{\text{GOD}}$ on input $(\mathsf{Input}, \mathsf{v})$ to obtain the function output y.

– If dealer is P_0, P_2 or P_3, $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ sets $\mathsf{v} = 0$ and performs the protocol steps honestly.

Figure 5.20: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Sh}}}$ for corrupt P_1

Shares unknown to $\mathcal{A}$ are sampled randomly in the simulation, whereas in the real protocol, they are sampled using the pseudorandom function (PRF). The indistinguishability of the simulation thus follows by a reduction to the security of the PRF. The same holds for the rest of the blocks.

The simulation for the joint sharing protocol (Π_{JSh}) is similar to that of the sharing protocol. The protocol's design is such that the simulator will always know the value to be sent as part of the joint sharing protocol. The communication is constituted by jsnd calls and is emulated according to the simulation of $\mathcal{F}_{\mathsf{jsnd}}$.

Multiplication Protocol (Π_{Mult} in Tetrad-R$^{\mathsf{II}}$)

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– Computes $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab} on behalf of P_1, P_2, P_3.

– Samples $\mathsf{u}^1, \mathsf{u}^2$ using the respective keys with $\mathcal{A}$ and computes r. The joint sharing of q is simulated as discussed earlier.

– Receives w from $\mathcal{A}$ on behalf of P_3.

– Simulating Π_{VrfyP0}: Joint sharing of $\mathsf{e_1}, \mathsf{e_2}, \mathsf{e}$ is simulated as discussed earlier. The rest of the steps are simulated honestly. This is possible since $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ knows the randomness and inputs that should be used by $\mathcal{A}$.

Online: P_0 has no communication in the online phase except the jsnd instances which are emulated by $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$.

Figure 5.21: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_0

Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– Computes $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab} on behalf of P_0, P_2, P_3.

– Samples u^1 using the respective keys with $\mathcal{A}$. Samples a random u^2 and computes r. The joint sharing of q is simulated as discussed earlier.

– Simulate the steps of Π_{VrfyP0} honestly.

Online:

– Computes $\mathsf{y}_1 + \mathsf{s}_1, \mathsf{y}_2 + \mathsf{s}_2, \mathsf{y}_3$ honestly.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ – i) $\mathcal{A}$ as sender to send $\mathsf{y}_1 + \mathsf{s}_1$ to P_2, and ii) $\mathcal{A}$ as receiver to obtain $\mathsf{y}_2 + \mathsf{s}_2$ from P_2.

– Simulates joint sharing as discussed earlier.

Figure 5.22: Simulator $\mathcal{S}^{P_1}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_1

Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

– Computes $\gamma^1_{\mathsf{ab}}, \gamma^2_{\mathsf{ab}}$, and γ^3_{ab} on behalf of P_0, P_1, P_2.

– Samples $\mathsf{u}^1, \mathsf{u}^2$ using the respective keys with $\mathcal{A}$ and computes r. The joint sharing of q is simulated as discussed earlier.

– Computes and sends w to $\mathcal{A}$ and simulate the steps of Π_{VrfyP0} honestly.

Online:

– Computes $\mathsf{y}_1 + \mathsf{s}_1, \mathsf{y}_2 + \mathsf{s}_2, \mathsf{y}_3$ honestly.

– Emulates two instances of $\mathcal{F}_{\mathsf{jsnd}}$ with $\mathcal{A}$ as sender to exchange $\mathsf{y}_1 + \mathsf{s}_1, \mathsf{y}_2 + \mathsf{s}_2$ among P_1, P_2.

– Simulates joint sharing as discussed earlier.

Figure 5.23: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_3

Reconstruction Protocol (Π_{Rec}, Fig. 5.5) Using the input of $\mathcal{A}$ obtained during simulation of sharing protocol, $\mathcal{S}_{\Pi_{\mathsf{Rec}}}$ invokes $\mathcal{F}_{\mathrm{GOD}}$ on behalf of $\mathcal{A}$ and obtains the function output y in clear. $\mathcal{S}_{\Pi_{\mathsf{Rec}}}$ calculates the missing share of $\mathcal{A}$ using y and the other shares. The missing share is then communicated to $\mathcal{A}$ by emulating the $\mathcal{F}_{\mathsf{jsnd}}$ functionality.

Chapter 6

ABY2.0: 2PC Semi-honest Protocols

This chapter provides details for the Layer I blocks of our 2PC framework ABY2.0. Some of the results in this chapter resulted in a publication at USENIX Security'21 [113][1]. Comparison of ABY2.0 with passively secure 2PC PPML framework of [102], in terms of the communication for multiplication, is presented in Table 6.1.

Work	#Active Parties	Security	Multiplication		Multiplication with Truncation[a]		Conversions[b]
			Comm_{pre}	Comm_{on}[c]	Comm_{pre}	Comm_{on}	
[102]	2	Semi-honest	$2\ell(\kappa+\ell)$	4ℓ	$2\ell(\kappa+\ell)$	4ℓ	A-B-G
ABY2.0	2	Semi-honest	$2\ell(\kappa+\ell)$	2ℓ	$2\ell(\kappa+\ell)$	2ℓ	A-B-G

[a] ℓ - size of ring in bits, κ - computational security parameter.
[b] A, B, G indicate support for arithmetic, boolean, and garbled worlds respectively.
[c] 'Comm' - communication, 'pre' - preprocessing, 'on' - online

Table 6.1: Comparison of semi-honest 2PC PPML frameworks

6.1 Preliminaries and Definitions

In our framework, we have two parties $\mathcal{P} = \{P_1, P_2\}$ who are connected by a bidirectional synchronous channel (e.g. instantiated via TLS over TCP/IP), and a static, semi-honest adversary that can corrupt at most one party. This framework is similar to that of the three-party framework ASTRA except for the absence of helper party P_0.

[1] This is joint work with Thomas Schneider and Hossein Yalame of TU Darmstadt. All co-authors contributed to the fruitful discussions that resulted in this publication. Ajith Suresh designed the new sharing scheme for two-party computation, provided new conversions between different MPC protocols, and benchmarked the protocols. Hossein Yalame designed the new circuits for parallel-prefix adder, comparison, and equality test based on multi-input AND gates and provided the depth-optimized variant of AES.

6.1.1 Sharing Semantics

For the arithmetic and boolean sharing, we follow masked evaluation technique, where a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is split into three shares. Two of the shares $(\lambda_\mathsf{v}^1, \lambda_\mathsf{v}^2)$ can be generated in the preprocessing phase independent of the value to be shared, and their sum can be interpreted as a mask (λ_v). The third share, dependent on v, can be computed in the online phase and can be treated as the masked value $\mathsf{m_v} = \mathsf{v} + \lambda_\mathsf{v}$.

Sharing Type	P_1	P_2
$[\cdot]$-sharing^a	v^1	v^2
$[\![\cdot]\!]$-sharing^b	$(\mathsf{m_v}, \lambda_\mathsf{v}^1)$	$(\mathsf{m_v}, \lambda_\mathsf{v}^2)$

$^a\mathsf{v} = \mathsf{v}^1 + \mathsf{v}^2$ $^b\lambda_\mathsf{v} = \lambda_\mathsf{v}^1 + \lambda_\mathsf{v}^2$, $\mathsf{m_v} = \mathsf{v} + \lambda_\mathsf{v}$

Table 6.2: Semantics for $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ in ABY2.0.

The sharing semantics is presented in Table 6.2, denoted by $[\![\cdot]\!]$, along with the semantics for $[\cdot]$-sharing. Both the sharings used are linear i.e. given sharings of $\mathsf{v}_1, \ldots, \mathsf{v}_m$ and public constants $c_1, \ldots, c_m$, sharing of $\sum_{i=1}^m c_i \mathsf{v}_i$ can be computed non-interactively for an integer m.

Notation 6.1 *(a) For the $[\![\cdot]\!]$-shares of n values $\mathsf{a}_1, \ldots, \mathsf{a}_n$, $\gamma_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^n \lambda_{\mathsf{a}_i}$ and $\mathsf{m}_{\mathsf{a}_1 \ldots \mathsf{a}_n} = \prod_{i=1}^n \mathsf{m}_{\mathsf{a}_i}$ (b) We use superscripts $\mathbf{B}$, and $\mathbf{G}$ to denote sharing semantics in boolean, and garbled world, respectively– $[\![\cdot]\!]^\mathbf{B}$, $[\![\cdot]\!]^\mathbf{G}$. We omit the superscript for arithmetic world.*

Sharing semantics for boolean sharing over $\mathbb{Z}_2$ is similar to arithmetic sharing except that addition is replaced with XOR. The semantics for garbled sharing are described in §6.3 with the relevant context.

6.1.2 Oblivious Transfer (OT)

In a 1-out-of-n Oblivious Transfer [70, 104] (OT) over ℓ-bit messages, the sender S inputs n messages $(x_1, \ldots, x_n)$ each of length ℓ bits, while the receiver R inputs the choice $c \in \{1, \ldots, n\}$. R receives x_c as output while S receives $\perp$ as output. The privacy guarantee is that S learns nothing about c, while R learns nothing about the inputs of S other than x_c. We use $\mathsf{n\text{-}OT}_\ell^m$ to denote m instances of 1-out-of-n OT on ℓ bit inputs.

OT is a fundamental building block for MPC [80] and requires expensive public-key cryptography [70]. The technique of OT Extension [71, 9, 81, 111] allows us to generate many OTs from

a small number (equal to the security parameter) of base OTs at the expense of symmetric-key operations alone. This reduces the cost of OT mainly to highly efficient symmetric-key primitives. Concretely, the OT Extension of [9] generates around 1 million 2-OT^1_ℓ per second with passive security. An orthogonal line of work considered pre-computation of OT [11], where all the cryptographic operations can be shifted to a setup phase, independent of the function to be evaluated. This technique enables a very efficient online phase for protocols that use OT. In the semi-honest setting, the state-of-the-art solution for OT extension [9] has communication $\kappa + 2\ell$ bits per OT for 2-OT^1_ℓ where κ denotes the computational security parameter.

A correlated OT (cOT) [9] is a variant of the traditional OT where the sender's input messages are correlated. In a cOT, the sender inputs a correlation function $f()$ and obtains the message pair $(x_0 \in_R \{0,1\}^\ell, x_1 = f(x_0))$ as the output. The receiver, on the other hand, inputs her choice c and obtains x_c as output. We use cOT^m_ℓ to denote m instances of 1-out-of-2 correlated OT on ℓ bit inputs. In the semi-honest setting, cOT^1_ℓ has communication $\kappa + \ell$ bits [9].

6.1.3 Homomorphic Encryption (HE)

The homomorphic property allows us to compute a ciphertext from a set of ciphertexts such that the plaintext underlying the former is a function of the underlying plaintexts of the latter. Towards this, one party called client generates a key-pair $(\mathsf{pk}, \mathsf{sk})$ for the HE scheme and sends pk to the other party called server. To perform a secure computation operation, the client encrypts its data using pk and sends this to the server. Now the server can locally compute the ciphertext corresponding to the operation and return the encrypted result to the client. The client can now decrypt the received ciphertext using her private key sk. An *additively HE* allows us to generate the ciphertext corresponding to the sum of the underlying plaintexts by doing operations on the ciphertexts. Prominent examples of additively HE schemes are Paillier [108], DGK [47] and RLWE-AHE [119]. On the other hand, fully homomorphic encryption schemes allow arbitrary computations under the encryption but are less efficient. See [3] for a more detailed description.

6.2 Arithmetic / Boolean 2PC

This section covers the details of our 2PC semi-honest protocol ABY2.0 over an arithmetic ring $\mathbb{Z}_{2^\ell}$. The protocol primarily consists of the following primitives – i) Sharing (§6.2.1), ii) Multiplication (§6.2.2), and iii) Reconstruction (§6.2.3).

6.2.1 Sharing

Protocol Π_{Sh} (Fig. 6.1) enables P_i to generate $[\![\cdot]\!]$-share of a value v. During the preprocessing phase, λ-shares are sampled non-interactively using the pre-shared keys (cf. §2.5.1) in a way that P_i will get the entire mask λ_{v}. During the online phase, P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2. For the special case when parties want to generate $[\![\mathsf{v}]\!]$ in the preprocessing, the protocol can be made non-interactive. W.l.o.g. consider the case when $P_i = P_1$. Parties set $\mathsf{m}_{\mathsf{v}} = 0$. P_1, P_2 sample λ^2_{v} non-interactively while P_1 sets $\lambda^1_{\mathsf{v}} = -(\mathsf{v} + \lambda^2_{\mathsf{v}})$. The case for $P_i = P_2$ is similar.

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

Input(s): $P_i : \mathsf{v}$, **Output:** $[\![\mathsf{v}]\!]$.

Preprocessing: Sample as follows: $P_i, P_1 : \lambda^1_{\mathsf{v}}$, $P_i, P_2 : \lambda^2_{\mathsf{v}}$.

Online: P_i computes $\mathsf{m}_{\mathsf{v}} = \mathsf{v} + \lambda_{\mathsf{v}}$ and sends to P_1, P_2.

Figure 6.1: $[\![\cdot]\!]$-sharing of a value v by party P_i in ABY2.0.

Lemma 6.1 (Communication) *Protocol* Π_{Sh} *(Fig. 6.1) requires a communication of at most* ℓ *bits and* 1 *round in the online phase.*

Proof: The preprocessing of Π_{Sh} is non-interactive as the parties sample non interactively using key setup $\mathcal{F}_{\text{KEY}}$ (§2.5.1). In the online phase, P_i sends m_{v} to either P_1 or P_2 (depending upon P_i) resulting in 1 round and communication of ℓ bits. □

6.2.1.1 Joint Sharing

Protocol Π_{JSh} enables parties P_1, P_2 to generate $[\![\cdot]\!]$-share of a value v known to both of them non-interactively. For this, parties set $\lambda^1_{\mathsf{v}} = \lambda^2_{\mathsf{v}} = 0$ and $\mathsf{m}_{\mathsf{v}} = \mathsf{v}$.

6.2.2 Multiplication

Given the shares of a, b, the goal of the multiplication protocol is to generate shares of $\mathsf{z} = \mathsf{ab}$. The protocol is designed such that P_i for $i \in \{1, 2\}$ obtain z^i in the online phase such that $\mathsf{z} = \mathsf{z}^1 + \mathsf{z}^2$. Parties then compute $[\![\mathsf{z}]\!]$ as $[\![\mathsf{z}^1]\!] + [\![\mathsf{z}^2]\!]$ to obtain the final output.

Online Note that,

$$\mathsf{z} = \mathsf{ab} = (\mathsf{m}_{\mathsf{a}} - \lambda_{\mathsf{a}})(\mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}}) = \mathsf{m}_{\mathsf{ab}} - \mathsf{m}_{\mathsf{a}}\lambda_{\mathsf{b}} - \mathsf{m}_{\mathsf{b}}\lambda_{\mathsf{a}} + \gamma_{\mathsf{ab}} \quad \text{(cf. notation 6.1)} \tag{6.1}$$

Let $\mathsf{z} = \mathsf{z}_1 + \mathsf{z}_2$, where z_1 and z_2 can be computed respectively by P_1 and P_2.

$$\begin{aligned} P_1 : \mathsf{z}_1 &= \mathsf{m_{ab}} - \lambda^1_{\mathsf{a}}\mathsf{m_b} - \lambda^1_{\mathsf{b}}\mathsf{m_a} + [\gamma_{\mathsf{ab}}]_1 \\ P_2 : \mathsf{z}_2 &= \quad\quad -\lambda^2_{\mathsf{a}}\mathsf{m_b} - \lambda^2_{\mathsf{b}}\mathsf{m_a} + [\gamma_{\mathsf{ab}}]_2 \end{aligned} \tag{6.2}$$

During preprocessing, parties rely on Π_{MultPre} to generate an additive sharing $([\cdot])$ of γ_{ab}. We note that Turbospeedz [17] achieves same online cost as that of ours, but with a more expensive preprocessing. We provide more details in §6.2.5.

Protocol $\Pi_{\mathsf{Mult}}(\mathsf{a}, \mathsf{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \mathsf{ab}$.

Preprocessing: Execute Π_{MultPre} on $[\lambda_{\mathsf{a}}]$ and $[\lambda_{\mathsf{b}}]$ to generate $[\gamma_{\mathsf{ab}}]$.

Online:

1. Compute: $P_1 : \mathsf{z}_1 = \mathsf{m_{ab}} - \lambda^1_{\mathsf{a}}\mathsf{m_b} - \lambda^1_{\mathsf{b}}\mathsf{m_a} + [\gamma_{\mathsf{ab}}]_1$, $P_2 : \mathsf{z}_2 = -\lambda^2_{\mathsf{a}}\mathsf{m_b} - \lambda^2_{\mathsf{b}}\mathsf{m_a} + [\gamma_{\mathsf{ab}}]_2$
2. If $\mathsf{isTr} = 1$, P_i sets $\mathsf{p}_i = \mathsf{z}^{\mathsf{t}}_i$, else $\mathsf{p}_i = \mathsf{z}_i$ where $i \in \{1, 2\}$. Execute $\Pi_{\mathsf{Sh}}(P_i, \mathsf{p}_i)$ to generate $[\![\mathsf{p}_i]\!]$.
3. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}_1]\!] + [\![\mathsf{p}_2]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 6.2: Multiplication with / without truncation in ABY2.0.

Preprocessing We now provide the details for instantiating Π_{MultPre} using two of the well-known primitives: i) Oblivious Transfer (OT) as used in [51, 78] and ii) Homomorphic Encryption (HE) as used in [68, 48, 119]. These two approaches have been rallied against each other in terms of practical efficiency in the past, and fair competition is still going on. In our work, we make only black-box access to these primitives, and hence any improvement in any of them will directly impact the overall efficiency of the setup phase of our protocols.

Note that $\gamma_{\mathsf{ab}} = (\lambda^1_{\mathsf{a}} + \lambda^2_{\mathsf{a}})(\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{b}}) = \lambda^1_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^1_{\mathsf{b}} + \lambda^2_{\mathsf{a}}\lambda^2_{\mathsf{b}}$. Here P_i for $i \in \{1, 2\}$ can locally compute $\lambda^i_{\mathsf{a}}\lambda^i_{\mathsf{b}}$ and hence the problem reduces to computing $\lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{b}}$ and $\lambda^2_{\mathsf{a}}\lambda^1_{\mathsf{b}}$.

OT based Π_{MultPre}*:* In our OT-based approach, we use Correlated OTs (cOT) [9] where the sender inputs a correlation function $f(\cdot)$ to cOT and obtains (m_0, m_1), where m_0 is a random element and $m_1 = f(m_0)$. We use cOT^n_ℓ to represent n parallel instances of 1-out-of-2 Correlated OTs on ℓ bit input strings.

To compute $[\lambda^1_{\mathsf{a}}\lambda^2_{\mathsf{b}}]$, the parties execute cOT^ℓ_ℓ with P_1 being the sender and P_2 being the

receiver. For the j-th instance of cOT where $j \in \{0, \ldots, \ell-1\}$, P_1 inputs the correlation $f_j(x) = x + 2^j \lambda_{\mathsf{a}}^1$ and obtains $(m_{j,0} = r_j, m_{j,1} = r_j + 2^j \lambda_{\mathsf{a}}^1)$. P_2 inputs choice bit b_j as the j-th bit of λ_{b}^2 and obtains m_{j,b_j} as output. Now the $[\cdot]$-shares are defined as $[\lambda_{\mathsf{a}}^1 \lambda_{\mathsf{b}}^2]_1 = \sum_{j=0}^{\ell-1}(-r_j)$ and $[\lambda_{\mathsf{a}}^1 \lambda_{\mathsf{b}}^2]_2 = \sum_{j=0}^{\ell-1} m_{j,b_j}$. Computation of $\lambda_{\mathsf{a}}^2 \lambda_{\mathsf{b}}^1$ proceeds similarly with the role of the parties reversed.

In the OT-based approach [51, 78], the technique of OT extension [9, 81, 111] can be used. One instance of Π_{MultPre} requires two instances of cOT_ℓ^ℓ where each instance has communication $\ell(\kappa+\ell)$ bits. Over a 64-bit ring, this corresponds to 3072 bytes. Recently, [26] came up with a very efficient OT extension technique named Silent OT Extension which claims to outperform state-of-the-art solutions for performing Π_{MultPre}. Since our protocol makes black-box calls to Π_{MultPre}, it can directly benefit from the performance improvements of [26].

HE-based Π_{MultPre}*:* In a HE based solution, P_1, using its public key pk_1, encrypts its messages $\lambda_{\mathsf{a}}^1, \lambda_{\mathsf{b}}^1$ in independent ciphertexts and sends the ciphertexts to P_2. In parallel, P_2 computes the ciphertexts corresponding to $\lambda_{\mathsf{a}}^2, \lambda_{\mathsf{b}}^2$ and a random element $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$ using pk_1. Upon receiving the ciphertexts from P_1, P_2 computes the ciphertext corresponding to $\mathsf{v} = \lambda_{\mathsf{a}}^1 \lambda_{\mathsf{b}}^2 + \lambda_{\mathsf{a}}^2 \lambda_{\mathsf{b}}^1 - \mathsf{r}$ using the homomorphic property of the underlying HE. P_2 then sends encryption of v to P_1 who then decrypts it using its secret key sk_1. Note that (v, r) forms an additive sharing of the desired value: $\lambda_{\mathsf{a}}^1 \lambda_{\mathsf{b}}^2 + \lambda_{\mathsf{a}}^2 \lambda_{\mathsf{b}}^1 = \mathsf{v} + \mathsf{r}$.

Recently, Ring LWE-based AHE [119] was shown to outperform the solutions based on OT for generating multiplication triples. The authors observed that the plaintext space is much larger than the range of the values being encrypted. Thus they used the technique of *ciphertext packing*, using Microsoft SEAL library, where ciphertexts corresponding to multiple plaintexts are packed into a single ciphertext. This optimizes the number of ciphertexts being sent back and the number of decryptions on P_1's side. In [119], the amortized communication cost for performing one instance of Π_{MultPre} over a 64-bit ring with a security level of 128 bits is 448 bytes, which is a 7× improvement over the best OT-based solutions [51] available at that time.

Lemma 6.2 (Communication) *Protocol* Π_{Mult} *(Fig. 6.2) (in ABY2.0) requires* $2\ell(\kappa+\ell)$ *bits of communication in the preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

Proof: During the preprocessing, as part of Π_{MultPre}, we use 2 instances of correlated OTs (cOT) [9] which incur a communication of $\ell + \kappa$ bits per cOT on ℓ-bit strings, where κ is the computational security parameter. During the online phase, each of P_1 and P_2 executes one instance of Π_{Sh} and the cost follows from Lemma 6.1. □

6.2.2.1 Truncation

To accommodate truncation, following ASTRA, P_i for $i \in \{1,2\}$ locally truncates z_i before executing the sharing in the online of Π_{Mult} (Fig. 6.2). The correctness follows from [102].

6.2.2.2 Multiplication with constant

Multiplication by a constant in MPC is typically local. Given constant α and $[\![\mathsf{v}]\!]$, the $[\![\cdot]\!]$-shares of the product $\mathsf{y} = \alpha\mathsf{v}$ can be locally computed as per (6.3).

$$\mathsf{m}_\mathsf{y} = \alpha\mathsf{m}_\mathsf{u}, \quad \lambda^1_\mathsf{y} = \alpha\lambda^1_\mathsf{v}, \quad \lambda^2_\mathsf{y} = \alpha\lambda^2_\mathsf{v} \tag{6.3}$$

However, in FPA, we need to perform a truncation on the output. Let $\alpha\mathsf{v} = \beta^1 + \beta^2$ where $\beta^1 = \alpha.(\mathsf{m}_\mathsf{v} - \lambda^1_\mathsf{v})$ and $\beta^2 = -\alpha.\lambda^2_\mathsf{v}$. P_i for $i \in \{1,2\}$ locally truncates β^i and executes the sharing protocol Π_{Sh} on the truncated value. Parties locally compute $[\![\alpha\mathsf{v}]\!] = [\![\beta^1]\!] + [\![\beta^2]\!]$ to obtain the final result.

6.2.3 Reconstruction

$\Pi_{\mathsf{Rec}}(\mathcal{P}, \mathsf{v})$ enables parties to compute v, given its $[\![\cdot]\!]$-share. For this, P_1 sends λ^1_v to P_2 and P_2 sends λ^2_v to P_1. Parties locally compute $\mathsf{v} = \mathsf{m}_\mathsf{v} - \lambda^1_\mathsf{v} - \lambda^2_\mathsf{v}$. Reconstruction towards a single party can be viewed as a special case.

Lemma 6.3 (Communication) *Protocol Π_{Rec} requires a communication of 2ℓ bits and 1 round.*

6.2.4 Multi-input Multiplication

6.2.4.1 3-input multiplication

To compute $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{abc}$, note that

$$\begin{aligned}\mathsf{z} = \mathsf{abc} &= (\mathsf{m}_\mathsf{a} - \lambda_\mathsf{a})(\mathsf{m}_\mathsf{b} - \lambda_\mathsf{b})(\mathsf{m}_\mathsf{c} - \lambda_\mathsf{c}) \\ &= \mathsf{m}_\mathsf{abc} - \mathsf{m}_\mathsf{ac}\lambda_\mathsf{b} - \mathsf{m}_\mathsf{bc}\lambda_\mathsf{a} - \mathsf{m}_\mathsf{ab}\lambda_\mathsf{c} + \mathsf{m}_\mathsf{a}\gamma_\mathsf{bc} + \mathsf{m}_\mathsf{b}\gamma_\mathsf{ac} + \mathsf{m}_\mathsf{c}\gamma_\mathsf{ab} - \gamma_\mathsf{abc} \quad \text{(cf. notation 6.1)}\end{aligned} \tag{6.4}$$

Similar to Π_{Mult}, parties rely on Π_{MultPre} to generate an additive sharing $([\cdot])$ of $\gamma_\mathsf{ab}, \gamma_\mathsf{bc}$ and γ_ac. Parties then generate $[\gamma_\mathsf{abc}]$ using another instance of Π_{MultPre} with inputs γ_ab and λ_c.

Lemma 6.4 (Communication) *Protocol Π_{Mult3} (in ABY2.0) requires $8\ell(\kappa + \ell)$ bits of communication in the preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

Proof: The preprocessing involves four instances of Π_{MultPre} each costing a communication of $2\ell(\kappa+\ell)$ bits. The online phase is similar to Π_{Mult} and the costs follow from Lemma 6.2. □

6.2.4.2 4-input multiplication

For the case of 4-input multiplication with $\mathsf{z} = \mathsf{abcd}$, note that

$$\begin{aligned}\mathsf{z} = \mathsf{abcd} - \mathsf{r} &= (\mathsf{m_a} - \lambda_\mathsf{a})(\mathsf{m_b} - \lambda_\mathsf{b})(\mathsf{m_c} - \lambda_\mathsf{c})(\mathsf{m_d} - \lambda_\mathsf{d}) \\ &= \mathsf{m_{abcd}} - \mathsf{m_{abc}}\lambda_\mathsf{d} - \mathsf{m_{abd}}\lambda_\mathsf{c} - \mathsf{m_{acd}}\lambda_\mathsf{b} - \mathsf{m_{bcd}}\lambda_\mathsf{a} + \mathsf{m_{ab}}\gamma_\mathsf{cd} + \mathsf{m_{ac}}\gamma_\mathsf{bd} + \mathsf{m_{ad}}\gamma_\mathsf{bc} + \mathsf{m_{bc}}\gamma_\mathsf{ad} \\ &\quad + \mathsf{m_{bd}}\gamma_\mathsf{ac} + \mathsf{m_{cd}}\gamma_\mathsf{ab} - \mathsf{m_a}\gamma_\mathsf{bcd} - \mathsf{m_b}\gamma_\mathsf{acd} - \mathsf{m_c}\gamma_\mathsf{abd} - \mathsf{m_d}\gamma_\mathsf{abc} + \gamma_\mathsf{abcd} \quad \text{(cf. notation 6.1)} \end{aligned} \tag{6.5}$$

Here the parties need to generate $[\cdot]$-shares of $\gamma_\mathsf{ab}, \gamma_\mathsf{ac}, \gamma_\mathsf{ad}, \gamma_\mathsf{bc}, \gamma_\mathsf{bd}, \gamma_\mathsf{cd}, \gamma_\mathsf{abc}, \gamma_\mathsf{abd}, \gamma_\mathsf{acd}, \gamma_\mathsf{bcd}$ and γ_abcd. This is computed similarly as in 3-input multiplication and the protocol is denoted as Π_{Mult4}.

Lemma 6.5 (Communication) *Protocol Π_{Mult4} (in ABY2.0) requires $22\ell(\kappa+\ell)$ bits of communication in the preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

6.2.4.3 Comparison with the LUT-based protocol of [52]

We compare our multi-input AND gate protocols with [52] for two, three and four inputs. [52] proposed two variants – i) OP-LUT - optimized online communication of $2N$ bits, and ii) SP-LUT - optimized total communication of $2\kappa + 2^N$ bits. The concrete details are given in Table 6.3.

Gate	Protocol	Preprocessing	Online	
		Communication	Communication	Rounds
AND $\mathsf{z} = \mathsf{ab}$	OP-LUT	206	4	**1**
	SP-LUT	190	6	**1**
	ABY2.0	134	**2**	**1**
AND3 $\mathsf{z} = \mathsf{abc}$	OP-LUT	285	6	**1**
	SP-LUT	221	11	**1**
	ABY2.0	250	**2**	**1**
AND4 $\mathsf{z} = \mathsf{abcd}$	OP-LUT	492	8	**1**
	SP-LUT	236	20	**1**
	ABY2.0	412	**2**	**1**

Table 6.3: Comparison of ABY2.0 and [52] (OP-LUT and SP-LUT). Communication is provided in bits. Best values for the online phase are marked in bold.

6.2.5 Comparison with Turbospeedz [17] and [106]

Here, we compare our 2PC protocol with Turbospeedz [17] and [106].

6.2.5.1 Comparison with Turbospeedz [17]

For the 2-input multiplication, Turbospeedz [17] presented a protocol that reduces the online communication of SPDZ-style protocols from 4 to 2 ring elements using a function-dependent preprocessing. Turbospeedz first executes a SPDZ-like preprocessing where random multiplication triples are generated. These triples are then associated with the multiplication gates using additional values that they call "external values" (cf. [17], §3.2). On the contrary, we obtain the preprocessing data directly and hence save communication of 4 ring elements and storage of 5 ring elements compared with Turbospeedz. Table 6.4 provides the communication and storage required for the 2-input multiplication protocol of ABY [51], Turbospeedz [17] and ABY2.0.

Phase	Parameter	ABY [51]	Turbospeedz [17]	ABY2.0
Preprocessing	Storage	3ℓ	9ℓ	4ℓ
	Communication	$\|\mathsf{Triple}\|$	$\|\mathsf{Triple}\| + 4\ell$	$\|\mathsf{Triple}\|$
Online	Storage	5ℓ	5ℓ	$\mathbf{3\ell}$
	Communication	4ℓ	$\mathbf{2\ell}$	$\mathbf{2\ell}$
Total	Storage	8ℓ	14ℓ	7ℓ
	Communication	$\|\mathsf{Triple}\| + 4\ell$	$\|\mathsf{Triple}\| + 6\ell$	$\|\mathsf{Triple}\| + 2\ell$

Table 6.4: Comparison of ABY2.0 with ABY [51] and Turbospeedz [17] in terms of storage and communication for a single multiplication. All values are given in bits. $|\mathsf{Triple}|$ denotes the communication required to generate a multiplication triple. Best values for the online phase are marked in bold.

For the multi-input multiplication (fan-in of N), the tree-based method (multiplying N elements by taking two at a time) requires $\log_2(N)$ rounds for both ABY [51] and Turbospeedz [17], while it requires communication of $4(N-1)$ ring elements for ABY and $2(N-1)$ elements for Turbospeedz in the online phase.

6.2.5.2 Comparison with [106]

Recently, [106] proposed round-efficient solutions for multi-input multiplication using a preprocessing for which the communication cost grows exponentially with the fan-in of the multiplication gate. However, for an N-input multiplication, [106] requires an online communication

of $2N-2$ ring elements. On the contrary, ABY2.0 requires only an online communication of 2 ring elements, and the preprocessing cost remains the same as that of [106]. Note that since the preprocessing cost grows exponentially with the number of inputs to the multiplication gate, [106] considered only up to 5-input multiplication gates in their work.

Π_{Mult} when input parties are the computing parties For the case of a two-input multiplication gate, [106] considered a special case where the input parties are the computing parties (cf. [106], §3.4). For this case, [106] proposed a protocol for which the online communication is 2 ring elements. For the same setting, we observe that our solution results in a protocol with zero online communication. To see this, recall the online phase of our multiplication protocol Π_{Mult} (Fig. 6.2). The modified protocol is as follows: During the online phase, party P_i for $i \in \{1,2\}$ locally computes z_i such that $\mathsf{z}_1 + \mathsf{z}_2 = \mathsf{z}$. Now to generate $[\![\mathsf{z}]\!]$, parties locally set $\lambda^1_{\mathsf{z}} = -\mathsf{z}_1, \lambda^2_{\mathsf{z}} = -\mathsf{z}_2$ and $\mathsf{m}_{\mathsf{z}} = 0$. It is easy to see that $\mathsf{z} = \mathsf{m}_{\mathsf{z}} - \lambda^1_{\mathsf{z}} - \lambda^2_{\mathsf{z}}$.

6.3 Garbled World

The GC world comprises a single execution with P_1 acting as garbler and P_2 as the evaluator.

Input Phase Given that the function input x is already available as $[\![\mathsf{x}]\!]^{\mathbf{B}}$, the boolean values $\alpha_{\mathsf{x}} = \mathsf{m}_{\mathsf{x}} \oplus \lambda^1_{\mathsf{x}}, \lambda^2_{\mathsf{x}}$ act as the *new* inputs for the garbled computation, and garbled sharing ($[\![\cdot]\!]^{\mathbf{G}}$) is generated for each of these values. The $[\![\cdot]\!]^{\mathbf{G}}$-shares thus generated defines the compound sharing, $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\alpha_{\mathsf{x}}]\!]^{\mathbf{G}}, [\![\lambda^2_{\mathsf{x}}]\!]^{\mathbf{G}})$ for every input x to the function to be evaluated via the GC. We first discuss the semantics for $[\![\cdot]\!]^{\mathbf{G}}$-sharing followed by steps for generating $[\![\cdot]\!]^{\mathbf{C}}$-sharing.

Garbled sharing semantics A value $\mathsf{v} \in \mathbb{Z}_2$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared (garbled shared) amongst $\mathcal{P}$ if P_1 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_1 = \mathsf{K}^0_{\mathsf{v}}$ and P_2 holds $[\![\mathsf{v}]\!]^{\mathbf{G}}_2 = \mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$. Here, $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}} = \mathsf{K}^0_{\mathsf{v}} \oplus \mathsf{v}\Delta$, and Δ, which is known only to the garbler P_1, denotes the global offset with its least significant bit set to 1 and is same for every wire in the circuit. A value $\mathsf{x} \in \mathbb{Z}_2$ is said to be $[\![\cdot]\!]^{\mathbf{C}}$-shared (compound shared) if each value from $(\alpha_{\mathsf{x}}, \lambda^2_{\mathsf{x}})$ is $[\![\cdot]\!]^{\mathbf{G}}$-shared. We write $[\![\mathsf{x}]\!]^{\mathbf{C}} = ([\![\alpha_{\mathsf{x}}]\!]^{\mathbf{G}}, [\![\lambda^2_{\mathsf{x}}]\!]^{\mathbf{G}})$.

Generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ and $[\![\mathsf{x}]\!]^{\mathbf{C}}$ Protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}(\mathcal{P}, \mathsf{v})$ enables generation of $[\![\mathsf{v}]\!]^{\mathbf{G}}$ given v. Garbler P_1 generates $\{\mathsf{K}^{\mathsf{b}}_{\mathsf{v}}\}_{\mathsf{b}\in\{0,1\}}$ which denotes the key for value b on wire v, following the free-XOR technique [82, 84]. If the value v is known to P_1, it sends $\mathsf{K}^{\mathsf{v}}_{\mathsf{v}}$ to P_2. For the case when the evaluator P_2 knows v, parties engage in a cOT^1_κ with P_1 being the sender and P_2 being the receiver. Here P_1 inputs the correlation function $f_R(\mathsf{y}) = \mathsf{y} \oplus \Delta$ and obtains $(\mathsf{K}^0_{\mathsf{v}}, \mathsf{K}^{\mathsf{v}}_{\mathsf{v}} = \mathsf{K}^0_{\mathsf{v}} \oplus \Delta)$

while P_2 inputs v as choice bit and receives $\mathsf{K}_{\mathsf{v}}^{\mathsf{v}}$ as the output. To generate $[\![\mathsf{x}]\!]^{\mathbf{C}}$, $\Pi_{\mathsf{Sh}}^{\mathbf{G}}$ is invoked for each of α_{x} and λ_{x}^2.

Evaluation Let $f(\mathsf{x})$ be the function to be evaluated. At this point, the function input is $[\![\cdot]\!]^{\mathbf{C}}$-shared. This renders $[\![\cdot]\!]^{\mathbf{G}}$-sharing for the input of the GC that corresponds to the function $f'\big(\alpha_{\mathsf{x}}, \lambda_{\mathsf{x}}^2\big)$ which first combines the given boolean-shares to compute the actual input and then applies f on it. Let GC denotes the garbled circuit to be sent to P_2 by garbler P_1. Sending of GC is overlapped with the key transfer (during generation of $[\![\mathsf{x}]\!]^{\mathbf{C}}$), to save rounds, where P_1 sends GC to P_2. On receiving the GC, P_2 evaluate it and obtain the key corresponding to the output, say z. This generates $[\![\mathsf{z}]\!]^{\mathbf{G}}$.

Output phase The goal of output computation is to compute the output z from $[\![\mathsf{z}]\!]^{\mathbf{G}}$. To reconstruct z towards P_2, P_1 sends the least significant bit p of $\mathsf{K}_{\mathsf{z}}^0$, referred to as the decoding information, to P_2. P_2 uses the received p to reconstruct z as $\mathsf{z} = \mathsf{p} \oplus \mathsf{q}$, where q denotes the least significant bit of $\mathsf{K}_{\mathsf{z}}^{\mathsf{z}}$. P_2 then sends z to P_1 completing the protocol.

6.4 Security proofs

The simulation for the semi-honest 2PC case is straightforward in the $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{MultPre}}\}$-hybrid model. Here $\mathcal{F}_{\mathsf{setup}}$ (§2.5.1) denotes the ideal functionality for the shared-key setup and $\mathcal{F}_{\mathsf{MultPre}}$ denotes the ideal functionality for the multiplication preprocessing Π_{MultPre}. 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 $\mathcal{A}$. Since $\mathcal{S}$ is given the input and output of the $\mathcal{A}$, it can compute all the intermediate values of the circuit Ckt in clear.

For the input sharing of value v, $\mathcal{S}$ receives the m_{v} from $\mathcal{A}$ on behalf of the honest parties. Similarly, for the inputs of honest parties, $\mathcal{S}$ interacts with the $\mathcal{A}$ with the inputs set to 0. The simulated view is indistinguishable from the ideal view due to the privacy of the underlying sharing scheme. The linear gates involve no communication, while simulation of the multiplication protocol is straightforward. Moreover, simulation for the joint sharing (Π_{JSh}) instances is similar to that of the sharing protocol. The protocol's design is such that $\mathcal{S}$ will always know the value to be sent as part of the joint sharing protocol. Finally, for the reconstruction towards $\mathcal{A}$, $\mathcal{S}$ calculates the missing share of $\mathcal{A}$ using y and the other shares. The missing share is then communicated to $\mathcal{A}$ as per the reconstruction protocol.

Part II

Layer II: Building Blocks

Introduction to Layer II

In this part, we provide the details of the Layer II blocks of our three-layer architecture (Fig. 1.1). To begin with, we provide a high-level overview of the building blocks next. Moreover, for some of the blocks, such as matrix multiplication and non-linear activation functions, the constructions are generic and instantiated with the protocols from the corresponding framework. We provide a detailed description for those blocks and omit the same from the specific chapters to avoid repetition.

Scalar Dot Product (Π_{dotp}) Scalar Dot Product forms the fundamental building block for most of the ML algorithms and hence designing efficient constructions for the same are of utmost importance. Given the $[\![\cdot]\!]$-shares of d-length vectors $\vec{\mathbf{a}}, \vec{\mathbf{b}}$, dot product protocol Π_{dotp} computes the $[\![\cdot]\!]$-shares of z with $\mathsf{z} = \vec{\mathbf{a}} \odot \vec{\mathbf{b}} = \sum_{i=1}^{\mathsf{d}} \mathsf{a}_i \mathsf{b}_i$. One trivial way is to invoke the multiplication protocol corresponding to each of the d underlying multiplications. This would result in communication linear in the vector size d. In this thesis, we propose methods to make the online communication independent of the vector size for all our settings. Moreover, the communication in the preprocessing phase is also made independent of the vector size for the case of three and four-party settings.

Matrix Operations and Convolutions Linear matrix operations, such as addition of two matrices $\mathbf{A}, \mathbf{B}$ to generate matrix $\mathbf{C} = \mathbf{A} + \mathbf{B}$, can be computed by extending the scalar operations (addition, in this case) with respect to each element of the matrix. Matrix multiplication, on the other hand, can be expressed as a collection of dot products, where the element in the i^{th} row and j^{th} column of $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{A}, \mathbf{B}$ are matrices of dimension $\mathsf{p} \times \mathsf{q}$, $\mathsf{q} \times \mathsf{r}$, respectively, can be computed as a dot product of the i^{th} row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$. Thus, computing $\mathbf{C}$ of dimension $\mathsf{p} \times \mathsf{r}$ requires pr dot products on vectors of length q. This improves the cost of matrix multiplication over the naive approach which requires pqr multiplications.

We abuse notation and follow the $[\![\cdot]\!]$-sharing semantics for matrices. For $\mathbf{X}^{u \times v}$, we have $\mathsf{m}_{\mathbf{X}} = \mathbf{X} \bigoplus [\lambda^1_{\mathbf{X}}] \bigoplus [\lambda^2_{\mathbf{X}}] \bigoplus [\lambda^3_{\mathbf{X}}]$ for the case of active frameworks ($\mathsf{SWIFT}, \mathsf{Tetrad}$) and $\mathsf{m}_{\mathbf{X}} =$

$\mathbf{X} \bigoplus [\lambda^1_{\mathbf{X}}] \bigoplus [\lambda^2_{\mathbf{X}}]$ for the case of passive frameworks (ASTRA, ABY2.0) . Here $\mathsf{m}_{\mathbf{X}}$, $[\lambda^1_{\mathbf{X}}]$, $[\lambda^2_{\mathbf{X}}]$, and $[\lambda^3_{\mathbf{X}}]$ are matrices of dimension $u \times v$, and $\bigoplus$ denote the matrix addition operation. Looking ahead $\bigominus, \bigodot$ will be used to denote matrix subtraction and multiplication operation, respectively.

Convolutions: Convolutions form an important building block in several neural network architectures and can be represented as matrix multiplications, as explained in the example below. Consider a 2-dimensional convolution (Conv) of a 3×3 input matrix $\mathbf{X}$ with a kernel $\mathbf{K}$ of size 2×2. This can be represented as a matrix multiplication as follows.

$$\mathsf{Conv}\left(\begin{bmatrix} \mathsf{x}_1 & \mathsf{x}_2 & \mathsf{x}_3 \\ \mathsf{x}_4 & \mathsf{x}_5 & \mathsf{x}_6 \\ \mathsf{x}_7 & \mathsf{x}_8 & \mathsf{x}_9 \end{bmatrix}, \begin{bmatrix} \mathsf{k}_1 & \mathsf{k}_2 \\ \mathsf{k}_3 & \mathsf{k}_4 \end{bmatrix}\right) = \begin{bmatrix} \mathsf{x}_1 & \mathsf{x}_2 & \mathsf{x}_4 & \mathsf{x}_5 \\ \mathsf{x}_2 & \mathsf{x}_3 & \mathsf{x}_5 & \mathsf{x}_6 \\ \mathsf{x}_4 & \mathsf{x}_5 & \mathsf{x}_7 & \mathsf{x}_8 \\ \mathsf{x}_5 & \mathsf{x}_6 & \mathsf{x}_8 & \mathsf{x}_9 \end{bmatrix} \begin{bmatrix} \mathsf{k}_1 \\ \mathsf{k}_2 \\ \mathsf{k}_3 \\ \mathsf{k}_4 \end{bmatrix}$$

Generally, convolving a $f \times f$ kernel over a $w \times h$ input with $p \times p$ padding using $s \times s$ stride having i input channels and o output channels, is equivalent to performing a matrix multiplication on matrices of dimension $(w' \cdot h') \times (i \cdot f \cdot f)$ and $(i \cdot f \cdot f) \times (o)$ where $w' = \frac{w - f + 2p}{s} + 1$ and $h' = \frac{h - f + 2p}{s} + 1$. We refer readers to [133, 130] for more details.

Secure Comparison (Π_{bitext}) Comparing two arithmetic values is one of the major hurdles in realizing efficient secure ML algorithms. Given arithmetic shares $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$, parties wish to check whether $\mathsf{a} > \mathsf{b}$. To compute $\mathsf{a} > \mathsf{b}$ in the FPA representation, given its $[\![\cdot]\!]$-sharing, Π_{bitext} uses the technique of extracting the most significant bit (msb) of the value $\mathsf{v} = \mathsf{a} - \mathsf{b}$ [101, 110, 85].

To compute the msb, we use two variants - i) the communication optimized parallel prefix adder (PPA) circuit from ABY3 [101] ($2(\ell - 1)$ AND gates, $\log \ell$ depth), and ii) the round optimized bit extraction circuit from ABY2 [113]. The circuit of ABY2 uses multi-input AND gates and has a multiplicative depth of $\log_4(\ell)$. These circuits take two ℓ-bit values in boolean sharing as the input and output the result in boolean sharing form.

Bit to Arithmetic (Π_{bit2A}) / Bit Injection (Π_{bitInj}) The bit to arithmetic protocol, Π_{bit2A}, enables computing the arithmetic sharing ($[\![\cdot]\!]$) of a bit b given its boolean sharing $[\![\mathsf{b}]\!]^{\mathbf{B}}$. Let b^{R} denotes the value of $\mathsf{b} \in \{0, 1\}$ over the arithmetic ring $\mathbb{Z}_{2^\ell}$. Then for $\mathsf{b} = \mathsf{b}_1 \oplus \mathsf{b}_2$, note that $\mathsf{b}^{\mathsf{R}} = (\mathsf{b}_1^{\mathsf{R}} - \mathsf{b}_2^{\mathsf{R}})^2$. Similarly, Π_{dbit2A} protocol computes the arithmetic sharing of $\mathsf{b}_1\mathsf{b}_2$ given the boolean sharings $[\![\mathsf{b}_1]\!]^{\mathbf{B}}$ and $[\![\mathsf{b}_2]\!]^{\mathbf{B}}$.

Given the boolean sharing of bit b and the arithemetic sharing of a value v, the bit injection protocol, Π_{bitInj}, enables computing the arithmetic sharing corresponding to the value bv.

Similarly, Π_{dbitInj} computes the arithmetic sharing of $\mathsf{b}_1\mathsf{b}_2\mathsf{v}$ given $[\![\mathsf{b}_1]\!]^{\mathbf{B}}$, $[\![\mathsf{b}_2]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$.

Equality Test (Π_{eq}) Given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$, the goal of the Equality Testing (Π_{eq}) protocol is to check whether $\mathsf{a} \stackrel{?}{=} \mathsf{b}$ or not. An equivalent formulation of the problem [21, 106] is to check if all the bits of $\mathsf{a} - \mathsf{b}$ are 0 or not. This simple primitive is crucial in building efficient protocol for applications like Circuit-based Private Set Intersection [117, 114, 115], the Table Lookup Protocol from [52], and Data Mining [21].

On a high level, the protocol starts with the parties computing the boolean shares of two value $\mathsf{v}_1, \mathsf{v}_2$ using the $[\![\cdot]\!]$-shares of a and b. The values $\mathsf{v}_1, \mathsf{v}_2$ are computed such that $\mathsf{v}_1 = \mathsf{v}_2$ implies $\mathsf{a} = \mathsf{b}$. For instance, in ASTRA, parties set $\mathsf{v}_1 = (\mathsf{m}_\mathsf{a} - \lambda^1_\mathsf{a}) - (\mathsf{m}_\mathsf{b} - \lambda^1_\mathsf{b})$ and $\mathsf{v}_2 = \lambda^2_\mathsf{a} - \lambda^2_\mathsf{b}$. Note that the value v_i can be locally computed by party P_i for $i \in \{1, 2\}$ and hence can generate the boolean shares.

The parties then locally compute the boolean shares of $\mathsf{v} = \mathsf{v}_1 \oplus \mathsf{v}_2$. If $\mathsf{v}_1 = \mathsf{v}_2$, then all the bits of v should be 0. Or in other words, all the bits of $\bar{\mathsf{v}}$ should be 1. This can be checked by computing an AND of all the bits of $\bar{\mathsf{v}}$. For this, the parties use 4-input AND gates and a tree structure, where 4 bits are taken at a time and the AND of them is computed in one go. This approach improves the round complexity by a factor of two ($\log_2(\ell)$ to $\log_4(\ell)$ for ℓ-bit inputs) over the traditional approach using 2-input AND gates. Parties can use the Π_{bit2A} protocol to generate the arithmetic equivalent of the result in shared form.

Piecewise-polynomial functions Piece-wise polynomial functions are constructed as a series of constant polynomials $f_1, \ldots, f_m$ with public coefficients and $c_1 < \ldots < c_m$ such that,

$$f(y) = \begin{cases} 0, & y < c_1 \\ f_1, & c_1 \leq y < c_2 \\ \ldots & \\ f_m, & c_m \leq y \end{cases}$$

For computing f, we first compute a set of bits $\mathsf{b}_1, \ldots, \mathsf{b}_m$ such that $\mathsf{b}_i = 1$ if $y \geq c_i$ and 0 otherwise. f can be computed as, $f(y) = \sum_{i=1}^{m} \mathsf{b}_i \cdot (f_i - f_{i-1})$, where $f_0 = 0$ and $f_m = 1$. Given the arithmetic shares $([\![\cdot]\!])$ of y, one can obtain the boolean shares $([\![\cdot]\!]^{\mathbf{B}})$ of the bits $\mathsf{b}_1, \ldots, \mathsf{b}_m$ using secure comparison. The bit injection protocol is then used to compute the $[\![\cdot]\!]$-shares of $\mathsf{b}_i \cdot (f_i - f_{i-1})$. Note that $f(y)$ can be viewed as a sum of m bit injections, and parties can add up the shares locally to obtain the final result. In $\Pi_{\mathsf{piecewise}}$, we optimize the communication further and show how to make the online communication independent of m.

Non-Linear Activation functions We use the following three widely used activation functions – (i) Rectified Linear Unit (ReLU), (ii) Sigmoid (Sig), and (iii) Softmax ($\mathsf{softmax}$).

(i) ReLU (ReLU): The ReLU function, $\mathsf{ReLU}(\mathsf{v}) = \max(0, \mathsf{v})$, can be written as

$$\mathsf{ReLU}(\mathsf{v}) = \begin{cases} 0, & \mathsf{v} < 0 \\ \mathsf{v} & 0 \leq \mathsf{v} \end{cases}$$

Thus, it can be viewed as $\mathsf{ReLU}(\mathsf{v}) = \bar{\mathsf{b}} \cdot \mathsf{v}$, where bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. Here $\bar{\mathsf{b}}$ denotes the complement of b. Given $[\![\mathsf{v}]\!]$, parties first extract the sign of v using the bit extraction protocol Π_{bitext}. The desired result can then be obtained using an invocation of the bit injection protocol Π_{bitInj}.

(ii) Sigmoid (Sig): The sigmoid function on value v is given as $\ln(\frac{1}{1+e^{-v}})$. However, computing the exact function is expensive in MPC and hence, we use the following MPC-friendly variant of the Sigmoid function [102, 101]:

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

Thus, $\mathsf{Sig}(\mathsf{v}) = 1 - \mathsf{b}_1\left(\mathsf{v} + \frac{1}{2}\right) + \mathsf{b}_2\left(\mathsf{v} - \frac{1}{2}\right)$, where $\mathsf{b}_1 = 1$ if $\mathsf{v} < -\frac{1}{2}$ and 0 otherwise, and $\mathsf{b}_2 = 1$ if $\mathsf{v} < \frac{1}{2}$ and 0 otherwise. Note that this can be viewed as an instance of a piecewise polynomial function.

(iii) Softmax ($\mathsf{softmax}$): Given a set of values, the softmax function is used to compute a probability distribution among the values such that each output is between 0 and 1, and all the outputs sum up to 1. This function is used at the output layer of the neural networks in Layer III of our architecture. For a set of d values, $\mathsf{v}_1, \ldots, \mathsf{v}_d$, the softmax on the ith value v_i is given as $\frac{e^{-\mathsf{v}_i}}{\sum_{j=1}^{d} e^{-\mathsf{v}_j}}$. Since the actual function is not MPC-friendly, we use the approximate variant of the same proposed by SecureML [102] and is defined as $\mathsf{softmax}(\mathsf{v}_i) = \frac{\mathsf{ReLU}(\mathsf{v}_i)}{\sum_{j=1}^{d} \mathsf{ReLU}(\mathsf{v}_j)}$. In order to perform the division, we switch from arithmetic to garbled world and then use a division garbled circuit.

Oblivious Selection Given $[\![\cdot]\!]$-shares of $\mathsf{x}_0, \mathsf{x}_1 \in \mathbb{Z}_{2^\ell}$ and $[\![\mathsf{b}]\!]^{\mathbf{B}}$ where $\mathsf{b} \in \{0, 1\}$, oblivious selection (Π_{obv}) enables parties to generate re-randomized $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{x}_\mathsf{b}$. The protocol is similar in spirit to the Oblivious Transfer primitive. Note that z can be written as $\mathsf{z} = \mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0) + \mathsf{x}_0$. To compute $[\![\cdot]\!]$-sharing of $\mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0)$, parties use an instance of piecewise

polynomial protocol $\Pi_{\mathsf{piecewise}}$ with $m = 1$. The $[\![\cdot]\!]$-share of z can then be obtained by adding the output of $\Pi_{\mathsf{piecewise}}$ with $[\![\mathsf{x}_0]\!]$.

Maximum / Minimum among two and three values The Π_{max2} protocol is used to compute the maximum among two values $\mathsf{x}_1, \mathsf{x}_2$ in a secure manner given $[\![\mathsf{x}_1]\!]$ and $[\![\mathsf{x}_2]\!]$. For this, the parties execute the secure comparison protocol on $[\![\mathsf{x}_1]\!], [\![\mathsf{x}_2]\!]$ to obtain $[\![\mathsf{b}]\!]^{\mathbf{B}} = [\![\mathsf{x}_1 > \mathsf{x}_2]\!]^{\mathbf{B}}$. Note that $\Pi_{\mathsf{max2}}(\mathsf{x}_1, \mathsf{x}_2) = \mathsf{b} \cdot (\mathsf{x}_1 - \mathsf{x}_2) + \mathsf{x}_2$ and can be computed using an instance of oblivious selection protocol Π_{obv}. The Π_{min2} protocol proceeds similarly except that $\Pi_{\mathsf{min2}}(\mathsf{x}_1, \mathsf{x}_2) = \mathsf{b} \cdot (\mathsf{x}_2 - \mathsf{x}_1) + \mathsf{x}_1$.

Given $[\![\cdot]\!]$-shares of $\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3$, the goal of the Π_{max3} protocol is to find the maximum value among the three. For this, first securely compare the pairs $(\mathsf{x}_1, \mathsf{x}_2), (\mathsf{x}_1, \mathsf{x}_3)$ and $(\mathsf{x}_2, \mathsf{x}_3)$ using the secure comparison protocol and obtain $[\![\mathsf{b}_1]\!]^{\mathbf{B}}, [\![\mathsf{b}_2]\!]^{\mathbf{B}}$ and $[\![\mathsf{b}_3]\!]^{\mathbf{B}}$ respectively. Here $\mathsf{b}_1 = 1$ if $\mathsf{x}_1 > \mathsf{x}_2$ and 0 otherwise. b_2 and b_3 are defined likewise . Now the maximum among the three, denoted by y, can be written as $\mathsf{y} = \mathsf{b}_1 \cdot \mathsf{b}_2 \cdot \mathsf{x}_1 + \overline{\mathsf{b}_1} \cdot \mathsf{b}_3 \cdot \mathsf{x}_2 + \overline{\mathsf{b}_2} \cdot \overline{\mathsf{b}_3} \cdot \mathsf{x}_3$. To compute this, parties can use Π_{dbitInj} to obtain each term in the expression for y and can locally add them to obtain the desired result. As an optimization, we can combine the communication in the online phase corresponding to all three executions of the Π_{dbitInj} protocol into one. The protocol for Π_{min3}, which computes the minimum among the three values can be obtained by slightly modifying the protocol for Π_{max3}. The difference lies in the expression for computing the minimum which will now be $\mathsf{y} = \overline{\mathsf{b}_1} \cdot \overline{\mathsf{b}_2} \cdot \mathsf{x}_1 + \mathsf{b}_1 \cdot \overline{\mathsf{b}_3} \cdot \mathsf{x}_2 + \mathsf{b}_2 \cdot \mathsf{b}_3 \cdot \mathsf{x}_3$.

ArgMin/ ArgMax Protocol Π_{argmin} (Fig. 6.3) allows parties to compute the index of the smallest element in a vector $\vec{\mathbf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_m)$ of m elements, where $\vec{\mathbf{x}}$ is $[\![\cdot]\!]$-shared, i.e. each element $\mathsf{x}_i \in \mathbb{Z}_{2^\ell}$ of $\vec{\mathbf{x}}$ is $[\![\cdot]\!]$-shared. The protocol outputs a $[\![\cdot]\!]^{\mathbf{B}}$-shared bit vector $\vec{\mathbf{b}}$ of size m which has a 1 at the index associated with the minimum value in $\vec{\mathbf{x}}$, and 0 elsewhere. We follow the standard tree-based approach [50] to recursively find the minimum value in $\vec{\mathbf{x}}$ while also updating $\vec{\mathbf{b}}$ to reflect the index of this smallest element. Each bit of $\vec{\mathbf{b}}$ is initialized to 1. The elements of $\vec{\mathbf{x}}$ are grouped into pairs and securely compared to find their pairwise minimum. Using this information, $\vec{\mathbf{b}}$ is updated such that b_j's are reset to 0 for x_j's $\in \vec{\mathbf{x}}$ which do not form the minimum in their respective pair; the other bits in $\vec{\mathbf{b}}$ still equal 1. The protocol recurses on the remaining elements $\mathsf{x}_j \in \vec{\mathbf{x}}$, which were the pairwise minimums. Eventually, only one $\mathsf{b}_j \in \vec{\mathbf{b}}$ equals 1, indicating that x_j is the minimum, with index j. Computing Π_{argmax} can be done similarly. The formal protocol appears in Fig. 6.3.

Protocol $\Pi_{\mathsf{argmin}}(\llbracket \vec{\mathbf{x}} \rrbracket)$

Let $\vec{\mathbf{b}}$ be the bit vector of size m, where m equals the size of $\vec{\mathbf{x}}$. Parties execute the following steps in the respective preprocessing and online phases.

1. If $m = 2$, do the following.
   (a) $\llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{bitext}}(\llbracket \mathsf{x}_1 \rrbracket, \llbracket \mathsf{x}_2 \rrbracket)$; $\llbracket \mathsf{d}_2 \rrbracket^{\mathbf{B}} = 1 \oplus \llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}}$; $\llbracket \mathsf{y} \rrbracket = \Pi_{\mathsf{obv}}(\llbracket \mathsf{x}_2 \rrbracket, \llbracket \mathsf{x}_1 \rrbracket, \llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}})$.
   (b) Return $(\llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{d}_2 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{y} \rrbracket)$.
2. Else, if $m = 3$, do the following
   (a) $\llbracket \mathsf{d}'_1 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{bitext}}(\llbracket \mathsf{x}_1 \rrbracket, \llbracket \mathsf{x}_2 \rrbracket)$; $\llbracket \mathsf{y}' \rrbracket = \Pi_{\mathsf{obv}}(\llbracket \mathsf{x}_2 \rrbracket, \llbracket \mathsf{x}_1 \rrbracket, \llbracket \mathsf{d}'_1 \rrbracket^{\mathbf{B}})$.
   (b) $\llbracket \mathsf{d}'_2 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{bitext}}(\llbracket \mathsf{y}' \rrbracket, \llbracket \mathsf{x}_3 \rrbracket)$; $\llbracket \mathsf{y} \rrbracket = \Pi_{\mathsf{obv}}(\llbracket \mathsf{x}_3 \rrbracket, \llbracket \mathsf{y}' \rrbracket, \llbracket \mathsf{d}'_2 \rrbracket^{\mathbf{B}})$.
   (c) $\llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{Mult}}(\llbracket \mathsf{d}'_1 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{d}'_2 \rrbracket^{\mathbf{B}})$; $\llbracket \mathsf{d}_2 \rrbracket^{\mathbf{B}} = \llbracket \mathsf{d}'_2 \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}}$; $\llbracket \mathsf{d}_3 \rrbracket^{\mathbf{B}} = 1 \oplus \llbracket \mathsf{d}'_1 \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{d}'_2 \rrbracket^{\mathbf{B}}$.
   (d) Return $(\llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{d}_2 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{d}_3 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{y} \rrbracket)$.
3. Else, let $\vec{\mathbf{x_1}} = (\mathsf{x}_1, \ldots, \mathsf{x}_{\lfloor m/2 \rfloor})$ and $\vec{\mathbf{x_2}} = (\mathsf{x}_{\lfloor m/2 \rfloor + 1}, \ldots, \mathsf{x}_m)$.
   (a) $\left(\llbracket \mathsf{d}_1 \rrbracket^{\mathbf{B}}, \ldots, \llbracket \mathsf{d}_{\lfloor m/2 \rfloor} \rrbracket^{\mathbf{B}}, \llbracket \mathsf{y}_1 \rrbracket\right) = \Pi_{\mathsf{argmin}}(\llbracket \vec{\mathbf{x_1}} \rrbracket)$.
   (b) $\left(\llbracket \mathsf{d}_{\lfloor m/2 \rfloor + 1} \rrbracket^{\mathbf{B}}, \ldots, \llbracket \mathsf{d}_m \rrbracket^{\mathbf{B}}, \llbracket \mathsf{y}_2 \rrbracket\right) = \Pi_{\mathsf{argmin}}(\llbracket \vec{\mathbf{x_2}} \rrbracket)$.
   (c) $\llbracket \mathsf{d} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{bitext}}(\llbracket \mathsf{y}_1 \rrbracket, \llbracket \mathsf{y}_2 \rrbracket)$; $\llbracket \mathsf{y} \rrbracket = \Pi_{\mathsf{obv}}(\llbracket \mathsf{y}_2 \rrbracket, \llbracket \mathsf{y}_1 \rrbracket, \llbracket \mathsf{d} \rrbracket^{\mathbf{B}})$.
   (d) $\llbracket \mathsf{b}_j \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{Mult}}(\llbracket \mathsf{d} \rrbracket^{\mathbf{B}}, \llbracket \mathsf{d}_j \rrbracket^{\mathbf{B}})$ for $j \in \{1, \ldots, \lfloor m/2 \rfloor\}$.
   (e) $\llbracket \mathsf{b}_j \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{Mult}}(1 \oplus \llbracket \mathsf{d} \rrbracket^{\mathbf{B}}, \llbracket \mathsf{d}_j \rrbracket^{\mathbf{B}})$ for $j \in \{\lfloor m/2 \rfloor + 1, \ldots, m\}$.
   (f) Return $\left(\llbracket \mathsf{b}_1 \rrbracket^{\mathbf{B}}, \ldots, \llbracket \mathsf{b}_m \rrbracket^{\mathbf{B}}, \llbracket \mathsf{y} \rrbracket\right)$.

Figure 6.3: Protocol to find index of smallest element in $\vec{\mathbf{x}}$

To begin with, parties initialize $\mathsf{b}_j = 1$ for $\mathsf{b}_j \in \vec{\mathbf{b}}$ by locally setting $\mathsf{m}_{\mathsf{b}_j} = 1$ and $\lambda^1_{\mathsf{b}_j} = \lambda^2_{\mathsf{b}_j} = \lambda^3_{\mathsf{b}_j} = 0$. The minimum, y_{ij}, of two elements, $\mathsf{x}_i, \mathsf{x}_j$ can be computed as: one invocation of bit extraction protocol to obtain $\llbracket \cdot \rrbracket^{\mathbf{B}}$-sharing of b_{ij}, where $\mathsf{b}_{ij} = 1$ if $\mathsf{x}_i < \mathsf{x}_j$, and $\mathsf{b}_{ij} = 0$ otherwise; one invocation of oblivious selection protocol $\Pi_{\mathsf{obv}}(\mathsf{x}_j, \mathsf{x}_i, \mathsf{b}_{ij})$, which outputs $\llbracket \cdot \rrbracket$-shares of $\mathsf{y}_{ij} = \mathsf{x}_j$ if $\mathsf{b}_{ij} = 0$, and $\mathsf{y}_{ij} = \mathsf{x}_i$, otherwise. To update $\vec{\mathbf{b}}$ to reflect the pairwise minimums, we view the elements $\mathsf{x}_j \in \vec{\mathbf{x}}$ as the leaves of a binary tree, in a bottom-up manner. For two elements in a pair, say $(\mathsf{x}_i, \mathsf{x}_j)$, whose pairwise minimum is y_{ij}, we let y_{ij} be the root node with x_i as its left child and x_j as its right child. Now, to update $\vec{\mathbf{b}}$, parties multiply b_{ij} with the bits in $\vec{\mathbf{b}}$ associated with the *left-reachable leaf nodes*, which comprise of all the leaf nodes (elements of $\vec{\mathbf{x}}$) that are reachable through the left child of the root. Similarly, parties multiply $1 \oplus \mathsf{b}_{ij}$

with the bits in $\vec{\mathsf{b}}$ associated with the *right-reachable leaf nodes*, which comprise of all the leaf nodes (elements of $\vec{\mathsf{x}}$) that are reachable through the right child of the root. Thus, if $\mathsf{b}_{ij} = 1$ indicating that $\mathsf{x}_i < \mathsf{x}_j$, b_i remains 1 as it gets multiplied by $\mathsf{b}_{ij} = 1$ while b_j gets reset to 0 as it gets multiplied by $1 \oplus \mathsf{b}_{ij} = 0$. The case for $\mathsf{b}_{ij} = 0$ holds for similar reasons. Given the values y_{ij} for the next level, and the updated $\vec{\mathsf{b}}$, the steps are applied recursively until the minimum element is obtained.

The protocol Π_{argmax} which allows the parties to compute the index of the largest element in a $[\![\cdot]\!]$-shared vector $\vec{\mathbf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_m)$, is similar to Π_{argmin} with the following difference. To find the maximum among two elements $([\![\mathsf{x}_i]\!], [\![\mathsf{x}_j]\!])$, parties run the bit extraction protocol to obtain $[\![\mathsf{b}_{ij}]\!]^{\mathbf{B}}$ as before, followed by $\Pi_{\mathsf{obv}}(\mathsf{x}_i, \mathsf{x}_j, \mathsf{b}_{ij})$, which outputs $[\![\cdot]\!]$-shares of $\mathsf{y}_{ij} = \mathsf{x}_i$ if $\mathsf{b}_{ij} = 0$, and $\mathsf{y}_{ij} = \mathsf{x}_j$, otherwise. Now, $\vec{\mathsf{b}}$ is updated in each level by multiplying $1 \oplus \mathsf{b}_{ij}$ with the bits in $\vec{\mathsf{b}}$ associated with the *left-reachable leaf nodes* (as described before) and multiplying b_{ij} with the bits in $\vec{\mathsf{b}}$ associated with the *right-reachable leaf nodes*.

Mixed-world Conversions The protocols for mixed-world conversions enable efficient transitions among the arithmetic, boolean, and garbled worlds. The efficiency lift of our framework compared to existing frameworks stands on the following useful observation– a large portion of computation in most of the MPC-based PPML framework is done over the arithmetic and boolean world; they switch to the garbled world to perform the non-linear operations (e.g. softmax) that are expensive in the arithmetic/boolean world and switch back to the arithmetic/boolean world immediately after. We leverage this phenomenon to construct *end-to-end* conversion techniques such as Arithmetic-Garbled-Arithmetic. The standard approach until now was to perform a *piece-wise* combination of *Arithmetic to Garbled* followed by a *Garbled to Arithmetic* conversion. End-to-end conversions benefit from not having to generate a full-fledged garbled-shared output after the computation. Instead, these conversions aim to produce a "partial" garbled-shared output that is enough to lead to an arithmetic sharing of the output. This results in end-to-end conversions of the form "x-Garbled-x" where x can be either arithmetic or boolean that need just a single round for our garbled world as opposed to the two in the existing works [101, 38].

Chapter 7

ASTRA: Semi-honest Blocks

This chapter provides details for the Layer II blocks of our 2PC framework ASTRA. Details for the Layer I blocks are provided in chapter 3.

7.1 Building Blocks

7.1.1 Dot Product (Scalar Product)

Given $[\![\vec{a}]\!], [\![\vec{b}]\!]$ with $|\vec{a}| = |\vec{b}| = d$, protocol Π_{dotp} (Fig. 7.1) computes $[\![z]\!]$ such that $z = (\vec{a} \odot \vec{b})^t$ if truncation is enabled, else $z = \vec{a} \odot \vec{b}$. For this, we combine the partial products from the multiplication protocol across d multiplications and communicate them in a single shot. This makes the communication cost of the dot product independent of the vector size.

Protocol $\Pi_{\mathsf{dotp}}(\vec{a}, \vec{b}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\vec{a}]\!], [\![\vec{b}]\!]$.
Output: $[\![o]\!]$ where $o = z^t$ if $\mathsf{isTr} = 1$ and $o = z$ if $\mathsf{isTr} = 0$ and $z = \vec{a} \odot \vec{b} = \sum_{i=1}^{d} a_i b_i$.

Preprocessing: Let $\gamma_{\vec{ab}} = \sum_{i=1}^{d} \gamma_{a_i b_i}$.

1. P_0, P_j sample $u^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $u^1 + u^2 = \gamma_{\vec{ab}} - r$ for $r \in_R \mathbb{Z}_{2^\ell}$.
2. Party P_0: Computes $r = \gamma_{\vec{ab}} - u^1 - u^2$. If $\mathsf{isTr} = 1$, sets $q = r^t$, else $q = r$.
   Executes $\Pi_{\mathsf{Sh}}(P_0, q)$ to generate $[\![q]\!]$.

Online: Let $y = (z - r) - \sum_{i=1}^{d} m_{a_i b_i}$.

1. Compute: $P_1 : y_1 = \sum_{i=1}^{d} (-\lambda^1_{a_i} m_{b_i} - \lambda^1_{b_i} m_{a_i}) + u^1$, $P_2 : y_2 = \sum_{i=1}^{d} (-\lambda^2_{a_i} m_{b_i} - \lambda^2_{b_i} m_{a_i}) + u^2$

2. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z}-\mathsf{r} = \mathsf{y}_1+\mathsf{y}_2+\sum_{i=1}^{\mathsf{d}} \mathsf{m}_{\mathsf{a}_i\mathsf{b}_i}$.

3. P_1, P_2: If $\mathsf{isTr}=1$, set $\mathsf{p}=(\mathsf{z}-\mathsf{r})^{\mathsf{t}}$, else $\mathsf{p}=\mathsf{z}-\mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1,P_2,\mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.

4. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o}=\mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr}=1$ and z otherwise.

Figure 7.1: Dot Product with / without Truncation in ASTRA.

Lemma 7.1 (Communication) *Protocol Π_{dotp} (Fig. 7.1) (in ASTRA) requires ℓ bits of communication in preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

7.1.2 Bit Extraction

To compute most significant bit (msb) of the value v, note that $\mathsf{v}=\mathsf{m}_{\mathsf{v}}+(-\lambda_{\mathsf{v}})$ as per the sharing semantics (cf. Table 3.2). P_0 generates the boolean sharing of $-\lambda_{\mathsf{v}}$ during the preprocessing, while P_1, P_2 generate $[\![\mathsf{m}_{\mathsf{v}}]\!]^{\mathbf{B}}$ during the online phase using joint sharing protocol. Parties compute the result by evaluating the bit extraction circuit [101, 113].

7.1.3 Bit to Arithmetic

Protocol $\Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$ (Fig. 7.2) enables computing $[\![\mathsf{b}]\!]$ of a bit b given its boolean sharing $[\![\mathsf{b}]\!]^{\mathbf{B}}$. Let b^{R} denotes the value of $\mathsf{b}\in\{0,1\}$ over the arithmetic ring $\mathbb{Z}_{2^\ell}$. Using our sharing semantics,

$$\mathsf{b}^{\mathsf{R}} = (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}} = \mathsf{m}_{\mathsf{b}}^{\mathsf{R}} + \lambda_{\mathsf{b}}^{\mathsf{R}}(1-2\mathsf{m}_{\mathsf{b}}^{\mathsf{R}}) \tag{7.1}$$

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

Input(s): $[\![\mathsf{b}]\!]^{\mathbf{B}}$, **Output:** $[\![\mathsf{y}]\!] = [\![\mathsf{b}^{\mathsf{R}}]\!]$.

Preprocessing: P_0, P_1 sample random $[\lambda_{\mathsf{b}}^{\mathsf{R}}]_1 \in \mathbb{Z}_{2^\ell}$. P_0 sends $[\lambda_{\mathsf{b}}^{\mathsf{R}}]_2 = \lambda_{\mathsf{b}}^{\mathsf{R}} - [\lambda_{\mathsf{b}}^{\mathsf{R}}]_1$ to P_2.

Online:

1. Locally compute: $P_1 : \mathsf{y}_1 = \mathsf{m}_{\mathsf{b}}^{\mathsf{R}} + [\lambda_{\mathsf{b}}^{\mathsf{R}}]_1 (1-2\mathsf{m}_{\mathsf{b}}^{\mathsf{R}})$ | $P_2 : \mathsf{y}_2 = [\lambda_{\mathsf{b}}^{\mathsf{R}}]_2 (1-2\mathsf{m}_{\mathsf{b}}^{\mathsf{R}})$

2. P_i for $i\in\{1,2\}$ executes Π_{Sh} on y_i to generate the respective $[\![\cdot]\!]$-shares.

3. Compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}_1]\!] + [\![\mathsf{y}_2]\!]$.

Figure 7.2: Bit to Arithmetic conversion in ASTRA.

During the preprocessing, P_0 generates $[\cdot]$-sharing of $\lambda_{\mathsf{b}}^{\mathsf{R}}$. The online phase consists of each P_1 and P_2 locally computing an additive sharing of b^{R}, generating the corresponding $[\![\cdot]\!]$-sharing

using Π_{Sh}, and locally adding the shares to obtain $[\![\mathsf{b}]\!]$.

Lemma 7.2 (Communication) *Protocol* Π_{bit2A} *(Fig. 7.2) requires* ℓ *bits of communication in preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

Proof: During preprocessing, generation of $\left[\lambda_{\mathsf{b}}^{\mathsf{R}}\right]$ involves communication of ℓ bits from P_0 to P_2. The online phase involves two instances of arithmetic sharing protocol in parallel, resulting in 1 round and a communication of 2ℓ bits. □

7.1.3.1 Bit to Arithmetic:II

Similar to Π_{bit2A} protocol, given the boolean sharings $[\![\mathsf{b}_1]\!]^{\mathbf{B}}$, $[\![\mathsf{b}_2]\!]^{\mathbf{B}}$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1-2\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}})$, $(1-2\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}})$ respectively. Using (7.1), we can write

$$\begin{aligned}(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}} = (\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_1})(\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_2}) \\ &= \mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_1} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \end{aligned} \tag{7.2}$$

During preprocessing, the $[\cdot]$-shares of $\lambda_{\mathsf{b}_1}^{\mathsf{R}}$, $\lambda_{\mathsf{b}_2}^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$ are computed similar to that of Π_{bit2A} (Fig. 7.2). The online phase is similar to that of Π_{bit2A} protocol.

Lemma 7.3 (Communication) *Protocol* Π_{dbit2A} *requires* 3ℓ *bits of communication in preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

7.1.4 Bit Injection

Given the boolean 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 $\mathsf{b}^{\mathsf{R}}\mathsf{v}$. Let Δ_{b} denote the value $(1-2\mathsf{m}_{\mathsf{b}}^{\mathsf{R}})$. Similar to Π_{bit2A},

$$\begin{aligned}\mathsf{b}^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) = (\mathsf{m}_{\mathsf{b}}^{\mathsf{R}} + \lambda_{\mathsf{b}}^{\mathsf{R}}\Delta_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\ &= \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}} - \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}} + \lambda_{\mathsf{b}}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}} - \lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}\Delta_{\mathsf{b}} \end{aligned} \tag{7.3}$$

During the preprocessing, P_0 generates the $[\cdot]$-shares of $\lambda_{\mathsf{b}}^{\mathsf{R}}$ and $\lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}$ similar to Π_{bit2A} protocol. During the online phase, P_1 and P_2 compute an additive sharing of $\mathsf{b}^{\mathsf{R}}\mathsf{v}$ and execute Π_{Sh} on them to generate the respctive $[\![\cdot]\!]$-shares.

Lemma 7.4 (Communication) *Protocol* Π_{bitInj} *requires* 2ℓ *bits of communication in preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

7.1.4.1 Sum of Bit Injections

Given m pair of values in the shared form, $\{[\![\mathsf{b}_i]\!]^{\mathbf{B}}, [\![\mathsf{v}_i]\!]\}_{i\in[m]}$, the goal of Π_{bitInjS} is to compute the $[\![\cdot]\!]$-share of $\mathsf{z} = \sum_{i=1}^{m} \mathsf{b}_i^{\mathsf{R}} \cdot \mathsf{v}_i$. For this, parties execute the preprocessing corresponding to m bit injections of the form $\mathsf{b}_i^{\mathsf{R}} \cdot \mathsf{v}_i$.

In the online phase, each of P_1 and P_2 locally compute an additive sharing of z_i, corresponding to $\mathsf{b}_i^{\mathsf{R}} \cdot \mathsf{v}_i$ first. Instead of generating the $[\![\cdot]\!]$-sharing for each of the m terms, parties locally add the shares and execute Π_{Sh} on the result. Concretely, parties locally compute $\mathsf{z}^j = \sum_{i=1}^{m} \mathsf{z}_i^j$ for $j \in \{1, 2\}$ and execute Π_{Sh} on z^j to obtain its $[\![\cdot]\!]$-sharing. This results in an online communication independent of m.

Lemma 7.5 (Communication) *Protocol Π_{bitInjS} requires $m \cdot 2\ell$ bits of communication in preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

7.1.4.2 Bit Injection:II

Similar to Π_{bitInj} protocol, given $[\![\mathsf{b}_1]\!]^{\mathbf{B}}, [\![\mathsf{b}_2]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1 - 2\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}})$, $(1 - 2\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}})$ respectively. Using (7.2) and (7.3), we can write

$$\begin{aligned}
(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\
&= (\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_1})(\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_2})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\
&= \mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_1} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \\
&\quad - \lambda_{\mathsf{v}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} - \lambda_{\mathsf{b}_1}^{\mathsf{R}}\lambda_{\mathsf{v}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_1} - \lambda_{\mathsf{b}_2}^{\mathsf{R}}\lambda_{\mathsf{v}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_2} - (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_{\mathsf{v}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \qquad (7.4)
\end{aligned}$$

During the preprocessing, P_0 generates the $[\cdot]$-shares of $\lambda_{\mathsf{b}_1}^{\mathsf{R}}$, $\lambda_{\mathsf{b}_2}^{\mathsf{R}}$, $\lambda_{\mathsf{b}_1}^{\mathsf{R}}\lambda_{\mathsf{v}}$, $\lambda_{\mathsf{b}_2}^{\mathsf{R}}\lambda_{\mathsf{v}}$, $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_{\mathsf{v}}$ similar to Π_{bit2A} protocol. The online phase is similar to that of Π_{bitInj} protocol.

Lemma 7.6 (Communication) *Protocol Π_{dbitInj} requires 6ℓ bits of communication in preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

7.1.5 Equality Test (Π_{eq})

To check whether $\mathsf{a} \stackrel{?}{=} \mathsf{b}$ or not, given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$, Π_{eq} proceeds with parties locally computing $[\![\mathsf{y}]\!] = [\![\mathsf{a}]\!] - [\![\mathsf{b}]\!]$. According to our sharing semantics, y can be written as $\mathsf{y} = \mathsf{y}_1 - \mathsf{y}_2$ where $\mathsf{y}_1 = \mathsf{m}_{\mathsf{y}}$ and $\mathsf{y}_2 = \lambda_{\mathsf{y}}$. P_0 generates $[\![\mathsf{y}_2]\!]^{\mathbf{B}}$ during the preprocessing while P_1, P_2 generate $[\![\mathsf{y}_1]\!]^{\mathbf{B}}$ in the online using Π_{JSh}. Note that $\mathsf{a} = \mathsf{b}$ implies $\mathsf{y}_1 = \mathsf{y}_2$ and hence all the bits of $\mathsf{v} = \overline{(\mathsf{y}_1 \oplus \mathsf{y}_2)}$

should be 1. As mentioned in the introduction of Part II (II), parties use four input AND gates and a tree structure, where 4 bits are taken at a time and the AND of them is computed in one go.

7.2 Mixed Protocol Framework

Table 7.1 compares our sharing conversions with ABY3 [101]. For uniformity, we consider a function, F, to be computed on an ℓ-bit inputs x, y using a garbled circuit (GC) in the mixed framework, which gives an ℓ-bit output $\mathsf{z} = \mathsf{F}(\mathsf{x}, \mathsf{y})$, where ℓ denotes the ring size in bits. Let G^{F} denote the corresponding GC. In the table, G^{Sn} denotes a n-input garbled subtraction circuit; G^{An} denotes n-input garbled addition circuit; $\hat{\mathsf{G}}$ denotes the garbled circuit with decoding information; $\mathsf{G}^{\mathsf{n}_1\times 1,\ldots,\mathsf{n}_m\times \mathsf{m}}$ denotes n_i instances of GC G^{i} for $i \in \{1,\ldots,\mathsf{m}\}$ and $|\mathsf{G}^{\mathsf{n}_1\times 1,\ldots,\mathsf{n}_m\times \mathsf{m}}|$ denotes its size.

Variant[a]	Conversion[b]	ABY3 [101]			ASTRA		
		$\text{Comm.}_{\text{pre}}$	Comm._{on}	$\text{Rounds}_{\text{on}}$	$\text{Comm.}_{\text{pre}}$	Comm._{on}	$\text{Rounds}_{\text{on}}$
2 GC	A-G-A	$2\ell\kappa + 2\lvert\hat{\mathsf{G}}^{2\times\mathsf{A2},\mathsf{S2},\mathsf{F}}\rvert$	$10\ell\kappa$	2	$2\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{A2},\mathsf{F}}\rvert$	$4\ell\kappa$	1
	A-G-B	$2\lvert\mathsf{G}^{2\times\mathsf{A2},\mathsf{F}}\rvert$	$8\ell\kappa + 2\ell$		$(6\ell\kappa+\ell)$ $2\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert$		
	B-G-A	$2\ell\kappa + 2\lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert$	$10\ell\kappa$		$+$ $2\lvert\hat{\mathsf{G}}^{\mathsf{A2},\mathsf{F}}\rvert$		
	B-G-B	$2\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$8\ell\kappa + 2\ell$		$2\lvert\hat{\mathsf{G}}^{\mathsf{F}}\rvert$		
1 GC	A-G-A	$\ell\kappa + \lvert\hat{\mathsf{G}}^{2\times\mathsf{A2},\mathsf{S2},\mathsf{F}}\rvert$	$5\ell\kappa$	2	$\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{A2},\mathsf{F}}\rvert$	$2\ell\kappa + \ell$	2
	A-G-B	$\lvert\mathsf{G}^{2\times\mathsf{A2},\mathsf{F}}\rvert$	$4\ell\kappa + \ell$		$(3\ell\kappa+\ell)$ $\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert$		
	B-G-A	$\ell\kappa + \lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert$	$5\ell\kappa$		$+$ $\lvert\hat{\mathsf{G}}^{\mathsf{A2},\mathsf{F}}\rvert$		
	B-G-B	$\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$4\ell\kappa + \ell$		$\lvert\hat{\mathsf{G}}^{\mathsf{F}}\rvert$		
Others[c]	A-B	–	$3\ell + 3\ell\log\ell$	$1 + \log\ell$	$\mathsf{u}_1 + \ell$	$2\mathsf{u}_2$	$\log_4 \ell$
	B-A	–	$3\ell + 3\ell\log\ell$	$1 + \log\ell$	ℓ^2	2ℓ	1

[a] Notations: ℓ - size of ring in bits, κ - computational security parameter, 'pre' - preprocessing, 'on' - online.
[b] 'A' - arithmetic, 'B' - boolean, 'G' - Garbled.
[c] $\mathsf{u}_1 = \mathsf{n}_2 + 4\mathsf{n}_3 + 11\mathsf{n}_4$, $\mathsf{u}_2 = \mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4$ denote the number of AND gates in the optimized adder circuit [113] with 2, 3, 4 inputs, respectively. For $\ell = 64$, $\mathsf{n}_2 = 216, \mathsf{n}_3 = 184, \mathsf{n}_4 = 179$.

Table 7.1: Mixed protocol conversions of ABY3 [101] and ASTRA.

7.2.1 Conversions involving Garbled World

Assume the GC is required to compute a function f on inputs $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and let the output be $f(\mathsf{x}, \mathsf{y})$. All the conversions described are for the 2 GC variant. Conversions for the 1 GC variant are straightforward, hence we omit the details.

Case I: Boolean-Garbled-Boolean Since the inputs to the GC are available in boolean form, say $[\![\mathsf{x}]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!]^{\mathbf{B}}$, parties generate $[\![\mathsf{x}]\!]^{\mathbf{C}}, [\![\mathsf{y}]\!]^{\mathbf{C}}$ by invoking the garbled sharing protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$. Additionally, P_0 samples $\mathsf{R} \in \mathbb{Z}_{2^\ell}$ to mask the function output, $f(\mathsf{x},\mathsf{y})$, and generate $[\![\mathsf{R}]\!]^{\mathbf{B}}$ and $[\![\mathsf{R}]\!]^{\mathbf{G}}$. Garblers $P_g \in \{P_0, P_2\}$ garble the circuit which computes $\mathsf{z} = f(\mathsf{x},\mathsf{y}) \oplus \mathsf{R}$, and send the GC along with the decoding information to evaluator P_1. Analogous steps are performed for evaluator P_2. Upon GC evaluation and output decoding, evaluators obtain $\mathsf{z} = f(\mathsf{x},\mathsf{y}) \oplus \mathsf{R}$, and jointly boolean share z to generate $[\![\mathsf{z}]\!]^{\mathbf{B}}$. Parties then compute $[\![f(\mathsf{x},\mathsf{y})]\!]^{\mathbf{B}} = [\![\mathsf{z}]\!]^{\mathbf{B}} \oplus [\![\mathsf{R}]\!]^{\mathbf{B}}$.

Case II: Boolean-Garbled-Arithmetic This is similar to *Case I* except that the circuit which computes $\mathsf{z} = f(\mathsf{x},\mathsf{y}) + \mathsf{R}$ is garbled instead. Boolean sharing of z is replaced with arithmetic, followed by computing $[\![f(\mathsf{x},\mathsf{y})]\!] = [\![\mathsf{z}]\!] - [\![\mathsf{R}]\!]$.

Cases III & IV: Input in Arithmetic Sharing The function to be computed $f(\mathsf{x},\mathsf{y})$, is modified as $f'(\mathsf{m}_\mathsf{x}, \lambda_\mathsf{x}, \mathsf{m}_\mathsf{y}, \lambda_\mathsf{y}) = f(\mathsf{m}_\mathsf{x} - \lambda_\mathsf{x}, \mathsf{m}_\mathsf{y} - \lambda_\mathsf{y})$ where inputs x,y are replaced by the pairs $\{\mathsf{m}_\mathsf{x}, \lambda_\mathsf{x}\}, \{\mathsf{m}_\mathsf{y}, \lambda_\mathsf{y}\}$. The circuit to be garbled thus, corresponds to the function f'. Parties generate $[\![\mathsf{m}_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda_\mathsf{x}]\!]^{\mathbf{G}}, [\![\mathsf{m}_\mathsf{y}]\!]^{\mathbf{G}}, [\![\lambda_\mathsf{y}]\!]^{\mathbf{G}}$ via $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$, following which, parties proceed with the rest of the computation whose steps are similar to *Case I*, and *II*, depending on the requirement on the output sharing.

7.2.2 Other Conversions

Arithmetic to Boolean To convert arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to boolean sharing, observe that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$ where $\mathsf{v}_1 = \mathsf{m}_\mathsf{v}$ is possessed by parties P_1, P_2, while $\mathsf{v}_2 = -\lambda_\mathsf{v}$ is possessed by P_0. Thus, $[\![\mathsf{v}]\!]^{\mathbf{B}}$ can be computed as $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} + [\![\mathsf{v}_2]\!]^{\mathbf{B}}$. For this, P_0 can generate $[\![\mathsf{v}_2]\!]^{\mathbf{B}}$ in the preprocessing, and $[\![\mathsf{v}_1]\!]^{\mathbf{B}}$ can be generated in the online by P_1, P_2 using joint sharing protocol. The protocol appears in Fig. 7.3. Boolean addition, when instantiated using the adder of ABY2.0 [113], requires $\log_4(\ell)$ rounds.

Protocol Π_{A2B}

Preprocessing: P_0 generates $[\![\mathsf{v}_2]\!]^{\mathbf{B}}$ using Π_{Sh}, where $\mathsf{v}_2 = -\lambda_\mathsf{v}$.

Online:

1. P_1, P_2 execute joint boolean sharing to generate $[\![\mathsf{v}_1]\!]^{\mathbf{B}}$, where $\mathsf{v}_1 = \mathsf{m}_\mathsf{v}$.
2. Parties obtain $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} + [\![\mathsf{v}_2]\!]^{\mathbf{B}}$ using a boolean adder circuit.

Figure 7.3: Arithmetic to Boolean Conversion in ASTRA.

Boolean to Arithmetic To convert a boolean sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ into an arithmetic sharing, note that

$$\mathsf{v} = \sum_{i=0}^{\ell-1} 2^i \mathsf{v}[i] = \sum_{i=0}^{\ell-1} 2^i (\lambda_{\mathsf{v}[i]} \oplus \mathsf{m}_{\mathsf{v}[i]}) = \sum_{i=0}^{\ell-1} 2^i \left(\mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]} + \lambda^{\mathsf{R}}_{\mathsf{v}[i]} (1 - 2\mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}) \right)$$

where $\lambda^{\mathsf{R}}_{\mathsf{v}[i]}, \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$ denote the arithmetic value of bits $\lambda_{\mathsf{v}[i]}, \mathsf{m}_{\mathsf{v}[i]}$ over the ring $\mathbb{Z}_{2^\ell}$. For each bit $\mathsf{v}[i]$ of v, P_0 generates the $[\cdot]$-shares of ${\lambda_{\mathsf{v}[i]}}^{\mathsf{R}}$ in the preprocessing, similar to Π_{bit2A} (Fig. 7.2). During the online phase, additive shares for each bit $\mathsf{v}[i]$ are locally computed similar to Π_{bit2A}. Parties then multiply the ith share with 2^i and locally add up to obtain an additive sharing of v. The rest of the steps are similar to Π_{bit2A}, and the formal protocol appears in Fig. 7.4.

Protocol $\Pi_{\mathsf{B2A}}(\mathcal{P}, [\![\mathsf{v}]\!]^{\mathbf{B}})$

Let $\mathsf{v}[i]$ denote the ith bit of v. Let $\mathsf{p}_i = \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$, and $\mathsf{q}_i = \lambda^{\mathsf{R}}_{\mathsf{v}[i]}$.

Preprocessing:

1. For $i \in \{0, 1, \ldots, \ell-1\}$, execute the preprocessing of Π_{bit2A} (Fig. 7.2) for each bit $\mathsf{v}[i]$, to generate $[\mathsf{q}_i] = ([\mathsf{q}_i]_1, [\mathsf{q}_i]_2)$.

Online: Let $\mathsf{y}_i = (\mathsf{v}[i])^{\mathsf{R}}$ and y denotes the arithmetic equivalent of v.

1. Locally compute:

$$P_1 : \mathsf{y}^1 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}^1_i = \sum_{i=0}^{\ell-1} 2^i (\mathsf{p}_i + [\mathsf{q}_i]_1 (1 - 2\mathsf{p}_i))$$
$$P_2 : \mathsf{y}^2 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}^2_i = \sum_{i=0}^{\ell-1} 2^i ([\mathsf{q}_i]_2 (1 - 2\mathsf{p}_i))$$

2. P_j for $j \in \{1, 2\}$ executes Π_{Sh} on y^j to generate the respective $[\![\cdot]\!]$-shares.
3. Compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}^1]\!] + [\![\mathsf{y}^2]\!]$.

Figure 7.4: Boolean to Arithmetic Conversion in ASTRA.

Chapter 8

SWIFT: 3PC Fair and Robust Blocks

This chapter provides details for the Layer II blocks of our 2PC framework SWIFT. Details for the Layer I blocks are provided in chapter 4. The robust constructions of the blocks are detailed in this chapter, and the fair variants can be derived easily.

8.1 Building Blocks

8.1.1 Dot Product (Scalar Product)

Given $[\![\vec{\mathsf{a}}]\!], [\![\vec{\mathsf{b}}]\!]$ with $|\vec{\mathsf{a}}| = |\vec{\mathsf{b}}| = \mathsf{d}$, protocol Π_{dotp} (Fig. 8.1) computes $[\![\mathsf{z}]\!]$ such that $\mathsf{z} = (\vec{\mathsf{a}} \odot \vec{\mathsf{b}})^{\mathsf{t}}$ if truncation is enabled, else $\mathsf{z} = \vec{\mathsf{a}} \odot \vec{\mathsf{b}}$. For this, we combine the partial products from the multiplication protocol across d multiplications and communicate them in a single shot. This makes the communication cost of the dot product independent of the vector size.

Protocol $\Pi_{\mathsf{dotp}}(\vec{\mathsf{a}}, \vec{\mathsf{b}}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\vec{\mathsf{a}}]\!], [\![\vec{\mathsf{b}}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \vec{\mathsf{a}} \odot \vec{\mathsf{b}} = \sum_{i=1}^{\mathsf{d}} \mathsf{a}_i \mathsf{b}_i$.

Preprocessing: Let $\lambda_{\vec{\mathsf{a}}} = \{\lambda_{\mathsf{a}_i}\}_{i \in [\mathsf{d}]}$, $\lambda_{\vec{\mathsf{b}}} = \{\lambda_{\mathsf{b}_i}\}_{i \in [\mathsf{d}]}$ and $\gamma_{\vec{\mathsf{a}}\vec{\mathsf{b}}} = \sum_{i=1}^{\mathsf{d}} \gamma_{\mathsf{a}_i \mathsf{b}_i}$.

1. Invoke $\mathcal{F}_{\mathsf{dotpPre}}$ on $\langle \lambda_{\vec{\mathsf{a}}} \rangle$ and $\langle \lambda_{\vec{\mathsf{b}}} \rangle$ to obtain $\langle \gamma_{\vec{\mathsf{a}}\vec{\mathsf{b}}} \rangle$.
2. If $\mathsf{isTr} = 0$:
   (a) Local computation of $\langle \mathsf{r} \rangle$: $\mathcal{P} \setminus \{P_2\} \leftarrow_R \mathsf{r}^1$; $\mathcal{P} \setminus \{P_1\} \leftarrow_R \mathsf{r}^2$; $\mathcal{P} \setminus \{P_3\} \leftarrow_R \mathsf{r}^3$.
   (b) Local computation of $[\![\mathsf{r}]\!]$: $\lambda_{\mathsf{r}}^1 = -\mathsf{r}^1$, $\lambda_{\mathsf{r}}^2 = -\mathsf{r}^2$, $\lambda_{\mathsf{r}}^3 = -\mathsf{r}^3$, $\mathsf{m}_{\mathsf{r}} = 0$. Set $[\![\mathsf{q}]\!] = [\![\mathsf{r}]\!]$.

3. If $\mathsf{isTr} = 1$, invoke Π_{trgen} (Fig. 8.4) to generate $(\langle \mathsf{r} \rangle, [\![\mathsf{r}^{\mathsf{t}}]\!])$. Set $[\![\mathsf{q}]\!] = [\![\mathsf{r}^{\mathsf{t}}]\!]$.

4. Locally compute $\langle(\gamma_{\vec{\mathbf{ab}}} - \mathsf{r})\rangle = \langle \gamma_{\vec{\mathbf{ab}}} \rangle - \langle \mathsf{r} \rangle$.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \sum_{i=1}^{\mathsf{d}} \mathsf{m}_{\mathsf{a}_i \mathsf{b}_i}$.

1. Parties locally compute the following:

$$P_1, P_3 : \mathsf{y}_1 = \sum_{i=1}^{\mathsf{d}} (-\lambda^1_{\mathsf{a}_i} \mathsf{m}_{\mathsf{b}_i} - \lambda^1_{\mathsf{b}_i} \mathsf{m}_{\mathsf{a}_i}) + (\gamma_{\vec{\mathbf{ab}}} - \mathsf{r})^1$$

$$P_2, P_3 : \mathsf{y}_2 = \sum_{i=1}^{\mathsf{d}} (-\lambda^2_{\mathsf{a}_i} \mathsf{m}_{\mathsf{b}_i} - \lambda^2_{\mathsf{b}_i} \mathsf{m}_{\mathsf{a}_i}) + (\gamma_{\vec{\mathbf{ab}}} - \mathsf{r})^2$$

$$P_1, P_2 : \mathsf{y}_3 = \sum_{i=1}^{\mathsf{d}} (-\lambda^3_{\mathsf{a}_i} \mathsf{m}_{\mathsf{b}_i} - \lambda^3_{\mathsf{b}_i} \mathsf{m}_{\mathsf{a}_i}) + (\gamma_{\vec{\mathbf{ab}}} - \mathsf{r})^3$$

2. P_1, P_3 jsnd y_1 to P_2, while P_2, P_3 jsnd y_2 to P_1. They locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \sum_{i=1}^{\mathsf{d}} \mathsf{m}_{\mathsf{a}_i \mathsf{b}_i}$.

3. P_1, P_2: If $\mathsf{isTr} = 1$, set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. Execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.

4. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 8.1: Dot Product with / without Truncation in SWIFT.

Analogous to the multiplication protocol, dot product offloads one call to a robust dot product protocol Π_{MultPre} to the preprocessing. By extending techniques of [24, 27], we give an instantiation for the dot product protocol used in our preprocessing whose (amortized) communication cost is constant, thereby making our preprocessing cost also *independent* of d. The ideal world functionality $\mathcal{F}_{\mathsf{dotpPre}}$ for realizing Π_{dotpPre} is presented in Fig. 8.2.

Instantiating $\mathcal{F}_{\mathsf{dotpPre}}$: A trivial way to instantiate Π_{dotpPre} is to treat a dot product operation as d multiplications. However, this results in a communication cost that is linearly dependent on the feature size. Instead, we instantiate Π_{dotpPre} by a semi-honest dot product protocol followed by a verification phase to check the correctness. For the verification phase, we extend the techniques of [24, 27] to provide support for verification of dot product tuples. Setting the verification phase parameters appropriately gives a Π_{dotpPre} whose (amortized) communication cost is independent of the feature size. We will provide the details next.

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

$\mathcal{F}_{\mathsf{dotpPre}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{dotpPre}}$ receives $\langle\cdot\rangle$-shares of $\vec{\mathbf{u}} = \{\mathsf{u}_1, \ldots, \mathsf{u}_\mathsf{d}\}$, $\vec{\mathbf{v}} = \{\mathsf{v}_1, \ldots, \mathsf{v}_\mathsf{d}\}$ from the parties. Let $P^\star$ denotes the party corrupted by $\mathcal{S}$. $\mathcal{F}_{\mathsf{dotpPre}}$ receives $(\mathsf{w}_i, \mathsf{w}_j)$ from $\mathcal{S}$ as its share for $\langle\mathsf{w}\rangle$ where $\mathsf{w} = \vec{\mathbf{u}} \odot \vec{\mathbf{v}}$. $\mathcal{F}_{\mathsf{dotpPre}}$ proceeds as follows:

1. Reconstructs $\vec{\mathbf{u}}, \vec{\mathbf{v}}$ using the shares received from honest parties and compute $\mathsf{w} = \vec{\mathbf{u}} \odot \vec{\mathbf{v}}$.
2. Computes the third share $\mathsf{w}_k = \mathsf{w} - \mathsf{w}_i - \mathsf{w}_j$ and sets $\langle\mathsf{w}\rangle_1 = (\mathsf{w}_1, \mathsf{w}_3), \langle\mathsf{w}\rangle_2 = (\mathsf{w}_2, \mathsf{w}_3), \langle\mathsf{w}\rangle_3 = (\mathsf{w}_1, \mathsf{w}_2)$.
3. Send $(\mathsf{Output}, \langle\mathsf{w}\rangle_s)$ to $P_s \in \mathcal{P}$.

Figure 8.2: Ideal functionality for Π_{dotpPre} in SWIFT.

To realize $\mathcal{F}_{\mathsf{dotpPre}}$, the approach is to run a semi-honest dot product protocol followed by a verification phase to check the correctness of the output. For verification, the work of [24] provides techniques to verify the correctness of m multiplication triples (and degree-two relations) at the cost of $\mathcal{O}(\sqrt{m})$ extended ring elements, albeit with abort security. While [27] improves their techniques to provide *robust* verification for multiplication, we show how to extend the techniques in [27] to robustly verify the correctness of m dot product tuples (dot product being a degree two relation), with vectors of dimension d, at a cost of $\mathcal{O}(\sqrt{\mathsf{d}m})$ extended ring elements. Thus, the cost to realize one instance of $\mathcal{F}_{\mathsf{dotpPre}}$ can be brought down to only the cost of a semi-honest dot product computation (which is 3 ring elements and independent of the vector dimension), where the cost due to verification can be amortized away by setting d, m appropriately.

Given vectors $\vec{\mathbf{u}} = (\mathsf{u}_1, \ldots, \mathsf{u}_\mathsf{d}), \vec{\mathbf{v}} = (\mathsf{v}_1, \ldots, \mathsf{v}_\mathsf{d})$, the semi-honest dot product protocol proceeds as follows. The parties, using the shared key setup, non-interactively generate *3-out-of-3* additive shares of zero using $\mathcal{F}_{\mathsf{zero}}$ (§4.1.1.1), i.e P_i has ζ_i, such that $\zeta_1 + \zeta_2 + \zeta_3 = 0$. Then, parties proceed with generating the $\langle\cdot\rangle$-shares of $\mathsf{w} = \vec{\mathbf{u}} \odot \vec{\mathbf{v}}$ as:

$$
\begin{aligned}
P_1 \;\; \text{computes and sends} \;\; \mathsf{y}_1 &= \zeta_1 + \sum_{j=1}^{\mathsf{d}} (\mathsf{u}_j^1 \mathsf{v}_j^3 + \mathsf{u}_j^3 \mathsf{v}_j^1 + \mathsf{u}_j^3 \mathsf{v}_j^3) \;\; \text{to} \;\; P_2 \\
P_2 \;\; \text{computes and sends} \;\; \mathsf{y}_2 &= \zeta_2 + \sum_{j=1}^{\mathsf{d}} (\mathsf{u}_j^2 \mathsf{v}_j^3 + \mathsf{u}_j^3 \mathsf{v}_j^2 + \mathsf{u}_j^2 \mathsf{v}_j^2) \;\; \text{to} \;\; P_3 \\
P_3 \;\; \text{computes and sends} \;\; \mathsf{y}_3 &= \zeta_3 + \sum_{j=1}^{\mathsf{d}} (\mathsf{u}_j^1 \mathsf{v}_j^2 + \mathsf{u}_j^2 \mathsf{v}_j^1 + \mathsf{u}_j^1 \mathsf{v}_j^1) \;\; \text{to} \;\; P_1
\end{aligned} \tag{8.1}
$$

Now, to complete the $\langle\cdot\rangle$-sharing of w, parties locally set $\mathsf{w}^1 = \mathsf{y}_3$, $\mathsf{w}^2 = \mathsf{y}_2$ and $\mathsf{w}^3 = \mathsf{y}_1$. To

check the correctness of the computation $\langle \mathsf{w} \rangle = \langle \vec{\mathbf{u}} \odot \vec{\mathbf{v}} \rangle$, each $P_i \in \mathcal{P}$ needs to prove that the y_i it sent in the semi-honest protocol satisfies 8.1. Without loss of generality, consider the case when $P_i = P_1$. Then, it has to prove

$$\zeta_1 + \sum_{j=1}^{\mathsf{d}} (\mathsf{u}_j^1 \mathsf{v}_j^3 + \mathsf{u}_j^3 \mathsf{v}_j^1 + \mathsf{u}_j^3 \mathsf{v}_j^3) - \mathsf{y}_1 = 0 \tag{8.2}$$

This difference in the expected message that should be sent (computed using P_1's correct input shares) and the actual message sent by P_1 is captured by a circuit c, defined below.

$$c\left(\{\mathsf{u}_j^1, \mathsf{u}_j^3, \mathsf{v}_j^1, \mathsf{v}_j^3\}_{j=1}^{\mathsf{d}}, \zeta_1, \mathsf{w}_1\right) = \zeta_1 + \sum_{j=1}^{\mathsf{d}} (\mathsf{u}_j^1 \mathsf{v}_j^3 + \mathsf{u}_j^3 \mathsf{v}_j^1 + \mathsf{u}_j^3 \mathsf{v}_j^3) - \mathsf{y}_1 \tag{8.3}$$

Here, c takes as input $u = 4\mathsf{d} + 2$ values: $\langle \cdot \rangle$-shares of $\vec{\mathbf{u}}, \vec{\mathbf{v}}$ held by P_1, i.e. $\{\mathsf{u}_j^1, \mathsf{u}_j^3, \mathsf{v}_j^1, \mathsf{v}_j^3\}_{j=1}^{\mathsf{d}}$, the additive share of zero, ζ_1, that P_1 holds, and the additive share y_1 sent by P_1. For correct computation with respect to P_1, we require the difference in the expected message and the actual message to be 0, i.e.,

$$c\left(\{\mathsf{u}_j^1, \mathsf{u}_j^3, \mathsf{v}_j^1, \mathsf{v}_j^3\}_{j=1}^{\mathsf{d}}, \zeta_1, \mathsf{w}_1\right) = 0 \tag{8.4}$$

We now explain how to verify the correctness for m dot product tuples assuming that the operations are carried out over a prime-order field. The verification can be extended to support operations over rings following the techniques of [24, 27]. To verify the correctness for m dot product tuples, $\{\vec{\mathbf{u}}_k, \vec{\mathbf{v}}_k, \mathsf{w}_k\}_{k=1}^m$ where $\mathsf{w}_k = \vec{\mathbf{u}}_k \odot \vec{\mathbf{v}}_k$, the output of c (which is the difference in the expected and actual message sent) for each of the corresponding dot product tuple must be 0. To check correctness of all dot products *at once*, it suffices to check if a random linear combination of the output of each c (for each dot product) is 0. This is because the random linear combination of the differences will be 0 with high probability if $\mathsf{w}_k = \vec{\mathbf{u}}_k \odot \vec{\mathbf{v}}_k$ for each $k \in \{1, \ldots, m\}$. We remark that the definition of $c(\cdot)$ in [27] enables the verification of only multiplication triples. With the re-definition of c as in 8.3, we can now verify the correctness of dot products while the rest of the verification steps remain similar to that in [27]. We elaborate on the details next.

A verification circuit, constructed as follows, enables P_i to prove the correctness of the additive share of w that it sent, for m instances of dot product at once. Note that the proof system is designed for the distributed-verifier setting where the proof generated by P_i will be shared among P_{i-1}, P_{i+1}, who can together verify its correctness. First, a sub-circuit g is defined as follows: group L small c circuits and take a random linear combination of the values on their

output wires. Since each c circuit takes $u = 4\mathsf{d} + 2$ inputs as described earlier, g takes in uL inputs. Precisely, g is defined as follows:

$$g(x_1, \ldots, x_{uL}) = \sum_{k=1}^{L} \theta_k \cdot c(x_{(k-1)u+1}, \ldots, x_{(k-1)u+u})$$

Since there are total m dot products to be verified, there will be $M = m/L$ sub-circuits g. Looking ahead, this grouping technique enables obtaining a sub-linear communication cost for verification because the communication cost turns out to be $\mathcal{O}(uL + M)$ and setting $uL = M$ gives the desired result. The sub-circuits g make up the circuit G which outputs a random linear combination of the values on the output wires of each g, i.e:

$$G(x_1, \ldots, x_{um}) = \sum_{k=1}^{M} \eta_k \cdot g(x_{(k-1)uL+1}, \ldots, x_{(k-1)uL+uL})$$

Here, θ_k and η_k are randomly sampled (non-interactively) by all parties. To prove correctness, P_i needs to prove that G outputs 0. For this, P_i defines $f_1 \ldots, f_{uL}$ random polynomials of degree M, one for each input wire of g. For $\ell \in \{1, \ldots, M\}$ and $j \in \{1, \ldots, uL\}$, $f_j(0)$ is chosen randomly and $f_j(\ell) = x_{(\ell-1)u+j}$ (i.e the jth input of the ℓth g gate). P_i further defines a $2M$ degree polynomial $p(\cdot)$ on the output wires of g, i.e $p(\cdot) = g(f_1, \ldots, f_{uL})$ where $p(\ell)$ for $\ell \in \{1, \ldots, M\}$ is the output of the ℓth g gate. The additional $M + 1$ points required to interpolate the $2M$ degree polynomial p, are obtained by evaluating $f_1, \ldots, f_{uL}$ on $M + 1$ additional points, followed by an application of g circuit. The proof generated by P_i consists of $f_1(0), \ldots, f_{uL}(0)$ and the coefficients of p. Recall that since we are in the distributed-verifier setting, the prover P_i additively shares the proof with P_{i-1}, P_{i+1}. Note here, that shares of $f_1(0), \ldots, f_{uL}(0)$ can be generated non-interactively.

To verify the proof, verifiers P_{i-1}, P_{i+1} need to check if the output of G is 0. This can be verified by computing the output of G as $b = \sum_{\ell=1}^{M} \eta_\ell \cdot p(\ell)$ and checking if $b = 0$, where η_ℓ's are non-interactively sampled by all after the proof is sent. If p is defined correctly, then this is indeed a random linear combination of the outputs of all the g-circuits. This necessitates the second check to verify the correctness of p as per its definition i.e $p(\cdot) = g(f_1(\cdot), \ldots, f_{uL}(\cdot))$. This is performed by checking if $p(r) = g(f_1(r), \ldots, f_{uL}(r))$ for a random $r \notin \{1, \ldots, M\}$ (for privacy to hold) sampled non-interactively by all after the proof is sent. For the first check, verifiers can locally compute additive shares of b (using the additive shares of coefficients of p obtained as part of the proof) and reconstruct b to check for equality with 0. For the second, verifiers locally compute additive shares of $p(r)$ using the shares of coefficients of p, and shares

of $f_1(r), \ldots, f_{uL}(r)$ by interpolating $f_1, \ldots, f_{uL}$ using (P_i's) inputs to the c-circuits which are implicitly additively shared between them (owing to the replicated sharing property). Verifiers exchange these values among themselves, reconstruct it and check if $p(r) = g(f_1(r), \ldots, f_{uL}(r))$. Note that the messages computed and exchanged by the verifiers depend only on the proof sent by P_i and the random values (r, η) sampled by all. P_i can independently compute these messages. Thus, to prevent a verifier from falsely rejecting a correct proof, we use jsnd to exchange these messages. To optimize the communication cost further, it suffices if a single verifier computes the output of verification.

Setting the parameters: The proof sent by P_i consists of the constant terms $f_j(0)$ for $j \in \{1, \ldots, uL\}$ and $2M + 1$ coefficients of p. The former can be can be generated non-interactively. Hence, P_i needs to communicate $2M + 1$ elements to the verifiers (one of which can be performed non-interactively). The message sent by the verifier consists of the additive share of $\sum_{\ell=1}^{M} \eta_\ell \cdot p(\ell)$ (for the first check) and $f_1(r), \ldots, f_{uL}(r), p(r)$ (for the second check). Thus, the verifier communicates $uL + 2$ elements. As the proof is executed three times, each time with one party acting as the prover and the other two acting as the verifiers, overall, each party communicates $uL + 2M + 3$ elements. Setting $uL = 2M$ and $M = \frac{m}{L}$ results in the total communication required for verifying m dot products to be $\mathcal{O}(\sqrt{\mathsf{d}m})$. Thus, verifying a single dot product has an amortized cost of $\mathcal{O}\left(\sqrt{\frac{\mathsf{d}}{m}}\right)$ which can be made very small by appropriately setting the values of d, m. Thus, the (amortized) cost of a maliciously secure dot product protocol can be made equal to that of a semi-honest dot product protocol, which is 3 ring elements.

To support verification over rings [27], verification operations are carried out on the extended ring $\mathbb{Z}_{2^\ell}/f(x)$, which is the ring of all polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ modulo a polynomial f, of degree d, irreducible over $\mathbb{Z}_2$. Each element in $\mathbb{Z}_{2^\ell}$ is lifted to a d-degree polynomial in $\mathbb{Z}_{2^\ell}[x]/f(x)$ (which results in blowing up the communication by a factor d). Thus, the per party communication amounts to $(uL + 2M + 3)d$ elements of $\mathbb{Z}_{2^\ell}$ for verifying m dot products of vector size d where $u = 4\mathsf{d} + 2$. Further, the probability of a cheating prover is bounded by $\frac{2^{(\ell-1)d} \cdot 2M+1}{2^{\ell d} - M}$ (cf. Theorem 4.7 of [27]). Thus, if γ is such that $2^\gamma \geq 2M$, then the cheating probability is

$$\frac{2^{(\ell-1)d} \cdot 2M + 1}{2^{\ell d} - M} \leq \frac{2^{(\ell-1)d} \cdot 2^\gamma + 1}{2^{\ell d} - M} \approx 2^{-(d-\gamma)}$$

We note that both, [27] and our technique require a communication cost of $\mathcal{O}(\sqrt{m\mathsf{d}})$ ring elements for verifying m dot products of vector size d. This is because multiplication is a

special case of dot product with $\mathsf{d} = 1$. However, since our verification is for dot products, we can get away with performing only m semi-honest dot products whose cost is equivalent to computing m semi-honest multiplications, whereas [27] requires to execute $m\mathsf{d}$ multiplications (as their technique can only verify correctness of multiplications), resulting in a dot product cost dependent on the vector size. Concretely, to get 40 bits of statistical security and for a vector size of 2^{10} (CIFAR-10 [88] dataset), the parameters mentioned above can be set as given in Table 8.1.

m^a	M^b	γ	d^c	Cost (per dot product)
2^{20}	2^{16}	17	57	7.125
2^{30}	2^{21}	22	62	0.242
2^{40}	2^{26}	27	67	0.008
2^{50}	2^{31}	32	72	0.0002

[a]#dot products to be verified [b]#g sub-circuits [c]degree of extension

Table 8.1: Cost of verification in terms of the number of ring elements communicated per dot product, and parameters for vector size $\mathsf{d} = 2^{10}$ and 40 bits of statistical security.

It is possible to further bring down the communication cost required for verifying m dot product tuples to $\mathcal{O}(\log(\mathsf{d}m))$ at the expense of requiring more rounds by further extending the technique of [24], which we leave as an exercise. We refer readers to [27] for formal details.

Lemma 8.1 (Communication) *Protocol* Π_{dotp} *(Fig. 8.1) (in SWIFT) requires* 3ℓ *bits of communication in preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

8.1.2 Bit Extraction

To compute most significant bit (msb) of the value v, note that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2 + \mathsf{v}_3$ for $\mathsf{v}_1 = \mathsf{m}_\mathsf{v} - \lambda^3_\mathsf{v}$, $\mathsf{v}_2 = -\lambda^1_\mathsf{v}$ and $\mathsf{v}_3 = -\lambda^2_\mathsf{v}$ as per the sharing semantics (cf. Table 4.2). Parties generate the boolean sharing of $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3$ using joint sharing protocol. It has been shown in ABY3 [101] that $\mathsf{v} = 2c + s$ where $\mathsf{FA}(\mathsf{v}_1[i], \mathsf{v}_2[i], \mathsf{v}_3[i]) \to (c[i], s[i])$ 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, parties 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 $\mathsf{msb}(2[\![c]\!]^{\mathbf{B}} + [\![s]\!]^{\mathbf{B}})$ by evaluating the bit extraction circuit [101, 113].

8.1.3 Bit to Arithmetic

Protocol $\Pi_{\mathsf{bit2A}}(\llbracket \mathsf{b} \rrbracket^{\mathbf{B}})$ (Fig. 8.3) enables computing $\llbracket \mathsf{b} \rrbracket$ of a bit b given its boolean sharing $\llbracket \mathsf{b} \rrbracket^{\mathbf{B}}$. Let b^{R} denotes the value of $\mathsf{b} \in \{0, 1\}$ over the arithmetic ring $\mathbb{Z}_{2^\ell}$. Using our sharing semantics,

$$\mathsf{b}^{\mathsf{R}} = (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}} = \mathsf{m}_{\mathsf{b}}^{\mathsf{R}} + \lambda_{\mathsf{b}}^{\mathsf{R}}(1 - 2\mathsf{m}_{\mathsf{b}}^{\mathsf{R}}) \tag{8.5}$$

Protocol $\Pi_{\mathsf{bit2A}}(\llbracket \mathsf{b} \rrbracket^{\mathbf{B}})$

Let $\mathsf{u} = \lambda_{\mathsf{b}}^{\mathsf{R}}$ and $\mathsf{v} = \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}$.
Input(s): $\llbracket \mathsf{b} \rrbracket^{\mathbf{B}}$, **Output:** $\llbracket \mathsf{y} \rrbracket = \llbracket \mathsf{b}^{\mathsf{R}} \rrbracket$.

Preprocessing:

1. (P_1, P_3), (P_2, P_3) and (P_1, P_2) locally generate $\langle \cdot \rangle$-shares of $(\lambda_{\mathsf{b}}^1)^{\mathsf{R}}$, $(\lambda_{\mathsf{b}}^2)^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}}^3)^{\mathsf{R}}$ respectively (Table 4.3).
2. Compute the $\langle \cdot \rangle$-shares of $(\lambda_{\mathsf{b}}^1)^{\mathsf{R}}(\lambda_{\mathsf{b}}^2)^{\mathsf{R}}$ using Π_{MultPre}.
3. Locally compute $\langle \sigma \rangle = \langle (\lambda_{\mathsf{b}}^1)^{\mathsf{R}} \rangle + \langle (\lambda_{\mathsf{b}}^2)^{\mathsf{R}} \rangle - 2\langle (\lambda_{\mathsf{b}}^1)^{\mathsf{R}}(\lambda_{\mathsf{b}}^2)^{\mathsf{R}} \rangle$.
4. Compute the $\langle \cdot \rangle$-shares of $\sigma_1^{\mathsf{R}}(\lambda_{\mathsf{b}}^3)^{\mathsf{R}}$ using Π_{MultPre}.
5. Locally compute $\langle \mathsf{u} \rangle = \langle \sigma \rangle + \langle (\lambda_{\mathsf{b}}^3)^{\mathsf{R}} \rangle - 2\langle \sigma(\lambda_{\mathsf{b}}^3)^{\mathsf{R}} \rangle$.

Online: Let $\mathsf{y} = \mathsf{b}^{\mathsf{R}}$.

1. Locally compute the following:

$$P_1, P_3 : \mathsf{y}_1 = \mathsf{v} + \mathsf{u}^1(1 - 2\mathsf{v}) \;\Big|\; P_2, P_3 : \mathsf{y}_2 = \mathsf{u}^2(1 - 2\mathsf{v}) \;\Big|\; P_1, P_2 : \mathsf{y}_3 = \mathsf{u}^3(1 - 2\mathsf{v})$$

2. $(P_1, P_3), (P_2, P_3), (P_1, P_2)$ execute Π_{JSh} on $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ to generate the respective $[\![\cdot]\!]$-shares.
3. Compute $\llbracket \mathsf{y} \rrbracket = \llbracket \mathsf{y}_1 \rrbracket + \llbracket \mathsf{y}_2 \rrbracket + \llbracket \mathsf{y}_3 \rrbracket$.

Figure 8.3: Bit to Arithmetic conversion in SWIFT.

During preprocessing, parties locally generate $\langle \cdot \rangle$-shares of $(\lambda_{\mathsf{b}}^1)^{\mathsf{R}}$, $(\lambda_{\mathsf{b}}^2)^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}}^3)^{\mathsf{R}}$ similar to Π_{JSh} (Table 4.3, ignore m values). Then, $\langle \sigma^{\mathsf{R}} \rangle$ can be computed in the preprocessing using two instances of Π_{MultPre} as given in (8.6).

$$\begin{aligned} \sigma_1^{\mathsf{R}} &= (\lambda_{\mathsf{b}}^1 \oplus \lambda_{\mathsf{b}}^2)^{\mathsf{R}} = (\lambda_{\mathsf{b}}^1)^{\mathsf{R}} + (\lambda_{\mathsf{b}}^2)^{\mathsf{R}} - 2(\lambda_{\mathsf{b}}^1)^{\mathsf{R}}(\lambda_{\mathsf{b}}^2)^{\mathsf{R}} \\ \sigma^{\mathsf{R}} &= (\sigma_1 \oplus \lambda_{\mathsf{b}}^3)^{\mathsf{R}} = \sigma_1^{\mathsf{R}} + (\lambda_{\mathsf{b}}^3)^{\mathsf{R}} - 2\sigma_1^{\mathsf{R}}(\lambda_{\mathsf{b}}^3)^{\mathsf{R}} \end{aligned} \tag{8.6}$$

The online phase consists of each pair of parties (P_1, P_3), (P_2, P_3) and (P_1, P_2) locally computing an additive sharing of b^{R} using (8.5), generating the corresponding $[\![\cdot]\!]$-sharing using Π_{JSh}, and locally adding the shares to obtain $[\![\mathsf{b}^{\mathsf{R}}]\!]$.

Lemma 8.2 (Communication) *Protocol Π_{bit2A} (Fig. 8.3) requires 6ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Proof: During the preprocessing, generation of $\langle\cdot\rangle$-shares of $(\lambda_{\mathsf{b}}^1)^{\mathsf{R}}$, $(\lambda_{\mathsf{b}}^2)^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}}^3)^{\mathsf{R}}$ is local. Two instances of Π_{MultPre} are executed in the preprocessing incurring a communication of 6ℓ bits. The online phase involves three instances of arithmetic joint sharing protocol in parallel, resulting in 1 round and a communication of 3ℓ bits. □

8.1.3.1 Bit to Arithmetic:II

Similar to Π_{bit2A} protocol, given the boolean sharings $[\![\mathsf{b}_1]\!]^{\mathbf{B}}$, $[\![\mathsf{b}_2]\!]^{\mathbf{B}}$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1-2\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}})$, $(1-2\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}})$ respectively. Using (8.5), we can write

$$\begin{aligned}(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}} = (\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_1})(\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_2}) \\ &= \mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_1} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \end{aligned} \tag{8.7}$$

During preprocessing, the $\langle\cdot\rangle$-shares of $\lambda_{\mathsf{b}_1}^{\mathsf{R}}$ and $\lambda_{\mathsf{b}_2}^{\mathsf{R}}$ are computed similar to that of Π_{bit2A} (Fig. 8.3). Parties then compute the $\langle\cdot\rangle$-shares of $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$ using another instance of Π_{MultPre}. The online phase is similar to that of Π_{bit2A} protocol.

Lemma 8.3 (Communication) *Protocol Π_{dbit2A} requires 15ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

8.1.4 Bit Injection

Given the boolean 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 $\mathsf{b}^{\mathsf{R}}\mathsf{v}$. Let Δ_{b} denote the value $(1 - 2\mathsf{m}_{\mathsf{b}}^{\mathsf{R}})$. Similar to Π_{bit2A},

$$\begin{aligned}\mathsf{b}^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) = (\mathsf{m}_{\mathsf{b}}^{\mathsf{R}} + \lambda_{\mathsf{b}}^{\mathsf{R}}\Delta_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\ &= \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}} - \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}} + \lambda_{\mathsf{b}}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}} - \lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}\Delta_{\mathsf{b}} \end{aligned} \tag{8.8}$$

During the preprocessing, parties generates the $\langle\cdot\rangle$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}}$ similar to Π_{bit2A} protocol. This is followed by generating the $\langle\cdot\rangle$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}}$ using Π_{MultPre}. The online phase is similar to that of Π_{bit2A} protocol.

Lemma 8.4 (Communication) *Protocol Π_{bitInj} requires 9ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

8.1.4.1 Sum of Bit Injections

Given m pair of values in the shared form, $\{[\![\mathsf{b}_i]\!]^{\mathbf{B}}, [\![\mathsf{v}_i]\!]\}_{i\in[m]}$, the goal of Π_{bitInjS} is to compute the $[\![\cdot]\!]$-share of $\mathsf{z} = \sum_{i=1}^{m} \mathsf{b}^{\mathsf{R}}_i \cdot \mathsf{v}_i$. For this, parties execute the preprocessing corresponding to m bit injections of the form $\mathsf{b}^{\mathsf{R}}_i \cdot \mathsf{v}_i$.

In the online phase, parties locally compute an additive sharing of z_i, corresponding to $\mathsf{b}^{\mathsf{R}}_i \cdot \mathsf{v}_i$ first. Instead of generating the $[\![\cdot]\!]$-sharing for each of the m terms, parties locally add the shares and execute Π_{JSh} on the result. This results in an online communication independent of m.

Lemma 8.5 (Communication) *Protocol Π_{bitInjS} requires $m \cdot 9\ell$ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

8.1.4.2 Bit Injection:II

Similar to Π_{bitInj} protocol, given $[\![\mathsf{b}_1]\!]^{\mathbf{B}}, [\![\mathsf{b}_2]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1-2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1})$, $(1-2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2})$ respectively. Using (8.7) and (8.8), we can write

$$\begin{aligned}
(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\
&= (\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_1}\Delta_{\mathsf{b}_1})(\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\Delta_{\mathsf{b}_2})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\
&= \mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2}\mathsf{m}_{\mathsf{v}} + \lambda^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \\
&\quad - \lambda_{\mathsf{v}}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2} - \lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda_{\mathsf{v}}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2}\Delta_{\mathsf{b}_1} - \lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda_{\mathsf{v}}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} - (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_{\mathsf{v}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \qquad (8.9)
\end{aligned}$$

During preprocessing, the $\langle\cdot\rangle$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}_1}$ and $\lambda^{\mathsf{R}}_{\mathsf{b}_2}$ are computed similar to that of Π_{bit2A} (Fig. 8.3). Parties then compute the $\langle\cdot\rangle$-shares of $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$, $\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda_{\mathsf{v}}$, $\lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda_{\mathsf{v}}$ and $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_{\mathsf{v}}$ using four instances of Π_{MultPre}. The online phase is similar to that of Π_{bit2A} protocol.

Lemma 8.6 (Communication) *Protocol Π_{dbitInj} requires 24ℓ bits of communication in preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

8.1.5 Truncation Pair Generation (Π_{trgen})

Protocol Π_{trgen} (Fig. 8.4) allows parties to generate a truncation pair of the form $(\langle \mathsf{r} \rangle, [\![\mathsf{r}^{\mathsf{t}}]\!])$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$. Analogous to the approach of ABY3 [101], parties non-interactively generate the boolean sharing of an ℓ-bit value r first. Parties then discard the shares for the lower x bit positions to obtain the boolean shares of the truncated value denoted by r^{t}. To obtain the arithmetic shares of the truncation pair, we do not rely on the approach of ABY3 as it requires more rounds. Instead, we implicitly perform a *boolean to arithmetic* conversion using techniques from bit to arithmetic protocol Π_{bit2A}.

Protocol Π_{trgen}

Let $i \in \{0, \ldots, \ell-1\}$ and $j \in \{0, \ldots, \ell-1-x\}$. Here x denotes the precision in FPA semantics.

1. P_s, P_3 for $s \in \{1, 2\}$ sample ℓ-bits, denoted by $\mathsf{r}_s[i]$.
2. Define ℓ-bit value $\mathsf{r} = \mathsf{r}_1 \oplus \mathsf{r}_2$. i.e. $\mathsf{r}[i] = \mathsf{r}_1[i] \oplus \mathsf{r}_2[i]$.
3. P_s, P_3 for $j \in \{1, 2\}$ execute Π_{JSh} on $(\mathsf{r}_s[i])^{\mathsf{R}}$ to generate the respective $[\![\cdot]\!]$-shares.
4. Locally compute $\langle \cdot \rangle$-shares of $(\mathsf{r}_1[i])^{\mathsf{R}}$ and $(\mathsf{r}_2[i])^{\mathsf{R}}$[a].
5. Define ℓ-sized vectors $\vec{\mathbf{a}}, \vec{\mathbf{b}}$ as: $\mathsf{a}_j = 2^{i+1}(\mathsf{r}_1[i])^{\mathsf{R}}$ and $\mathsf{b}_i = (\mathsf{r}_2[i])^{\mathsf{R}}$.
6. Define $(\ell - x)$-sized vectors $\vec{\mathbf{c}}, \vec{\mathbf{d}}$ as: $\mathsf{c}_j = 2^{j+1}(\mathsf{r}_1[j+x])^{\mathsf{R}}$ and $\mathsf{b}_j = (\mathsf{r}_2[j+x])^{\mathsf{R}}$.
7. Locally compute $\langle \vec{\mathbf{a}} \rangle, \langle \vec{\mathbf{b}} \rangle, \langle \vec{\mathbf{c}} \rangle, \langle \vec{\mathbf{d}} \rangle$.
8. Compute the $\langle \cdot \rangle$-shares of $\mathsf{x} = \vec{\mathbf{a}} \odot \vec{\mathbf{b}}$ and $\mathsf{y} = \vec{\mathbf{c}} \odot \vec{\mathbf{d}}$ using Π_{dotpPre} protocol[b].
9. Locally compute $\langle \mathsf{r} \rangle = \sum_{i=0}^{\ell-1} 2^i (\langle (\mathsf{r}_1[i])^{\mathsf{R}} \rangle + \langle (\mathsf{r}_2[i])^{\mathsf{R}} \rangle) - \langle \mathsf{x} \rangle$.
10. Locally compute $[\![\mathsf{r}^{\mathsf{t}}]\!] = \sum_{j=0}^{\ell-1-x} 2^j ([\![(\mathsf{r}_1[j+x])^{\mathsf{R}}]\!] + [\![(\mathsf{r}_2[j+x])^{\mathsf{R}}]\!]) - [\![\mathsf{y}]\!]$.

[a] by discarding the m value that is set to 0 as per Table 4.3
[b] $[\![\mathsf{y}]\!]$ can be computed by locally setting $\mathsf{m}_{\mathsf{y}} = 0$

Figure 8.4: Truncation pair generation in SWIFT.

Concretely, P_1, P_3 sample an ℓ-bit value r_1 while P_2, P_3 sample r_2. For the i^{th} bit position, define $\mathsf{r}[i] = \mathsf{r}_1[i] \oplus \mathsf{r}_2[i]$ for $i \in \{0, \ldots, \ell-1\}$. For r defined as above, we have $\mathsf{r}^{\mathsf{t}}[j] = \mathsf{r}_1[j+x] \oplus \mathsf{r}_2[j+x]$ for $j \in \{0, \ldots, \ell-1-x\}$. Further,

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

$$= \sum_{i=0}^{\ell-1} 2^i \left((\mathsf{r}_1[i])^{\mathsf{R}} + (\mathsf{r}_2[i])^{\mathsf{R}} \right) - \sum_{i=0}^{\ell-1} \left(\left(2^{i+1} (\mathsf{r}_1[i])^{\mathsf{R}} \right) \cdot (\mathsf{r}_2[i])^{\mathsf{R}} \right) \tag{8.10}$$

Similarly, for r^{t},

$$\mathsf{r}^{\mathsf{t}} = \sum_{j=0}^{\ell-1-x} 2^j \left((\mathsf{r}_1[j+x])^{\mathsf{R}} + (\mathsf{r}_2[j+x])^{\mathsf{R}} \right) - \sum_{j=0}^{\ell-1-x} \left(\left(2^{j+1} (\mathsf{r}_1[j+x])^{\mathsf{R}} \right) \cdot (\mathsf{r}_2[j+x])^{\mathsf{R}} \right) \tag{8.11}$$

Given the boolean shares, parties can evaluate (8.10) and (8.11) using two instances of Π_{dotpPre} as shown in Fig. 8.4.

Lemma 8.7 (Communication) *Protocol Π_{trgen} (Fig. 8.4) requires 6ℓ bits of communication.*

8.1.6 Equality Test (Π_{eq})

To check whether $\mathsf{a} \stackrel{?}{=} \mathsf{b}$ or not, given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$, Π_{eq} proceeds with parties locally computing $[\![\mathsf{y}]\!] = [\![\mathsf{a}]\!] - [\![\mathsf{b}]\!]$. According to our sharing semantics, y can be written as $\mathsf{y} = \mathsf{y}_1 - \mathsf{y}_2$ where $\mathsf{y}_1 = \mathsf{m}_{\mathsf{y}} - \lambda_{\mathsf{y}}^3$ and $\mathsf{y}_2 = \lambda_{\mathsf{y}}^1 + \lambda_{\mathsf{y}}^2$.

During preprocessing, (P_1, P_3) and (P_2, P_3) generate the $[\![\cdot]\!]^{\mathbf{B}}$-shares of λ_{y}^1 and λ_{y}^2 respectively using Π_{JSh}. Parties then compute $[\![\mathsf{y}_2]\!]^{\mathbf{B}}$ using a boolean adder (PPA) circuit. During the online phase, P_1, P_2 generate $[\![\mathsf{y}_1]\!]^{\mathbf{B}}$ using Π_{JSh}. Note that $\mathsf{a} = \mathsf{b}$ implies $\mathsf{y}_1 = \mathsf{y}_2$ and hence all the bits of $\mathsf{v} = \overline{(\mathsf{y}_1 \oplus \mathsf{y}_2)}$ should be 1. As mentioned in the introduction of Part II (II), parties use four input AND gates and a tree structure, where 4 bits are taken at a time and the AND of them is computed in one go.

8.2 Mixed Protocol Framework

Table 8.2 compares our sharing conversions with ABY3 [101]. For uniformity, we consider a function, F, to be computed on an ℓ-bit inputs x, y using a garbled circuit (GC) in the mixed framework, which gives an ℓ-bit output $\mathsf{z} = \mathsf{F}(\mathsf{x}, \mathsf{y})$, where ℓ denotes the ring size in bits. Let G^{F} denote the corresponding GC. In the table, G^{Sn} denotes a n-input garbled subtraction circuit; G^{An} denotes n-input garbled addition circuit; $\hat{\mathsf{G}}$ denotes the garbled circuit with decoding information; $\mathsf{G}^{\mathsf{n}_1 \times 1, \ldots, \mathsf{n}_{\mathsf{m}} \times \mathsf{m}}$ denotes n_{i} instances of GC G^{i} for $i \in \{1, \ldots, \mathsf{m}\}$ and $|\mathsf{G}^{\mathsf{n}_1 \times 1, \ldots, \mathsf{n}_{\mathsf{m}} \times \mathsf{m}}|$ denotes its size.

Variant[a]	Conversion[b]	ABY3 [101] Comm.$_{\text{pre}}$	ABY3 [101] Comm.$_{\text{on}}$	ABY3 [101] Rounds$_{\text{on}}$	SWIFT Comm.$_{\text{pre}}$	SWIFT Comm.$_{\text{on}}$	SWIFT Rounds$_{\text{on}}$
2 GC	A-G-A A-G-B B-G-A B-G-B	$2\lvert\hat{\mathsf{G}}^{2\times\mathsf{A3},\mathsf{S3},\mathsf{F}}\rvert$ $(2\ell\kappa)$ $2\lvert\mathsf{G}^{2\times\mathsf{A3},\mathsf{F}}\rvert$ $+$ $2\lvert\hat{\mathsf{G}}^{\mathsf{S3},\mathsf{F}}\rvert$ $2\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$10\ell\kappa$	2	$2\lvert\hat{\mathsf{G}}^{2\times\mathsf{A3},\mathsf{A2},\mathsf{F}}\rvert$ $(12\ell\kappa)$ $2\lvert\hat{\mathsf{G}}^{2\times\mathsf{A3},\mathsf{F}}\rvert$ $+$ $2\lvert\hat{\mathsf{G}}^{\mathsf{A2},\mathsf{F}}\rvert$ $2\lvert\hat{\mathsf{G}}^{\mathsf{F}}\rvert$	$4\ell\kappa+\ell$	1
1 GC	A-G-A A-G-B B-G-A B-G-B	$\lvert\hat{\mathsf{G}}^{2\times\mathsf{A3},\mathsf{S3},\mathsf{F}}\rvert$ $(\ell\kappa)$ $\lvert\mathsf{G}^{2\times\mathsf{A3},\mathsf{F}}\rvert$ $+$ $\lvert\hat{\mathsf{G}}^{\mathsf{S3},\mathsf{F}}\rvert$ $\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$5\ell\kappa$	2	$\lvert\hat{\mathsf{G}}^{2\times\mathsf{A3},\mathsf{A2},\mathsf{F}}\rvert$ $(6\ell\kappa)$ $\lvert\hat{\mathsf{G}}^{2\times\mathsf{A3},\mathsf{F}}\rvert$ $+$ $\lvert\hat{\mathsf{G}}^{\mathsf{A2},\mathsf{F}}\rvert$ $\lvert\hat{\mathsf{G}}^{\mathsf{F}}\rvert$	$2\ell\kappa+2\ell$	2
Others[c]	A-B	$12\ell+12\ell\log\ell$	$9\ell+9\ell\log\ell$	$1+\log\ell$	$\mathsf{u}_1+6\ell+6\ell\log\ell$	$3\mathsf{u}_2$	$\log_4\ell$
	B-A	$12\ell+12\ell\log\ell$	$9\ell+9\ell\log\ell$	$1+\log\ell$	$6\ell^2$	3ℓ	1

[a] Notations: ℓ - size of ring in bits, κ - computational security parameter, 'pre' - preprocessing, 'on' - online.
[b] 'A' - arithmetic, 'B' - boolean, 'G' - Garbled.
[c] $\mathsf{u}_1 = 3\mathsf{n}_2 + 12\mathsf{n}_3 + 33\mathsf{n}_4$, $\mathsf{u}_2 = \mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4$ denote the number of AND gates in the optimized adder circuit [113] with 2, 3, 4 inputs, respectively. For $\ell = 64$, $\mathsf{n}_2 = 216, \mathsf{n}_3 = 184, \mathsf{n}_4 = 179$.

Table 8.2: Mixed protocol conversions of ABY3 [101] and SWIFT.

8.2.1 Conversions involving Garbled World

Assume the GC is required to compute a function f on inputs $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and let the output be $f(\mathsf{x}, \mathsf{y})$. All the conversions described are for the 2 GC variant. Conversions for the 1 GC variant are straightforward, hence we omit the details.

Case I: Boolean-Garbled-Boolean Since the inputs to the GC are available in boolean form, say $[\![\mathsf{x}]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!]^{\mathbf{B}}$, parties generate $[\![\mathsf{x}]\!]^{\mathbf{C}}, [\![\mathsf{y}]\!]^{\mathbf{C}}$ by invoking the garbled sharing protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$. (P_1, P_3) sample $\mathsf{R}_1 \in \mathbb{Z}_{2^\ell}$ to mask the function output, $f(\mathsf{x}, \mathsf{y})$, and generate $[\![\mathsf{R}_1]\!]^{\mathbf{B}}$ and $[\![\mathsf{R}_1]\!]^{\mathbf{G}}$. Similarly, (P_2, P_3) sample $\mathsf{R}_2 \in \mathbb{Z}_{2^\ell}$ and generate $[\![\mathsf{R}_2]\!]^{\mathbf{B}}$ and $[\![\mathsf{R}_2]\!]^{\mathbf{G}}$. Garblers $P_g \in \{P_1, P_3\}$ garble the circuit which computes $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) \oplus \mathsf{R}_1 \oplus \mathsf{R}_2$, and send the GC along with the decoding information to evaluator P_1. Analogous steps are performed for evaluator P_2. Upon GC evaluation and output decoding, evaluators obtain $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) \oplus \mathsf{R}_1 \oplus \mathsf{R}_2$, and jointly boolean share z to generate $[\![\mathsf{z}]\!]^{\mathbf{B}}$. Parties then compute $[\![f(\mathsf{x}, \mathsf{y})]\!]^{\mathbf{B}} = [\![\mathsf{z}]\!]^{\mathbf{B}} \oplus [\![\mathsf{R}_1]\!]^{\mathbf{B}} \oplus [\![\mathsf{R}_2]\!]^{\mathbf{B}}$.

Case II: Boolean-Garbled-Arithmetic This is similar to *Case I* except that the circuit which computes $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) + \mathsf{R}_1 + \mathsf{R}_2$ is garbled instead. Boolean sharing of z is replaced with arithmetic, followed by computing $[\![f(\mathsf{x}, \mathsf{y})]\!] = [\![\mathsf{z}]\!] - [\![\mathsf{R}_1]\!] - [\![\mathsf{R}_2]\!]$.

Cases III & IV: Input in Arithmetic Sharing The function to be computed $f(\mathsf{x}, \mathsf{y})$, is modified as $f'(\mathsf{m}_\mathsf{x}, \lambda^1_\mathsf{x}, \lambda^2_\mathsf{x}, \lambda^3_\mathsf{x}, \mathsf{m}_\mathsf{y}, \lambda^1_\mathsf{y}, \lambda^2_\mathsf{y}, \lambda^3_\mathsf{y}) = f(\mathsf{m}_\mathsf{x} - \lambda^1_\mathsf{x} - \lambda^2_\mathsf{x} - \lambda^3_\mathsf{x}, \mathsf{m}_\mathsf{y} - \lambda^1_\mathsf{y} - \lambda^2_\mathsf{y} - \lambda^3_\mathsf{y})$ where inputs

x, y are replaced by the sets $\{\mathsf{m}_\mathsf{x}, \lambda^1_\mathsf{x}, \lambda^2_\mathsf{x}, \lambda^3_\mathsf{x}\}$, $\{\mathsf{m}_\mathsf{y}, \lambda^1_\mathsf{y}, \lambda^2_\mathsf{y}, \lambda^3_\mathsf{y}\}$. The circuit to be garbled thus, corresponds to the function f'. Parties generate $[\![\cdot]\!]^{\mathbf{G}}$-shares via $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$, following which, parties proceed with the rest of the computation whose steps are similar to *Case I*, and *II*, depending on the requirement on the output sharing. For the instance with P_1 as the evaluator, function f' can be further optimized as $f(\alpha_\mathsf{x} - \lambda^1_\mathsf{x} - \lambda^3_\mathsf{x}, \alpha_\mathsf{y} - \lambda^1_\mathsf{y} - \lambda^3_\mathsf{y})$ with $\alpha_\mathsf{x} = \mathsf{m}_\mathsf{x} - \lambda^2_\mathsf{x}$ and $\alpha_\mathsf{y} = \mathsf{m}_\mathsf{y} - \lambda^2_\mathsf{y}$. Similar optimization can be done for the other garbling instance as well.

8.2.2 Other Conversions

Arithmetic to Boolean To convert arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to boolean, observe that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$ where $\mathsf{v}_1 = \mathsf{m}_\mathsf{v}$ and $\mathsf{v}_2 = -\lambda_\mathsf{v}$. Thus, $[\![\mathsf{v}]\!]^{\mathbf{B}}$ can be computed as $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} + [\![\mathsf{v}_2]\!]^{\mathbf{B}}$. For this, parties generate $[\![\mathsf{v}_2]\!]^{\mathbf{B}}$ in the preprocessing, and $[\![\mathsf{v}_1]\!]^{\mathbf{B}}$ can be generated in the online locally by setting $\mathsf{m}_{\mathsf{v}_1} = \mathsf{v}_1$ and $\lambda_{\mathsf{v}_1} = \lambda_{\mathsf{v}_2} = \lambda_{\mathsf{v}_3} = 0$. The protocol appears in Fig. 8.5. Boolean addition, when instantiated using the adder of ABY2.0 [113], requires $\log_4(\ell)$ rounds.

Protocol Π_{A2B}

Let $\mathsf{v}_1 = \mathsf{m}_\mathsf{v}$ and $\mathsf{v}_2 = -\lambda_\mathsf{v}$.

Preprocessing:

1. Non-interactively generate $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathsf{u}_i = -\lambda^i_\mathsf{v}$ for $i \in \{1,2,3\}$ using Π_{JSh} (§4.2.1.1).
2. Evaluate $\mathsf{FA}(\mathsf{v}_1[i], \mathsf{v}_2[i], \mathsf{v}_3[i]) \to (c[i], s[i])$ for $i \in \{0, \ldots, \ell - 1\}$ to generate $[\![c[i]]\!]^{\mathbf{B}}$ and $[\![s[i]]\!]^{\mathbf{B}}$.
3. Compute $2[\![c]\!]^{\mathbf{B}} + [\![s]\!]^{\mathbf{B}}$ using a boolean adder circuit [101, 113].

Online:

1. Locally generate $[\![\mathsf{v}_1]\!]^{\mathbf{B}}$ as $\mathsf{m}_{\mathsf{v}_1} = \mathsf{v}_1$ and $\lambda^{=}_{\mathsf{v}_1} \lambda^{=}_{\mathsf{v}_2} \lambda^{=}_{\mathsf{v}_3} 0$.
2. Compute $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} + [\![\mathsf{v}_2]\!]^{\mathbf{B}}$ using a boolean adder circuit [113].

Figure 8.5: Arithmetic to Boolean Conversion in SWIFT.

To generate $[\![\mathsf{v}_2]\!]^{\mathbf{B}}$, let $\mathsf{v}_2 = \mathsf{u}_1 + \mathsf{u}_2 + \mathsf{u}_3$ where $\mathsf{u}_i = -\lambda^i_\mathsf{v}$ for $i \in \{1,2,3\}$. Parties non-interactively generate the $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3$ using joint sharing protocol (§4.2.1.1). For a full adder circuit $\mathsf{FA}(\mathsf{v}_1[i], \mathsf{v}_2[i], \mathsf{v}_3[i]) \to (c[i], s[i])$ for $i \in \{0, \ldots, \ell - 1\}$, it has been shown in ABY3 [101] that $\mathsf{v}_2 = 2c + s$ where s and c denote the sum and carry bits respectively. Parties 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 $2[\![c]\!]^{\mathbf{B}} + [\![s]\!]^{\mathbf{B}}$ by evaluating a boolean adder circuit [101, 113].

Boolean to Arithmetic To convert a boolean sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ into an arithmetic sharing, note that

$$\mathsf{v} = \sum_{i=0}^{\ell-1} 2^i \mathsf{v}[i] = \sum_{i=0}^{\ell-1} 2^i (\lambda_{\mathsf{v}[i]} \oplus \mathsf{m}_{\mathsf{v}[i]}) = \sum_{i=0}^{\ell-1} 2^i \left(\mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]} + \lambda^{\mathsf{R}}_{\mathsf{v}[i]} (1 - 2\mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}) \right)$$

where ${\lambda_{\mathsf{v}[i]}}^{\mathsf{R}}, \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$ denote the arithmetic value of bits $\lambda_{\mathsf{v}[i]}, \mathsf{m}_{\mathsf{v}[i]}$ over the ring $\mathbb{Z}_{2^\ell}$. For each bit $\mathsf{v}[i]$ of v, parties generate the $\langle\cdot\rangle$-shares of ${\lambda_{\mathsf{v}[i]}}^{\mathsf{R}}$ in the preprocessing, similar to Π_{bit2A} (Fig. 8.3). During the online phase, additive shares for each bit $\mathsf{v}[i]$ are locally computed similar to Π_{bit2A}. Parties then multiply the ith share with 2^i and locally add up to obtain an additive sharing of v. The rest of the steps are similar to Π_{bit2A}, and the formal protocol appears in Fig. 8.6.

Protocol $\Pi_{\mathsf{B2A}}(\mathcal{P}, {[\![\mathsf{v}]\!]}^{\mathbf{B}})$

Let $\mathsf{v}[i]$ denote the ith bit of v. Let $\mathsf{p}_i = \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$, and $\mathsf{q}_i = \lambda^{\mathsf{R}}_{\mathsf{v}[i]}$.

Preprocessing:

1. For $i \in \{0, 1, \ldots, \ell-1\}$, execute the preprocessing of Π_{bit2A} (Fig. 8.3) for each bit $\mathsf{v}[i]$, to generate $\langle \mathsf{q}_i \rangle = (\mathsf{q}_i^1, \mathsf{q}_i^2, \mathsf{q}_i^3)$.

Online: Let $\mathsf{y}_i = (\mathsf{v}[i])^{\mathsf{R}}$ and y denotes the arithmetic equivalent of v.

1. Locally compute the following:

$$P_1, P_3 : \mathsf{y}^1 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^1 = \sum_{i=0}^{\ell-1} 2^i (\mathsf{p}_i + \mathsf{q}_i^1 (1 - 2\mathsf{p}_i))$$
$$P_2, P_3 : \mathsf{y}^2 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^2 = \sum_{i=0}^{\ell-1} 2^i (\mathsf{q}_i^2 (1 - 2\mathsf{p}_i))$$
$$P_1, P_2 : \mathsf{y}^3 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^3 = \sum_{i=0}^{\ell-1} 2^i (\mathsf{q}_i^3 (1 - 2\mathsf{p}_i))$$

2. $(P_1, P_3), (P_2, P_3), (P_1, P_2)$ execute Π_{JSh} on $\mathsf{y}^1, \mathsf{y}^2, \mathsf{y}^3$ to generate the respective $[\![\cdot]\!]$-shares.

3. Locally compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}^1]\!] + [\![\mathsf{y}^2]\!] + [\![\mathsf{y}^3]\!]$.

Figure 8.6: Boolean to Arithmetic Conversion in SWIFT.

Chapter 9

Tetrad: 4PC Fair and Robust Protocols

This chapter provides details for the Layer II blocks of our 2PC framework Tetrad. Details for the Layer I blocks are provided in chapter 5.

9.1 Building Blocks

9.1.1 Dot Product (Scalar Product)

Given $[\![\vec{\mathbf{a}}]\!], [\![\vec{\mathbf{b}}]\!]$ with $|\vec{\mathbf{a}}| = |\vec{\mathbf{b}}| = \mathsf{d}$, protocol Π_{dotp} (Fig. 9.1) computes $[\![\mathsf{z}]\!]$ such that $\mathsf{z} = (\vec{\mathbf{a}} \odot \vec{\mathbf{b}})^{\mathsf{t}}$ if truncation is enabled, else $\mathsf{z} = \vec{\mathbf{a}} \odot \vec{\mathbf{b}}$. For this, we combine the partial products from the multiplication protocol across d multiplications and communicate them in a single shot. This makes the communication cost of the dot product independent of the vector size. The protocols for robust setting follows similarly from Tetrad-R$^{\mathsf{I}}$ and Tetrad-R$^{\mathsf{II}}$.

Protocol $\Pi_{\mathsf{dotp}}(\vec{\mathbf{a}}, \vec{\mathbf{b}}, \mathsf{isTr})$

Let isTr be a bit that denotes whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\vec{\mathbf{a}}]\!], [\![\vec{\mathbf{b}}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \vec{\mathbf{a}} \odot \vec{\mathbf{b}} = \sum_{i=1}^{\mathsf{d}} \mathsf{a}_i \mathsf{b}_i$.

Preprocessing:

1. Locally compute the following:

$$P_0, P_1 : \gamma^1_{\vec{\mathbf{a}}\vec{\mathbf{b}}} = \sum_{i=1}^{\mathsf{d}} (\lambda^1_{\mathsf{a}_i} \lambda^3_{\mathsf{b}_i} + \lambda^3_{\mathsf{a}_i} \lambda^1_{\mathsf{b}_i} + \lambda^3_{\mathsf{a}_i} \lambda^3_{\mathsf{b}_i})$$

$$P_0, P_2 : \gamma^2_{\vec{\mathsf{ab}}} = \sum_{i=1}^{\mathsf{d}} (\lambda^2_{\mathsf{a}_i}\lambda^3_{\mathsf{b}_i} + \lambda^3_{\mathsf{a}_i}\lambda^2_{\mathsf{b}_i} + \lambda^2_{\mathsf{a}_i}\lambda^2_{\mathsf{b}_i})$$

$$P_0, P_3 : \gamma^3_{\vec{\mathsf{ab}}} = \sum_{i=1}^{\mathsf{d}} (\lambda^1_{\mathsf{a}_i}\lambda^2_{\mathsf{b}_i} + \lambda^2_{\mathsf{a}_i}\lambda^1_{\mathsf{b}_i} + \lambda^1_{\mathsf{a}_i}\lambda^1_{\mathsf{b}_i})$$

2. P_0, P_3 and P_j sample random $\mathsf{u}^j \in_R \mathbb{Z}_{2^\ell}$ for $j \in \{1, 2\}$. Let $\mathsf{u}^1 + \mathsf{u}^2 = \gamma^3_{\vec{\mathsf{ab}}} - \mathsf{r}$ for a random $\mathsf{r} \in_R \mathbb{Z}_{2^\ell}$.

3. P_0, P_3 compute $\mathsf{r} = \gamma^3_{\vec{\mathsf{ab}}} - \mathsf{u}^1 - \mathsf{u}^2$ and set $\mathsf{q} = \mathsf{r}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$, else set $\mathsf{q} = \mathsf{r}$. P_0, P_3 execute $\Pi_{\mathsf{JSh}}(P_0, P_3, \mathsf{q})$ to generate $[\![\mathsf{q}]\!]$.

4. P_0, P_1, P_2 sample random $\mathsf{s}_1, \mathsf{s}_2 \in_R \mathbb{Z}_{2^\ell}$ and set $\mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2$[a]. P_0 sends $\mathsf{w} = \gamma^1_{\vec{\mathsf{ab}}} + \gamma^2_{\vec{\mathsf{ab}}} + \mathsf{s}$ to P_3.

Online: Let $\mathsf{y} = (\mathsf{z} - \mathsf{r}) - \sum_{i=1}^{\mathsf{d}} \mathsf{m}_{\mathsf{a}_i\mathsf{b}_i}$.

1. Locally compute the following:

$$P_1 : \mathsf{y}_1 = \sum_{i=1}^{\mathsf{d}} (-\lambda^1_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i} - \lambda^1_{\mathsf{b}_i}\mathsf{m}_{\mathsf{a}_i}) + \gamma^1_{\vec{\mathsf{ab}}} + \mathsf{u}^1$$

$$P_2 : \mathsf{y}_2 = \sum_{i=1}^{\mathsf{d}} (-\lambda^2_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i} - \lambda^2_{\mathsf{b}_i}\mathsf{m}_{\mathsf{a}_i}) + \gamma^2_{\vec{\mathsf{ab}}} + \mathsf{u}^2$$

$$P_1, P_2 : \mathsf{y}_3 = \sum_{i=1}^{\mathsf{d}} (-\lambda^3_{\mathsf{a}_i}\mathsf{m}_{\mathsf{b}_i} - \lambda^3_{\mathsf{b}_i}\mathsf{m}_{\mathsf{a}_i})$$

2. P_1 sends y_1 to P_2, while P_2 sends y_2 to P_1, and they locally compute $\mathsf{z} - \mathsf{r} = (\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{y}_3) + \sum_{i=1}^{\mathsf{d}} \mathsf{m}_{\mathsf{a}_i\mathsf{b}_i}$.

3. If $\mathsf{isTr} = 1$, P_1, P_2 set $\mathsf{p} = (\mathsf{z} - \mathsf{r})^{\mathsf{t}}$, else $\mathsf{p} = \mathsf{z} - \mathsf{r}$. P_1, P_2 execute $\Pi_{\mathsf{JSh}}(P_1, P_2, \mathsf{p})$ to generate $[\![\mathsf{p}]\!]$.

4. Parties locally compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}]\!] + [\![\mathsf{q}]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

5. *Verification:* P_3 computes $\mathsf{v} = \sum_{i=1}^{\mathsf{d}} (-(\lambda^1_{\mathsf{a}_i} + \lambda^2_{\mathsf{a}_i})\mathsf{m}_{\mathsf{b}_i} - (\lambda^1_{\mathsf{b}_i} + \lambda^2_{\mathsf{b}_i})\mathsf{m}_{\mathsf{a}_i}) + \mathsf{u}^1 + \mathsf{u}^2 + \mathsf{w}$ and sends $\mathsf{H}(\mathsf{v})$ to P_1 and P_2. Parties P_1, P_2 `abort` iff $\mathsf{H}(\mathsf{v}) \neq \mathsf{H}(\mathsf{y}_1 + \mathsf{y}_2 + \mathsf{s})$.

[a] For the fair protocol, it is enough for P_0, P_1, P_2 to sample s directly.

Figure 9.1: Dot Product with / without Truncation in Tetrad.

Lemma 9.1 (Communication) *Protocol Π_{dotp} (Fig. 9.1) (in Tetrad) requires 2ℓ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

Lemma 9.2 (Communication) *Protocol* Π_{dotp} *(in* Tetrad-R^{II}*) requires* 3ℓ *bits of communication in preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

9.1.2 Bit Extraction

To compute most significant bit (msb) of the value v, note that $\mathsf{v} = (\mathsf{m}_\mathsf{v} - \lambda^3_\mathsf{v}) + (-\lambda^1_\mathsf{v} - \lambda^2_\mathsf{v})$ as per the sharing semantics (cf. Table 5.2). P_0, P_3 execute $\Pi^{\mathbf{B}}_{\mathsf{JSh}}$ on $(-\lambda^1_\mathsf{v} - \lambda^2_\mathsf{v})$ during the preprocessing, while P_0, P_3 execute $\Pi^{\mathbf{B}}_{\mathsf{JSh}}$ on $(\mathsf{m}_\mathsf{v} - \lambda^3_\mathsf{v})$ during the online phase to generate the respective boolean sharing. Parties finally compute the result by evaluating the bit extraction circuit [101, 113].

9.1.3 Bit to Arithmetic

Protocol $\Pi_{\mathsf{bit2A}}(\llbracket \mathsf{b} \rrbracket^{\mathbf{B}})$ (Fig. 9.2) enables computing $\llbracket \mathsf{b} \rrbracket$ of a bit b given its boolean sharing $\llbracket \mathsf{b} \rrbracket^{\mathbf{B}}$. Let b^R denotes the value of $\mathsf{b} \in \{0, 1\}$ over the arithmetic ring $\mathbb{Z}_{2^\ell}$. Then for $\mathsf{b} = \mathsf{b}_1 \oplus \mathsf{b}_2$, note that $\mathsf{b}^\mathsf{R} = (\mathsf{b}^\mathsf{R}_1 - \mathsf{b}^\mathsf{R}_2)^2$. Let $\mathsf{b}_1 = \mathsf{m}_\mathsf{b} \oplus \lambda^3_\mathsf{v}$ and $\mathsf{b}_2 = \lambda^1_\mathsf{v} \oplus \lambda^2_\mathsf{v}$. To compute $\llbracket \mathsf{b} \rrbracket$, a pair of parties can generate the arithmetic sharing corresponding to b^R_1 and b^R_2 by executing Π_{JSh}. $\llbracket \mathsf{b} \rrbracket$ can be computed by invoking Π_{Mult} once with inputs $\mathsf{x} = \mathsf{y} = \mathsf{b}^\mathsf{R}_1 - \mathsf{b}^\mathsf{R}_2$.

We obtain a communication-optimized variant by trading off computation in the preprocessing. For this, note that

$$\mathsf{b}^\mathsf{R} = (\mathsf{m}_\mathsf{b} \oplus \lambda_\mathsf{b})^\mathsf{R} = \mathsf{m}^\mathsf{R}_\mathsf{b} + \lambda^\mathsf{R}_\mathsf{b}(1 - 2\mathsf{m}^\mathsf{R}_\mathsf{b}) \tag{9.1}$$

Let $\mathsf{v} = \mathsf{m}^\mathsf{R}_\mathsf{b}$ and $\mathsf{u} = \lambda^\mathsf{R}_\mathsf{b}$. During the preprocessing, P_0 generates $\langle\cdot\rangle$-sharing of u and a check is executed to verify the correctness. The online phase consists of each pair of parties (P_1, P_3), (P_2, P_3) and (P_1, P_2) locally computing an additive sharing of b^R, generating the corresponding $\llbracket\cdot\rrbracket$-sharing using Π_{JSh}, and locally adding the shares to obtain $\llbracket \mathsf{b}^\mathsf{R} \rrbracket$.

For verifying the $\langle\cdot\rangle$-sharing of u by P_0, we let P_3 obtain the bit $(\lambda_\mathsf{b} \oplus \mathsf{r}_\mathsf{b})$ as well as its arithmetic equivalent $(\lambda_\mathsf{b} \oplus \mathsf{r}_\mathsf{b})^\mathsf{R}$ in clear. Here r_b denotes a random bit known to P_0, P_1, P_2. P_3 checks if both the received values are equivalent and raise a complaint if they are inconsistent. To catch a corrupt P_0 from sharing a wrong u value, parties use the $\langle\cdot\rangle$-shares of u to compute $(\lambda_\mathsf{b} \oplus \mathsf{r}_\mathsf{b})^\mathsf{R}$. Moreover, the verification steps are designed in such a way that every value communicated can be locally computed by at least two parties. This enables to use jsnd for communication and hence the desired security guarantee is achieved.

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

Let $\mathsf{u} = \lambda_{\mathsf{b}}^{\mathsf{R}}$ and $\mathsf{v} = \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}$.
Input(s): $[\![\mathsf{b}]\!]^{\mathbf{B}}$, **Output:** $[\![\mathsf{y}]\!] = [\![\mathsf{b}^{\mathsf{R}}]\!]$.

Preprocessing:

1. Generation of $\langle \mathsf{u} \rangle$: P_0, P_3, P_i for $i \in \{1,2\}$ sample u^i. P_0 sends $\mathsf{u}^3 = \mathsf{u} - \mathsf{u}^1 - \mathsf{u}^2$ to P_1, P_2.
2. P_0, P_1, P_2 sample random $\mathsf{r_b} \in \{0,1\}$ and $\mathsf{r} \in \mathbb{Z}_{2^\ell}$.
3. P_1, P_2 jsnd $\lambda_{\mathsf{b}}^3 \oplus \mathsf{r_b}$ to P_3. P_3 locally sets $\lambda_{\mathsf{b}} \oplus \mathsf{r_b} = (\lambda_{\mathsf{b}}^1 \oplus \lambda_{\mathsf{b}}^2) \oplus (\lambda_{\mathsf{b}}^3 \oplus \mathsf{r_b})$.
4. Parties compute: $P_1, P_0 : \mathsf{w}_1 = \mathsf{r}_{\mathsf{b}}^{\mathsf{R}} + (\mathsf{u}^1 + \mathsf{u}^3)(1 - 2\mathsf{r}_{\mathsf{b}}^{\mathsf{R}}) + \mathsf{r}$, $\quad P_2, P_0 : \mathsf{w}_2 = (\mathsf{u}^2)(1 - 2\mathsf{r}_{\mathsf{b}}^{\mathsf{R}}) - \mathsf{r}$.
5. P_1, P_0 jsnd w_1 to P_3, while P_2, P_0 jsnd $\mathsf{H}(\mathsf{w}_2)$ to P_3.
6. P_3 sets $\mathsf{flag} = \texttt{continue}$ if $\mathsf{H}((\lambda_{\mathsf{b}} \oplus \mathsf{r}_b)^{\mathsf{R}} - \mathsf{w}_1) = \mathsf{H}(\mathsf{w}_2)$, else $\mathsf{flag} = \texttt{abort}$. P_3 sends flag to P_0, P_1, P_2. Parties mutually exchange the flag and accept the value that forms the majority.
7. For robust setting, if $\mathsf{flag} = \texttt{abort}$, then $\mathsf{TTP} = P_1$ (or P_2).

Online: Let $\mathsf{y} = \mathsf{b}^{\mathsf{R}}$.

1. Locally compute the following:

$$P_1, P_3 : \mathsf{y}_1 = \mathsf{v} + \mathsf{u}^1(1 - 2\mathsf{v}) \quad \Big| \quad P_2, P_3 : \mathsf{y}_2 = \mathsf{u}^2(1 - 2\mathsf{v}) \quad \Big| \quad P_1, P_2 : \mathsf{y}_3 = \mathsf{u}^3(1 - 2\mathsf{v})$$

2. $(P_1, P_3), (P_2, P_3), (P_1, P_2)$ execute Π_{JSh} on $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ to generate the respective $[\![\cdot]\!]$-shares.
3. Compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}_1]\!] + [\![\mathsf{y}_2]\!] + [\![\mathsf{y}_3]\!]$.

Figure 9.2: Bit to Arithmetic conversion in Tetrad.

Lemma 9.3 (Communication) *Protocol* Π_{bit2A} *(Fig. 9.2) requires* $3\ell + 1$ *bits of communication in preprocessing, and* 1 *round and* 3ℓ *bits of communication in the online phase.*

Proof: During preprocessing, generation of $\langle \mathsf{u} \rangle$ involves communication of ℓ bits from P_0 to each of P_1, P_2. As part of verification, two instances of jsnd are executed, one on 1 bit and other on ℓ bits. The communication for hash gets amortized over multiple instances. The online phase involves three instances of joint sharing protocol resulting in 1 rounds and a communication of 3ℓ bits. The costs follow from Lemma 5.1. □

9.1.3.1 Bit to Arithmetic:II

Similar to Π_{bit2A} protocol, given the boolean sharings $[\![\mathsf{b}_1]\!]^{\mathbf{B}}, [\![\mathsf{b}_2]\!]^{\mathbf{B}}$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1-2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1})$, $(1-2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2})$ respectively. Using (9.1), we can write

$$\begin{aligned}(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}} = (\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_1}\Delta_{\mathsf{b}_1})(\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\Delta_{\mathsf{b}_2}) \\ &= \mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2} + \lambda^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2}\Delta_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \end{aligned} \tag{9.2}$$

During preprocessing, the $\langle\cdot\rangle$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}_1}$ and $\lambda^{\mathsf{R}}_{\mathsf{b}_2}$ are computed similar to that of Π_{bit2A} (Fig. 9.2). Once the $\langle\cdot\rangle$-shares are generated, parties invoke the Π_{MultS} (Fig. 5.4) on $\langle\lambda^{\mathsf{R}}_{\mathsf{b}_1}\rangle$ and $\langle\lambda^{\mathsf{R}}_{\mathsf{b}_2}\rangle$ to generate the $\langle\cdot\rangle$-shares of $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$. The online phase is similar to that of Π_{bit2A} protocol.

Lemma 9.4 (Communication) *Protocol Π_{dbit2A} requires $9\ell+2$ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

9.1.4 Bit Injection

Given the boolean 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 $\mathsf{b}^{\mathsf{R}}\mathsf{v}$. Let Δ_{b} denote the value $(1-2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}})$. Similar to Π_{bit2A},

$$\begin{aligned}\mathsf{b}^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) = (\mathsf{m}^{\mathsf{R}}_{\mathsf{b}} + \lambda^{\mathsf{R}}_{\mathsf{b}}\Delta_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\ &= \mathsf{m}^{\mathsf{R}}_{\mathsf{b}}\mathsf{m}_{\mathsf{v}} - \mathsf{m}^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}} + \lambda^{\mathsf{R}}_{\mathsf{b}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}} - \lambda^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}}\Delta_{\mathsf{b}} \end{aligned} \tag{9.3}$$

During the preprocessing, P_0 generates the $\langle\cdot\rangle$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}}$ similar to Π_{bit2A} protocol. Parties then invoke the Π_{MultS} (Fig. 5.4) on $\langle\lambda^{\mathsf{R}}_{\mathsf{b}}\rangle$ and $\langle\lambda_{\mathsf{v}}\rangle$ to generate the $\langle\cdot\rangle$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}}$. During the online phase, (P_1, P_3), (P_2, P_3) and (P_1, P_2) compute an additive sharing of $\mathsf{b}^{\mathsf{R}}\mathsf{v}$ using (9.3) and execute Π_{JSh} on them to generate the respective $[\![\cdot]\!]$-shares. Parties locally add the shares to obtain the output.

Lemma 9.5 (Communication) *Protocol Π_{bitInj} requires $6\ell+1$ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

9.1.4.1 Sum of Bit Injections

Given m pair of values in the shared form, $\{[\![\mathsf{b}_i]\!]^{\mathbf{B}}, [\![\mathsf{v}_i]\!]\}_{i\in[m]}$, the goal of Π_{bitInjS} is to compute the $[\![\cdot]\!]$-share of $\mathsf{z} = \sum_{i=1}^{m} \mathsf{b}^{\mathsf{R}}_i \cdot \mathsf{v}_i$. For this, parties execute the preprocessing corresponding to m bit injections of the form $\mathsf{b}^{\mathsf{R}}_i \cdot \mathsf{v}_i$.

In the online phase, each pair of parties (P_1, P_3), (P_2, P_3) and (P_1, P_2) locally compute an additive sharing of z_i, corresponding to $\mathsf{b}_i^{\mathsf{R}} \cdot \mathsf{v}_i$ first. Instead of generating the $[\![\cdot]\!]$-sharing for each of the m terms, parties locally add the shares and execute Π_{JSh} on the result. Concretely, parties locally compute $\mathsf{z}^j = \sum_{i=1}^{m} \mathsf{z}_i^j$ for $j \in \{1,2,3\}$ and execute Π_{JSh} on z^j to obtain its $[\![\cdot]\!]$-sharing. Finally, parties locally add up the shares similar to Π_{bitInj} protocol. This results in an online communication independent of m.

Lemma 9.6 (Communication) *Protocol Π_{bitInjS} requires $m \cdot (6\ell + 1)$ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

9.1.4.2 Bit Injection:Π

Similar to Π_{bitInj} protocol, given $[\![\mathsf{b}_1]\!]^{\mathbf{B}}, [\![\mathsf{b}_2]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1 - 2\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}})$, $(1 - 2\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}})$ respectively. Using (9.2) and (9.3), we can write

$$\begin{aligned}
(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\
&= (\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_1})(\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_2})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\
&= \mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_1} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \\
&\quad - \lambda_{\mathsf{v}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} - \lambda_{\mathsf{b}_1}^{\mathsf{R}}\lambda_{\mathsf{v}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_1} - \lambda_{\mathsf{b}_2}^{\mathsf{R}}\lambda_{\mathsf{v}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_2} - (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_{\mathsf{v}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \qquad (9.4)
\end{aligned}$$

During preprocessing, the $\langle\cdot\rangle$-shares of $\lambda_{\mathsf{b}_1}^{\mathsf{R}}$, $\lambda_{\mathsf{b}_2}^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$ are computed similar to that of Π_{dbit2A}. Once the $\langle\cdot\rangle$-shares are generated, parties invoke the Π_{MultS} (Fig. 5.4) on $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$ and $\langle\lambda_{\mathsf{v}}\rangle$ to generate the $\langle\cdot\rangle$-shares of $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_{\mathsf{v}}$. Similarly, parties compute $\langle\lambda_{\mathsf{b}_1}^{\mathsf{R}}\lambda_{\mathsf{v}}\rangle$ and $\langle\lambda_{\mathsf{b}_2}^{\mathsf{R}}\lambda_{\mathsf{v}}\rangle$ using two instances of Π_{MultS}. The online phase is similar to that of Π_{bitInj} protocol.

Lemma 9.7 (Communication) *Protocol Π_{dbitInj} requires $18\ell + 2$ bits of communication in preprocessing, and 1 round and 3ℓ bits of communication in the online phase.*

9.1.5 Equality Test (Π_{eq})

To check whether $\mathsf{a} \stackrel{?}{=} \mathsf{b}$ or not, given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$, Π_{eq} proceeds with parties locally computing $[\![\mathsf{y}]\!] = [\![\mathsf{a}]\!] - [\![\mathsf{b}]\!]$. According to our sharing semantics, y can be written as $\mathsf{y} = \mathsf{y}_1 - \mathsf{y}_2$ where $\mathsf{y}_1 = \mathsf{m}_{\mathsf{y}} - \lambda_{\mathsf{y}}^3$ and $\mathsf{y}_2 = \lambda_{\mathsf{y}}^1 + \lambda_{\mathsf{y}}^2$. Parties (P_1, P_2) and (P_0, P_3) generate $[\![\mathsf{y}_1]\!]^{\mathbf{B}}$ and $[\![\mathsf{y}_2]\!]^{\mathbf{B}}$ resepctively using the joint sharing protocol Π_{JSh}. Note that $\mathsf{a} = \mathsf{b}$ implies $\mathsf{y}_1 = \mathsf{y}_2$ and hence all the bits of $\mathsf{v} = \overline{(\mathsf{y}_1 \oplus \mathsf{y}_2)}$ should be 1. As mentioned in the introduction of Part II (II), parties use four input AND gates and a tree structure, where 4 bits are taken at a time and the AND of them is computed in one go.

9.2 Mixed Protocol Framework

Table 9.1 compares our sharing conversions with Trident [38]. For uniformity, we consider a function, F, to be computed on an ℓ-bit inputs x, y using a garbled circuit (GC) in the mixed framework, which gives an ℓ-bit output $\mathsf{z} = \mathsf{F}(\mathsf{x}, \mathsf{y})$, where ℓ denotes the ring size in bits. Let G^{F} denote the corresponding GC. In the table, G^{Sn} denotes a n-input garbled subtraction circuit; G^{An} denotes n-input garbled addition circuit; $\hat{\mathsf{G}}$ denotes the garbled circuit with decoding information; $\mathsf{G}^{\mathsf{n}_1\times 1,\ldots,\mathsf{n}_m\times \mathsf{m}}$ denotes n_i instances of GC G^{i} for $i \in \{1,\ldots,\mathsf{m}\}$ and $|\mathsf{G}^{\mathsf{n}_1\times 1,\ldots,\mathsf{n}_m\times \mathsf{m}}|$ denotes its size.

Variant[a]	Conversion[b]	Trident [38]			Tetrad		
		Comm.$_{\text{pre}}$	Comm.$_{\text{on}}$	Rounds$_{\text{on}}$	Comm.$_{\text{pre}}$	Comm.$_{\text{on}}$	Rounds$_{\text{on}}$
2 GC	A-G-A A-G-B B-G-A B-G-B	$(6\ell\kappa+\ell)$ + $2\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert$ $2\lvert\mathsf{G}^{\mathsf{S2},\mathsf{F}}\rvert$ $2\lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert$ $2\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$4\ell\kappa+2\ell$	2	$(6\ell\kappa+\ell)$ + $2\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert$ $2\lvert\mathsf{G}^{\mathsf{S2},\mathsf{F}}\rvert$ $2\lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert$ $2\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$4\ell\kappa+\ell$	1
1 GC	A-G-A A-G-B B-G-A B-G-B	$(3\ell\kappa+\ell)$ + $\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert$ $\lvert\mathsf{G}^{\mathsf{S2},\mathsf{F}}\rvert$ $\lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert$ $\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$2\ell\kappa+3\ell$	2	$(3\ell\kappa+\ell)$ + $\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert$ $\lvert\mathsf{G}^{\mathsf{S2},\mathsf{F}}\rvert$ $\lvert\hat{\mathsf{G}}^{\mathsf{S2},\mathsf{F}}\rvert$ $\lvert\mathsf{G}^{\mathsf{F}}\rvert$	$2\ell\kappa+2\ell$	2
Others[c]	A-B B-A	$2\ell+3\ell\log\ell$ $3\ell^2\ell$	$\ell+3\ell\log\ell$ 3ℓ	$1+\log\ell$ 1	$\mathsf{u}_1+\ell$ $3\ell^2+\ell$	$3\mathsf{u}_2+\ell$ 3ℓ	$\log_4\ell$ 1

[a] Notations: ℓ - size of ring in bits, κ - computational security parameter, 'pre' - preprocessing, 'on' - online.
[b] 'A' - arithmetic, 'B' - boolean, 'G' - Garbled.
[c] $\mathsf{u}_1 = 2\mathsf{n}_2 + 8\mathsf{n}_3 + 22\mathsf{n}_4$, $\mathsf{u}_2 = \mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4$ denote the number of AND gates in the optimized adder circuit [113] with 2, 3, 4 inputs, respectively. For $\ell = 64$, $\mathsf{n}_2 = 216, \mathsf{n}_3 = 184, \mathsf{n}_4 = 179$.

Table 9.1: Mixed protocol conversions of Trident [38] and Tetrad.

9.2.1 Conversions involving Garbled World

Assume the GC is required to compute a function f on inputs $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and let the output be $f(\mathsf{x}, \mathsf{y})$. All the conversions described are for the 2 GC variant. Conversions for the 1 GC variant are straightforward, hence we omit the details.

Case I: Boolean-Garbled-Boolean Since the inputs to the GC are available in boolean form, say $[\![\mathsf{x}]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!]^{\mathbf{B}}$, parties generate $[\![\mathsf{x}]\!]^{\mathbf{C}}, [\![\mathsf{y}]\!]^{\mathbf{C}}$ by invoking the garbled sharing protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$. Additionally, parties P_0, P_3 sample $\mathsf{R} \in \mathbb{Z}_{2^\ell}$ to mask the function output, $f(\mathsf{x}, \mathsf{y})$, and generate $[\![\mathsf{R}]\!]^{\mathbf{B}}$ (using the joint sharing protocol) and $[\![\mathsf{R}]\!]^{\mathbf{G}}$. Garblers $P_g \in \{P_0, P_2, P_3\}$ garble the circuit which computes $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) \oplus \mathsf{R}$, and send the GC along with the decoding information to

evaluator P_1. Analogous steps are performed for evaluator P_2. Upon GC evaluation and output decoding, evaluators obtain $\mathsf{z} = f(\mathsf{x},\mathsf{y}) \oplus \mathsf{R}$, and jointly boolean share z to generate $[\![\mathsf{z}]\!]^{\mathbf{B}}$. Parties then compute $[\![f(\mathsf{x},\mathsf{y})]\!]^{\mathbf{B}} = [\![\mathsf{z}]\!]^{\mathbf{B}} \oplus [\![\mathsf{R}]\!]^{\mathbf{B}}$.

Case II: Boolean-Garbled-Arithmetic This is similar to *Case I* except that the circuit which computes $\mathsf{z} = f(\mathsf{x},\mathsf{y}) + \mathsf{R}$ is garbled instead. Boolean sharing of z is replaced with arithmetic, followed by computing $[\![f(\mathsf{x},\mathsf{y})]\!] = [\![\mathsf{z}]\!] - [\![\mathsf{R}]\!]$.

Cases III & IV: Input in Arithmetic Sharing The function to be computed $f(\mathsf{x},\mathsf{y})$, is modified as $f'(\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}, \mathsf{m}_\mathsf{y}, \alpha_\mathsf{y}, \lambda^3_\mathsf{y}) = f(\mathsf{m}_\mathsf{x} - \alpha_\mathsf{x} - \lambda^3_\mathsf{x}, \mathsf{m}_\mathsf{y} - \alpha_\mathsf{y} - \lambda^3_\mathsf{y})$ where inputs x,y are replaced by the triples $\{\mathsf{m}_\mathsf{x}, \alpha_\mathsf{x}, \lambda^3_\mathsf{x}\}, \{\mathsf{m}_\mathsf{y}, \alpha_\mathsf{y}, \lambda^3_\mathsf{y}\}$ and $\alpha_\mathsf{x} = \lambda^1_\mathsf{x} + \lambda^2_\mathsf{x}$ and $\alpha_\mathsf{y} = \lambda^1_\mathsf{y} + \lambda^2_\mathsf{y}$. The circuit to be garbled thus, corresponds to the function f'. Parties generate $[\![\mathsf{m}_\mathsf{x}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{x}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{x}]\!]^{\mathbf{G}}$, $[\![\mathsf{m}_\mathsf{y}]\!]^{\mathbf{G}}, [\![\alpha_\mathsf{y}]\!]^{\mathbf{G}}, [\![\lambda^3_\mathsf{y}]\!]^{\mathbf{G}}$ via $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$, following which, parties proceed with the rest of the computation whose steps are similar to *Case I*, and *II*, depending on the requirement on the output sharing.

9.2.2 Other Conversions

Arithmetic to Boolean To convert arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to boolean sharing, observe that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$ where $\mathsf{v}_1 = \mathsf{m}_\mathsf{v} - \lambda^3_\mathsf{v}$ is possessed by parties P_1, P_2, while $\mathsf{v}_2 = -(\lambda^1_\mathsf{v} + \lambda^2_\mathsf{v})$ is possessed by parties P_0, P_3. Thus, $[\![\mathsf{v}]\!]^{\mathbf{B}}$ can be computed as $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} + [\![\mathsf{v}_2]\!]^{\mathbf{B}}$, where $[\![\mathsf{v}_2]\!]^{\mathbf{B}}$ can be generated in the preprocessing phase, and $[\![\mathsf{v}_1]\!]^{\mathbf{B}}$ can be generated in the online phase by the respective parties executing joint boolean sharing protocol. The protocol appears in Fig. 9.3. Boolean addition, when instantiated using the adder of ABY2.0 [113], requires $\log_4(\ell)$ rounds.

Protocol Π_{A2B}

Preprocessing: P_0, P_3 execute joint boolean sharing to generate $[\![\mathsf{v}_2]\!]^{\mathbf{B}}$, where $\mathsf{v}_2 = -(\lambda^1_\mathsf{v} + \lambda^2_\mathsf{v})$.

Online:

1. P_1, P_2 execute joint boolean sharing to generate $[\![\mathsf{v}_1]\!]^{\mathbf{B}}$, where $\mathsf{v}_1 = \mathsf{m}_\mathsf{v} - \lambda^3_\mathsf{v}$.
2. Parties obtain $[\![\mathsf{v}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} + [\![\mathsf{v}_2]\!]^{\mathbf{B}}$ using a boolean adder circuit.

Figure 9.3: Arithmetic to Boolean Conversion in Tetrad.

Boolean to Arithmetic To convert a boolean sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ into an arithmetic sharing, note that

$$\mathsf{v} = \sum_{i=0}^{\ell-1} 2^i \mathsf{v}[i] = \sum_{i=0}^{\ell-1} 2^i (\lambda_{\mathsf{v}[i]} \oplus \mathsf{m}_{\mathsf{v}[i]}) = \sum_{i=0}^{\ell-1} 2^i \left(\mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]} + \lambda^{\mathsf{R}}_{\mathsf{v}[i]} (1 - 2\mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}) \right)$$

where ${\lambda_{\mathsf{v}[i]}}^{\mathsf{R}}, \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$ denote the arithmetic value of bits $\lambda_{\mathsf{v}[i]}, \mathsf{m}_{\mathsf{v}[i]}$ over the ring $\mathbb{Z}_{2^\ell}$. For each bit $\mathsf{v}[i]$ of v, parties generate the $\langle\cdot\rangle$-shares of ${\lambda_{\mathsf{v}[i]}}^{\mathsf{R}}$ in the preprocessing, similar to Π_{bit2A} (Fig. 9.2). During the online phase, additive shares for each bit $\mathsf{v}[i]$ are locally computed similar to Π_{bit2A}. Parties then multiply the ith share with 2^i and locally add up to obtain an additive sharing of v. The rest of the steps are similar to Π_{bit2A}, and the formal protocol appears in Fig. 9.4.

Protocol $\Pi_{\mathsf{B2A}}(\mathcal{P}, {[\![\mathsf{v}]\!]}^{\mathbf{B}})$

Let $\mathsf{v}[i]$ denote the ith bit of v. Let $\mathsf{p}_i = \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$, and $\mathsf{q}_i = \lambda^{\mathsf{R}}_{\mathsf{v}[i]}$.

Preprocessing:

1. For $i \in \{0, 1, \ldots, \ell-1\}$, execute the preprocessing of Π_{bit2A} (Fig. 9.2) for each bit $\mathsf{v}[i]$, to generate $\langle \mathsf{q}_i \rangle = (\mathsf{q}_i^1, \mathsf{q}_i^2, \mathsf{q}_i^3)$.

Online: Let $\mathsf{y}_i = (\mathsf{v}[i])^{\mathsf{R}}$ and y denotes the arithmetic equivalent of v.

1. Locally compute the following:

$$P_1, P_3 : \mathsf{y}^1 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^1 = \sum_{i=0}^{\ell-1} 2^i (\mathsf{p}_i + \mathsf{q}_i^1 (1 - 2\mathsf{p}_i))$$
$$P_2, P_3 : \mathsf{y}^2 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^2 = \sum_{i=0}^{\ell-1} 2^i (\mathsf{q}_i^2 (1 - 2\mathsf{p}_i))$$
$$P_1, P_2 : \mathsf{y}^3 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}_i^3 = \sum_{i=0}^{\ell-1} 2^i (\mathsf{q}_i^3 (1 - 2\mathsf{p}_i))$$

2. $(P_1, P_3), (P_2, P_3), (P_1, P_2)$ execute Π_{JSh} on $\mathsf{y}^1, \mathsf{y}^2, \mathsf{y}^3$ to generate the respective $[\![\cdot]\!]$-shares.

3. Locally compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}^1]\!] + [\![\mathsf{y}^2]\!] + [\![\mathsf{y}^3]\!]$.

Figure 9.4: Boolean to Arithmetic Conversion in Tetrad.

We remark that the protocol Π_{B2A} can be used to efficiently generate edaBits [55] in our setting. For this, the parties non-interactively generate the boolean sharing for ℓ-bits and perform the Π_{B2A} conversion to obtain the equivalent arithmetic value.

Chapter 10

ABY2.0: 2PC Semi-honest Blocks

This chapter provides details for the Layer II blocks of our 2PC framework ABY2.0. Details for the Layer I blocks are provided in chapter 6.

10.1 Building Blocks

10.1.1 Dot Product (Scalar Product)

Given $[\![\vec{\mathsf{a}}]\!], [\![\vec{\mathsf{b}}]\!]$ with $|\vec{\mathsf{a}}| = |\vec{\mathsf{b}}| = \mathsf{d}$, protocol Π_{dotp} (Fig. 10.1) computes $[\![\mathsf{z}]\!]$ such that $\mathsf{z} = (\vec{\mathsf{a}} \odot \vec{\mathsf{b}})^{\mathsf{t}}$ if truncation is enabled, else $\mathsf{z} = \vec{\mathsf{a}} \odot \vec{\mathsf{b}}$. The protocol is similar to the multiplication protocol Π_{Mult} (Fig. 6.2) except that the parties combine the partial products in the online phase across d multiplications and communicate them in a single shot. This makes the communication cost of the dot product in the online phase independent of the vector size.

Protocol $\Pi_{\mathsf{dotp}}(\vec{\mathsf{a}}, \vec{\mathsf{b}}, \mathsf{isTr})$

isTr is a bit denoting whether truncation is required ($\mathsf{isTr} = 1$) or not ($\mathsf{isTr} = 0$).
Input(s): $[\![\vec{\mathsf{a}}]\!], [\![\vec{\mathsf{b}}]\!]$.
Output: $[\![\mathsf{o}]\!]$ where $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and $\mathsf{o} = \mathsf{z}$ if $\mathsf{isTr} = 0$ and $\mathsf{z} = \vec{\mathsf{a}} \odot \vec{\mathsf{b}} = \sum_{j=1}^{\mathsf{d}} \mathsf{a}_j \mathsf{b}_j$.

Preprocessing: Execute Π_{MultPre} on $\left[\lambda_{\mathsf{a}_j}\right]$ and $\left[\lambda_{\mathsf{b}_j}\right]$ to generate $\left[\gamma_{\mathsf{a_j b_j}}\right]$ for $j \in [\mathsf{d}]$.

Online:

1. Locally compute:

$$P_1 : \mathsf{z}_1 = \sum_{j=1}^{\mathsf{d}} (\mathsf{m}_{\mathsf{a_j b_j}} - \lambda^1_{\mathsf{a}_j} \mathsf{m}_{\mathsf{b}_j} - \lambda^1_{\mathsf{b}_j} \mathsf{m}_{\mathsf{a}_j} + \left[\gamma_{\mathsf{a_j b_j}}\right]_1)$$

$$P_2 : \mathsf{z}_2 = \sum_{j=1}^{\mathsf{d}} (-\lambda^2_{\mathsf{a}_j} \mathsf{m}_{\mathsf{b}_j} - \lambda^2_{\mathsf{b}_j} \mathsf{m}_{\mathsf{a}_j} + \left[\gamma_{\mathsf{a}_j \mathsf{b}_j}\right]_2)$$

2. If $\mathsf{isTr} = 1$, P_i sets $\mathsf{p}_i = \mathsf{z}^{\mathsf{t}}_i$, else $\mathsf{p}_i = \mathsf{z}_i$ where $i \in \{1, 2\}$. Execute $\Pi_{\mathsf{Sh}}(P_i, \mathsf{p}_i)$ to generate $[\![\mathsf{p}_i]\!]$.

3. Compute $[\![\mathsf{o}]\!] = [\![\mathsf{p}_1]\!] + [\![\mathsf{p}_2]\!]$. Here $\mathsf{o} = \mathsf{z}^{\mathsf{t}}$ if $\mathsf{isTr} = 1$ and z otherwise.

Figure 10.1: Dot Product with / without Truncation in ABY2.0.

Lemma 10.1 (Communication) *Protocol* Π_{dotp} *(Fig. 10.1) (in ABY2.0) requires* $2\mathsf{d}\ell(\kappa + \ell)$ *bits of communication in the preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

10.1.2 Bit Extraction

To compute most significant bit (msb) of the value v, note that $\mathsf{v} = (\mathsf{m}_\mathsf{v} - \lambda^1_\mathsf{v}) + (-\lambda^2_\mathsf{v})$ as per the sharing semantics (cf. Table 6.2). P_2 generates the boolean sharing of $-\lambda^2_\mathsf{v}$ during the preprocessing, while P_1 generates $[\![(\mathsf{m}_\mathsf{v} - \lambda^1_\mathsf{v})]\!]^{\mathbf{B}}$ during the online phase using sharing protocol. Parties compute the result by evaluating the bit extraction circuit [101, 113].

10.1.3 Bit to Arithmetic

Protocol $\Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$ (Fig. 10.2) enables computing $[\![\mathsf{b}]\!]$ of a bit b given its boolean sharing $[\![\mathsf{b}]\!]^{\mathbf{B}}$. Let b^{R} denotes the value of $\mathsf{b} \in \{0, 1\}$ over the arithmetic ring $\mathbb{Z}_{2^\ell}$. Using our sharing semantics,

$$\mathsf{b}^{\mathsf{R}} = (\mathsf{m}_\mathsf{b} \oplus \lambda_\mathsf{b})^{\mathsf{R}} = \mathsf{m}^{\mathsf{R}}_\mathsf{b} + \lambda^{\mathsf{R}}_\mathsf{b}(1 - 2\mathsf{m}^{\mathsf{R}}_\mathsf{b}) \tag{10.1}$$

During the preprocessing, parties interactively generate $[\cdot]$-sharing of $\lambda^{\mathsf{R}}_\mathsf{b}$ using steps similar to that of Π_{MultPre}. The online phase consists of each P_1 and P_2 locally computing an additive sharing of b^{R}, generating the corresponding $[\![\cdot]\!]$-sharing using Π_{Sh}, and locally adding the shares to obtain $[\![\mathsf{b}]\!]$.

Now we describe how to generate $\left[\lambda^{\mathsf{R}}_\mathsf{b}\right]$ in the preprocessing. Since $\lambda_\mathsf{b} = \lambda^1_\mathsf{b} \oplus \lambda^2_\mathsf{b}$, we can write $\lambda^{\mathsf{R}}_\mathsf{b} = \lambda^{\mathsf{R}}_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_2} - 2\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}$. Parties execute cOT^1_ℓ with P_1 being the sender and P_2 being the receiver. P_1 inputs the correlation $f(x) = x + \lambda^{\mathsf{R}}_{\mathsf{b}_1}$ and obtains $(m_0 = r, m_1 = r + \lambda^{\mathsf{R}}_{\mathsf{b}_1})$. P_2 inputs $c = \lambda^2_\mathsf{b}$ as the choice bit and obtains m_c as output. Now the $[\cdot]$-shares are defined as $\left[\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\right]_1 = -r$ and $\left[\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\right]_2 = m_{\lambda^2_\mathsf{b}}$.

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

Input(s): $[\![\mathsf{b}]\!]^{\mathbf{B}}$, **Output:** $[\![\mathsf{y}]\!] = [\![\mathsf{b}^{\mathsf{R}}]\!]$.

Preprocessing:

1. Generating $[\cdot]$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}$:
   (a) Execute cOT^1_ℓ with P_1 being the sender with input $f(x) = x + \lambda^{\mathsf{R}}_{\mathsf{b}_1}$ and P_2 being the receiver with input $c = \lambda^2_{\mathsf{b}}$.
   (b) P_1 obtains $(m_0 = r, m_1 = r + \lambda^{\mathsf{R}}_{\mathsf{b}_1})$ while P_2 obtains m_c.
   (c) Set $\left[\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\right]_1 = -r$ and $\left[\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\right]_2 = m_c$.
2. P_i for $i \in \{1, 2\}$ locally computes $\left[\lambda^{\mathsf{R}}_{\mathsf{b}}\right]_i = \lambda^{\mathsf{R}}_{\mathsf{b}_i} - 2\left[\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\right]_i$.

Online:

1. Locally compute: $P_1 : \mathsf{y}_1 = \mathsf{m}^{\mathsf{R}}_{\mathsf{b}} + \left[\lambda^{\mathsf{R}}_{\mathsf{b}}\right]_1(1 - 2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}}) \;\Big|\; P_2 : \mathsf{y}_2 = \left[\lambda^{\mathsf{R}}_{\mathsf{b}}\right]_2(1 - 2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}})$
2. P_i for $i \in \{1, 2\}$ executes Π_{Sh} on y_i to generate the respective $[\![\cdot]\!]$-shares.
3. Compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}_1]\!] + [\![\mathsf{y}_2]\!]$.

Figure 10.2: Bit to Arithmetic conversion in ABY2.0.

Lemma 10.2 (Communication) *Protocol* Π_{bit2A} *(Fig. 10.2) requires* $\kappa + \ell$ *bits of communication in preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

Proof: During preprocessing, generation of $\left[\lambda^{\mathsf{R}}_{\mathsf{b}}\right]$ involves one instance of cOT^1_ℓ. The online phase involves two instances of arithmetic sharing protocol in parallel, resulting in 1 round and a communication of 2ℓ bits. □

10.1.3.1 Bit to Arithmetic:II

Similar to Π_{bit2A} protocol, given the boolean sharings $[\![\mathsf{b}_1]\!]^{\mathbf{B}}, [\![\mathsf{b}_2]\!]^{\mathbf{B}}$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1-2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1})$, $(1-2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2})$ respectively. Using (10.1), we can write

$$\begin{aligned}(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}} = (\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_1}\Delta_{\mathsf{b}_1})(\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\Delta_{\mathsf{b}_2}) \\ &= \mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2} + \lambda^{\mathsf{R}}_{\mathsf{b}_1}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_2}\Delta_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\mathsf{m}^{\mathsf{R}}_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \qquad (10.2)\end{aligned}$$

During preprocessing, the $[\cdot]$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}_1}$, and $\lambda^{\mathsf{R}}_{\mathsf{b}_2}$ are computed similar to that of Π_{bit2A} (Fig. 10.2). In parallel, parties execute Π_{MultPre} on the boolean $[\cdot]$-shares of λ_{b_1} and λ_{b_2} to generate $\left[\gamma_{\mathsf{b}_1\mathsf{b}_2}\right] =$

$[\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2}]$ in boolean form. Once $[\gamma_{\mathsf{b}_1\mathsf{b}_2}]$ is generated, parties compute the $[\cdot]$-shares of its arithmetic equivalent similar to that of Π_{bit2A}. The online phase is similar to that of Π_{bit2A} protocol.

Lemma 10.3 (Communication) *Protocol* Π_{dbit2A} *requires* $5\kappa + 3\ell + 2$ *bits of communication in preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

10.1.4 Bit Injection

Given the boolean 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 $\mathsf{b}^{\mathsf{R}}\mathsf{v}$. Let Δ_{b} denote the value $(1 - 2\mathsf{m}^{\mathsf{R}}_{\mathsf{b}})$. Similar to Π_{bit2A},

$$\begin{aligned}\mathsf{b}^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) = (\mathsf{m}^{\mathsf{R}}_{\mathsf{b}} + \lambda^{\mathsf{R}}_{\mathsf{b}}\Delta_{\mathsf{b}})(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\ &= \mathsf{m}^{\mathsf{R}}_{\mathsf{b}}\mathsf{m}_{\mathsf{v}} - \mathsf{m}^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}} + \lambda^{\mathsf{R}}_{\mathsf{b}}\mathsf{m}_{\mathsf{v}}\Delta_{\mathsf{b}} - \lambda^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}}\Delta_{\mathsf{b}} \end{aligned} \tag{10.3}$$

During the preprocessing, parties generate the $[\cdot]$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}}$ similar to Π_{bit2A} protocol. To compute $\lambda^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}}$, one naive method is to multiply $\lambda^{\mathsf{R}}_{\mathsf{b}}$ and λ_{v} using Π_{MultPre}. The cost can be reduced further as follows. Note that

$$\begin{aligned}\lambda^{\mathsf{R}}_{\mathsf{b}}\lambda_{\mathsf{v}} &= (\lambda^{\mathsf{R}}_{\mathsf{b}_1} + \lambda^{\mathsf{R}}_{\mathsf{b}_2} - 2\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2})(\lambda^1_{\mathsf{v}} + \lambda^2_{\mathsf{v}}) \\ &= \lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^1_{\mathsf{v}} + \lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^2_{\mathsf{v}} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda^1_{\mathsf{v}} + \lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda^2_{\mathsf{v}} - 2\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda^1_{\mathsf{v}} - 2\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda^2_{\mathsf{v}} \end{aligned} \tag{10.4}$$

Here P_1 can locally compute $\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^1_{\mathsf{v}}$ while P_2 can compute $\lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda^2_{\mathsf{v}}$. The $[\cdot]$-shares for the remaining four terms can be generated using four instances of cOT^1_ℓ similar to Π_{bit2A} resulting in a communication of $4(\kappa + \ell)$ bits. For instance, to compute $[\cdot]$-shares of $\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^{\mathsf{R}}_{\mathsf{b}_2}\lambda^1_{\mathsf{v}}$, parties engage in an instance of cOT^1_ℓ with P_1 as sender with input $\lambda^{\mathsf{R}}_{\mathsf{b}_1}\lambda^1_{\mathsf{v}}$ and P_2 as receiver with choice bit λ_{b_2}.

During the online phase, P_1 and P_2 compute an additive sharing of $\mathsf{b}^{\mathsf{R}}\mathsf{v}$ and execute Π_{Sh} on them to generate the respctive $[\![\cdot]\!]$-shares.

Lemma 10.4 (Communication) *Protocol* Π_{bitInj} *requires* $5(\kappa + \ell)$ *bits of communication in preprocessing, and* 1 *round and* 2ℓ *bits of communication in the online phase.*

10.1.4.1 Sum of Bit Injections

Given m pair of values in the shared form, $\{[\![\mathsf{b}_i]\!]^{\mathbf{B}}, [\![\mathsf{v}_i]\!]\}_{i\in[m]}$, the goal of Π_{bitInjS} is to compute the $[\![\cdot]\!]$-share of $\mathsf{z} = \sum_{i=1}^{m} \mathsf{b}^{\mathsf{R}}_i \cdot \mathsf{v}_i$. For this, parties execute the preprocessing corresponding to m bit injections of the form $\mathsf{b}^{\mathsf{R}}_i \cdot \mathsf{v}_i$.

In the online phase, each of P_1 and P_2 locally compute an additive sharing of z_i, corresponding to $\mathsf{b}_i^{\mathsf{R}} \cdot \mathsf{v}_\mathsf{i}$ first. Instead of generating the $[\![\cdot]\!]$-sharing for each of the m terms, parties locally add the shares and execute Π_{Sh} on the result. Concretely, parties locally compute $\mathsf{z}^j = \sum_{i=1}^{m} \mathsf{z}_i^j$ for $j \in \{1,2\}$ and execute Π_{Sh} on z^j to obtain its $[\![\cdot]\!]$-sharing. This results in an online communication independent of m.

Lemma 10.5 (Communication) *Protocol Π_{bitInjS} requires $5m(\kappa+\ell)$ bits of communication in preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

10.1.4.2 Bit Injection:II

Similar to Π_{bitInj} protocol, given $[\![\mathsf{b}_1]\!]^{\mathbf{B}}, [\![\mathsf{b}_2]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$, protocol Π_{dbit2A} computes the arithmetic sharing of $(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v}$. Let Δ_{b_1}, Δ_{b_2} denote the value $(1-2\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}})$, $(1-2\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}})$ respectively. Using (10.2) and (10.3), we can write

$$\begin{aligned}
(\mathsf{b}_1\mathsf{b}_2)^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}_1} \oplus \lambda_{\mathsf{b}_1})^{\mathsf{R}}(\mathsf{m}_{\mathsf{b}_2} \oplus \lambda_{\mathsf{b}_2})^{\mathsf{R}}(\mathsf{m}_\mathsf{v} - \lambda_\mathsf{v}) \\
&= (\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_1})(\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_2})(\mathsf{m}_\mathsf{v} - \lambda_\mathsf{v}) \\
&= \mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_\mathsf{v} + \lambda_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_\mathsf{v}\Delta_{\mathsf{b}_1} + \lambda_{\mathsf{b}_2}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_\mathsf{v}\Delta_{\mathsf{b}_2} + (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\mathsf{m}_\mathsf{v}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \\
&\quad - \lambda_\mathsf{v}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}} - \lambda_{\mathsf{b}_1}^{\mathsf{R}}\lambda_\mathsf{v}\mathsf{m}_{\mathsf{b}_2}^{\mathsf{R}}\Delta_{\mathsf{b}_1} - \lambda_{\mathsf{b}_2}^{\mathsf{R}}\lambda_\mathsf{v}\mathsf{m}_{\mathsf{b}_1}^{\mathsf{R}}\Delta_{\mathsf{b}_2} - (\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_\mathsf{v}\Delta_{\mathsf{b}_1}\Delta_{\mathsf{b}_2} \qquad (10.5)
\end{aligned}$$

During preprocessing, the $[\cdot]$-shares of $\lambda_{\mathsf{b}_1}^{\mathsf{R}}, \lambda_{\mathsf{b}_2}^{\mathsf{R}}$ and $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}$ are computed similar to that of Π_{dbit2A}. Once the $[\cdot]$-shares are generated, parties compute $\langle \lambda_{\mathsf{b}_1}^{\mathsf{R}}\lambda_\mathsf{v} \rangle$ and $\langle \lambda_{\mathsf{b}_2}^{\mathsf{R}}\lambda_\mathsf{v} \rangle$ using steps similar to Π_{bitInj}. Using the boolean shares of $\left[\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2}\right]$ computed as part of $\left[(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\right]$ and the $[\cdot]$-shares of λ_v, parties compute the $[\cdot]$-shares of $(\lambda_{\mathsf{b}_1}\lambda_{\mathsf{b}_2})^{\mathsf{R}}\lambda_\mathsf{v}$ similar to protocol Π_{bitInj}. The online phase is similar to that of Π_{bitInj} protocol.

Lemma 10.6 (Communication) *Protocol Π_{dbitInj} requires $14\kappa+12\ell+2$ bits of communication in preprocessing, and 1 round and 2ℓ bits of communication in the online phase.*

10.1.5 Equality Test (Π_{eq})

To check whether $\mathsf{a} \stackrel{?}{=} \mathsf{b}$ or not, given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$, Π_{eq} proceeds with parties locally computing $[\![\mathsf{y}]\!] = [\![\mathsf{a}]\!] - [\![\mathsf{b}]\!]$. According to our sharing semantics, y can be written as $\mathsf{y} = \mathsf{y}_1 - \mathsf{y}_2$ where $\mathsf{y}_1 = \mathsf{m}_\mathsf{y} - \lambda_\mathsf{y}^1$ and $\mathsf{y}_2 = \lambda_\mathsf{y}^2$. P_2 generates $[\![\mathsf{y}_2]\!]^{\mathbf{B}}$ during the preprocessing while P_1 generates $[\![\mathsf{y}_1]\!]^{\mathbf{B}}$ in the online using Π_{Sh}. Note that $\mathsf{a} = \mathsf{b}$ implies $\mathsf{y}_1 = \mathsf{y}_2$ and hence all the bits of $\mathsf{v} = \overline{(\mathsf{y}_1 \oplus \mathsf{y}_2)}$ should be 1. As mentioned in the introduction of Part II (II), parties use four input AND gates and a tree structure, where 4 bits are taken at a time and the AND of them is computed in one go.

10.2 Mixed Protocol Framework

Table 10.1 compares our sharing conversions with ABY [51]. For uniformity, we consider a function, F, to be computed on an ℓ-bit inputs x, y using a garbled circuit (GC) in the mixed framework, which gives an ℓ-bit output $\mathsf{z} = \mathsf{F}(\mathsf{x}, \mathsf{y})$, where ℓ denotes the ring size in bits. Let G^{F} denote the corresponding GC. In the table, G^{Sn} denotes a n-input garbled subtraction circuit; G^{An} denotes n-input garbled addition circuit; $\hat{\mathsf{G}}$ denotes the garbled circuit with decoding information; $\mathsf{G}^{\mathsf{n}_1\times 1,\ldots,\mathsf{n}_m\times \mathsf{m}}$ denotes n_i instances of GC G^{i} for $i \in \{1,\ldots,\mathsf{m}\}$ and $|\mathsf{G}^{\mathsf{n}_1\times 1,\ldots,\mathsf{n}_m\times \mathsf{m}}|$ denotes its size.

Variant[a]	Conversion[b]	ABY [51]			ABY2.0		
		$\text{Comm.}_{\text{pre}}$	Comm._{on}	$\text{Rounds}_{\text{on}}$	$\text{Comm.}_{\text{pre}}$	Comm._{on}	$\text{Rounds}_{\text{on}}$
1 GC	A-G-A	$14\ell\kappa + \lvert\mathsf{G}^{2\times\mathsf{A2},\mathsf{F}}\rvert$	$6\ell\kappa + (\ell^2+7\ell)/2$	4	$(3\ell\kappa+2\ell)$ + $\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{A2},\mathsf{F}}\rvert$ + $\lvert\hat{\mathsf{G}}^{2\times\mathsf{S2},\mathsf{F}}\rvert$ + $\lvert\hat{\mathsf{G}}^{\mathsf{A2},\mathsf{F}}\rvert$ + $\lvert\hat{\mathsf{G}}^{\mathsf{F}}\rvert$	$2\ell\kappa+\ell$	2
	A-G-B	$12\ell\kappa + \lvert\mathsf{G}^{\mathsf{F}}\rvert$	$6\ell\kappa + 2\ell$	2			
	B-G-A	$14\ell\kappa + \lvert\mathsf{G}^{\mathsf{F}}\rvert$	$4\ell\kappa + (\ell^2+7\ell)/2$	4			
	B-G-B	$12\ell\kappa + \lvert\mathsf{G}^{\mathsf{F}}\rvert$	$4\ell\kappa + 2\ell$	2			
Others[c]	A-B	$2\ell\log\ell(\kappa+\ell)$	$4\ell\log\ell$	$\log\ell$	$2\mathsf{u}_1(\kappa+\ell)$	$2\mathsf{u}_2+\ell$	$1+\log_4\ell$
	B-A	$2\ell\kappa$	$(\ell^2+3\ell)/2$	2	$\ell\kappa+\ell^2$	2ℓ	1

[a] Notations: ℓ - size of ring in bits, κ - computational security parameter, 'pre' - preprocessing, 'on' - online.
[b] 'A' - arithmetic, 'B' - boolean, 'G' - Garbled.
[c] $\mathsf{u}_1 = \mathsf{n}_2 + 4\mathsf{n}_3 + 11\mathsf{n}_4$, $\mathsf{u}_2 = \mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4$ denote the number of AND gates in the optimized adder circuit [113] with 2, 3, 4 inputs, respectively. For $\ell = 64$, $\mathsf{n}_2 = 216, \mathsf{n}_3 = 184, \mathsf{n}_4 = 179$.

Table 10.1: Mixed protocol conversions of ABY [51] and ABY2.0.

10.2.1 Conversions involving Garbled World

Assume the GC is required to compute a function f on inputs $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and let the output be $f(\mathsf{x}, \mathsf{y})$.

Case I: Boolean-Garbled-Boolean Since the inputs to the GC are available in boolean form, say $[\![\mathsf{x}]\!]^{\mathbf{B}}, [\![\mathsf{y}]\!]^{\mathbf{B}}$, parties generate $[\![\mathsf{x}]\!]^{\mathbf{C}}, [\![\mathsf{y}]\!]^{\mathbf{C}}$ by invoking the garbled sharing protocol $\Pi^{\mathbf{G}}_{\mathsf{Sh}}$. Additionally, P_1 samples $\mathsf{R} \in \mathbb{Z}_{2^\ell}$ to mask the function output, $f(\mathsf{x}, \mathsf{y})$, and generate $[\![\mathsf{R}]\!]^{\mathbf{B}}$ and $[\![\mathsf{R}]\!]^{\mathbf{G}}$. $P_g = P_1$ garbles the circuit which computes $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) \oplus \mathsf{R}$, and sends the GC along with the decoding information to evaluator P_2. Upon GC evaluation and output decoding, P_2 obtains $\mathsf{z} = f(\mathsf{x}, \mathsf{y}) \oplus \mathsf{R}$, and boolean share z to generate $[\![\mathsf{z}]\!]^{\mathbf{B}}$. Parties then compute $[\![f(\mathsf{x}, \mathsf{y})]\!]^{\mathbf{B}} = [\![\mathsf{z}]\!]^{\mathbf{B}} \oplus [\![\mathsf{R}]\!]^{\mathbf{B}}$.

Case II: Boolean-Garbled-Arithmetic This is similar to *Case I* except that the circuit which computes $\mathsf{z} = f(\mathsf{x},\mathsf{y}) + \mathsf{R}$ is garbled instead. Boolean sharing of z is replaced with arithmetic, followed by computing $[\![f(\mathsf{x},\mathsf{y})]\!] = [\![\mathsf{z}]\!] - [\![\mathsf{R}]\!]$.

Cases III & IV: Input in Arithmetic Sharing The function to be computed $f(\mathsf{x},\mathsf{y})$, is modified as $f'(\mathsf{m}_\mathsf{x}, \lambda^1_\mathsf{x}, \lambda^2_\mathsf{x}, \mathsf{m}_\mathsf{y}, \lambda^1_\mathsf{y}, \lambda^2_\mathsf{y}) = f(\mathsf{m}_\mathsf{x} - \lambda^1_\mathsf{x} - \lambda^2_\mathsf{x}, \mathsf{m}_\mathsf{y} - \lambda^1_\mathsf{y} - \lambda^2_\mathsf{y})$ where inputs x,y are replaced by the sets $\{\mathsf{m}_\mathsf{x}, \lambda^1_\mathsf{x}, \lambda^2_\mathsf{x}, \lambda^3_\mathsf{x}\}$, $\{\mathsf{m}_\mathsf{y}, \lambda^1_\mathsf{y}, \lambda^2_\mathsf{y}, \lambda^3_\mathsf{y}\}$. The circuit to be garbled thus, corresponds to the function f'. Parties generate $[\![\mathsf{m}_\mathsf{x}]\!]^\mathbf{G}, [\![\lambda^1_\mathsf{x}]\!]^\mathbf{G}, [\![\lambda^2_\mathsf{x}]\!]^\mathbf{G}, [\![\mathsf{m}_\mathsf{y}]\!]^\mathbf{G}, [\![\lambda^1_\mathsf{y}]\!]^\mathbf{G}, [\![\lambda^2_\mathsf{y}]\!]^\mathbf{G}$ via $\Pi^\mathbf{G}_\mathsf{Sh}$, following which, parties proceed with the rest of the computation whose steps are similar to *Case I*, and *II*, depending on the requirement on the output sharing. Function f' can be further optimized as $f(\alpha_\mathsf{x} - \lambda^2_\mathsf{x}, \alpha_\mathsf{y} - \lambda^2_\mathsf{y})$ with $\alpha_\mathsf{x} = \mathsf{m}_\mathsf{x} - \lambda^1_\mathsf{x}$ and $\alpha_\mathsf{y} = \mathsf{m}_\mathsf{y} - \lambda^1_\mathsf{y}$. Similar optimization can be done for the other garbling instance as well.

10.2.2 Other Conversions

Arithmetic to Boolean To convert arithmetic sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to boolean sharing, observe that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$ where $\mathsf{v}_1 = \mathsf{m}_\mathsf{v} - \lambda^1_\mathsf{v}$ is possessed by P_1, while $\mathsf{v}_2 = -\lambda^2_\mathsf{v}$ is possessed by P_2. Thus, $[\![\mathsf{v}]\!]^\mathbf{B}$ can be computed as $[\![\mathsf{v}]\!]^\mathbf{B} = [\![\mathsf{v}_1]\!]^\mathbf{B} + [\![\mathsf{v}_2]\!]^\mathbf{B}$. For this, P_2 can generate $[\![\mathsf{v}_2]\!]^\mathbf{B}$ in the preprocessing, and $[\![\mathsf{v}_1]\!]^\mathbf{B}$ can be generated in the online by P_1. The protocol appears in Fig. 10.3. Boolean addition, when instantiated using the adder of [113], requires $\log_4(\ell)$ rounds.

Protocol Π_{A2B}

Preprocessing: P_2 generates $[\![\mathsf{v}_2]\!]^\mathbf{B}$ using Π_Sh, where $\mathsf{v}_2 = -\lambda^2_\mathsf{v}$.

Online:

1. P_1 generates $[\![\mathsf{v}_1]\!]^\mathbf{B}$ using Π_Sh, where $\mathsf{v}_1 = \mathsf{m}_\mathsf{v} - \lambda^1_\mathsf{v}$.
2. Parties obtain $[\![\mathsf{v}]\!]^\mathbf{B} = [\![\mathsf{v}_1]\!]^\mathbf{B} + [\![\mathsf{v}_2]\!]^\mathbf{B}$ using a boolean adder circuit.

Figure 10.3: Arithmetic to Boolean Conversion in ABY2.0.

Boolean to Arithmetic To convert a boolean sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ into an arithmetic sharing, note that

$$\mathsf{v} = \sum_{i=0}^{\ell-1} 2^i \mathsf{v}[i] = \sum_{i=0}^{\ell-1} 2^i (\lambda_{\mathsf{v}[i]} \oplus \mathsf{m}_{\mathsf{v}[i]}) = \sum_{i=0}^{\ell-1} 2^i \left(\mathsf{m}^\mathsf{R}_{\mathsf{v}[i]} + \lambda^\mathsf{R}_{\mathsf{v}[i]}(1 - 2\mathsf{m}^\mathsf{R}_{\mathsf{v}[i]})\right)$$

where $\lambda^{\mathsf{R}}_{\mathsf{v}[i]}, \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$ denote the arithmetic value of bits $\lambda_{\mathsf{v}[i]}, \mathsf{m}_{\mathsf{v}[i]}$ over the ring $\mathbb{Z}_{2^\ell}$. For each bit $\mathsf{v}[i]$ of v, parties generate the $[\cdot]$-shares of $\lambda^{\mathsf{R}}_{\mathsf{v}[i]}$ in the preprocessing, similar to Π_{bit2A} (Fig. 10.2). During the online phase, additive shares for each bit $\mathsf{v}[i]$ are locally computed similar to Π_{bit2A}. Parties then multiply the ith share with 2^i and locally add up to obtain an additive sharing of v. The rest of the steps are similar to Π_{bit2A}, and the formal protocol appears in Fig. 10.4.

Protocol $\Pi_{\mathsf{B2A}}(\mathcal{P}, [\![\mathsf{v}]\!]^{\mathbf{B}})$

Let $\mathsf{v}[i]$ denote the ith bit of v. Let $\mathsf{p}_i = \mathsf{m}^{\mathsf{R}}_{\mathsf{v}[i]}$, and $\mathsf{q}_i = \lambda^{\mathsf{R}}_{\mathsf{v}[i]}$.

Preprocessing:

1. For $i \in \{0, 1, \ldots, \ell - 1\}$, execute the preprocessing of Π_{bit2A} (Fig. 10.2) for each bit $\mathsf{v}[i]$, to generate $[\mathsf{q}_i] = ([\mathsf{q}_i]_1, [\mathsf{q}_i]_2)$.

Online: Let $\mathsf{y}_i = (\mathsf{v}[i])^{\mathsf{R}}$ and y denotes the arithmetic equivalent of v.

1. Locally compute:

$$P_1 : \mathsf{y}^1 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}^1_i = \sum_{i=0}^{\ell-1} 2^i (\mathsf{p}_i + [\mathsf{q}_i]_1 (1 - 2\mathsf{p}_i))$$
$$P_2 : \mathsf{y}^2 = \sum_{i=0}^{\ell-1} 2^i \mathsf{y}^2_i = \sum_{i=0}^{\ell-1} 2^i ([\mathsf{q}_i]_2 (1 - 2\mathsf{p}_i))$$

2. P_j for $j \in \{1, 2\}$ executes Π_{Sh} on y^j to generate the respective $[\![\cdot]\!]$-shares.

3. Compute $[\![\mathsf{y}]\!] = [\![\mathsf{y}^1]\!] + [\![\mathsf{y}^2]\!]$.

Figure 10.4: Boolean to Arithmetic Conversion in ABY2.0.

Part III

Layer III: Applications

Introduction to Layer III

Solutions to privacy-preserving machine learning via MPC have been looked at in various works [102, 101, 133, 110, 37, 38, 85]. Our work considers PPML algorithms such as linear regression, logistic regression, deep neural networks (NN) and support vector machines (SVM) for benchmarking. We consider both the training and inference phases of all the algorithms except SVM. The training phase of SVM requires additional tools and primitives and is out of the scope of this work. We first give an overview of the ML algorithms, followed by the architectural details of the neural networks and support vector machine that we consider for benchmarking and the corresponding datasets.

Overview of ML algorithms

Here we provide an overview of ML algorithms and the detailed benchmarking results. The training phase in most machine learning algorithms consists of two stages– i) forward propagation, where the model computes the output, and ii) backward propagation, where the model parameters are adjusted according to the computed output and the actual output. We define one *iteration* in the training phase as one forward propagation followed by a backward propagation. We refer readers to [102, 101, 50, 110, 38, 134] for formal details.

Linear Regression

For linear regression, one iteration can be viewed as updating the weight vector $\vec{\mathbf{w}}$ using the Gradient Descent algorithm (GD). The update function for $\vec{\mathbf{w}}$ is given by

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

where α denotes the learning rate and $\mathbf{X}_i$ denotes a subset of batch size B, randomly selected from the entire dataset in the ith iteration. Here the forward propagation consists of computing $\mathbf{X}_i \circ \vec{\mathbf{w}}$, while the weight vector is updated in the backward propagation. The update function

consists of a series of matrix multiplications, which can be achieved using dot product protocols. The operations of subtraction, as well as multiplication by a public constant, can be performed locally. We observe that the update function as mentioned above can be computed entirely in the arithmetic domain and can be viewed in the form of $[\![\cdot]\!]$-shares as

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

Logistic Regression

The iteration for the case of logistic regression is similar to that of linear regression, apart from an activation function being applied on $\mathbf{X}_i \circ \vec{\mathbf{w}}$ in the forward propagation. We instantiate the activation function using the sigmoid function. The update function for $\vec{\mathbf{w}}$ is given by

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

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

Neural Networks

A neural network can be divided into various layers, where each layer contains a predefined number of nodes. These nodes are a linear function composed of a non-linear "activation" function. The nodes at the input layer or the first layer are evaluated on the input features to evaluate a neural network. The outputs from these nodes are fed as inputs to the nodes in the next layer. This process is repeated for all the layers to obtain the output. The underlying operation involved is the computation of activation matrices in all the layers. This constitutes the forward propagation phase. The backward propagation involves adjusting model parameters according to the difference between the computed and actual output and comprises computing error matrices.

Concretely, each layer comprises matrix multiplications followed by an application of the ReLU function. The maxpool layer additionally follows convolutional layers after the ReLU layer. After evaluating the layers in a sequential manner, at the output layer, we use the MPC friendly variant of the softmax activation function, $\mathsf{softmax}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum_{j=1}^{n}\mathsf{ReLU}(u_j)}$, proposed by SecureML [102]. To perform the division, we switch from arithmetic to garbled world and then use a division garbled circuit [118] followed by a switch back to the arithmetic world.

The network is trained using the Gradient Descent, where the forward propagation comprises

of computing activation matrices for all the layers in the network. Here, the activation matrix for all the layers except the output, is defined as $\mathbf{A}_i = \mathsf{ReLU}(\mathbf{U}_i)$, where $\mathbf{U}_i = \mathbf{A}_{i-1} \bigodot \mathbf{W}_i$. $\mathbf{A}_0$ is initialized to $\mathbf{X}_j$, where $\mathbf{X}_j$ is a subset of batch size B, randomly selected from the entire dataset for the j$^{\text{th}}$ iteration. The activation matrix for the output layer is defined as $\mathbf{A}_m = \mathsf{softmax}(\mathbf{U}_m)$.

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

Support Vector Machines (inference)

We consider Support Vector Machines (SVM) which is a type of supervised learning algorithm used for classification. SVM is a function which takes as input an n-dimensional *feature vector*, $\vec{\mathbf{x}}$, and outputs the *category* to which the feature vector belongs. SVM is implemented as a matrix $\mathbf{F}$, of dimension $q \times n$ where each row of $\mathbf{F}$ is called the support vector and a vector $\vec{\mathbf{b}} = (b_1, \ldots, b_q)$, is called the *bias*. Each element of $\mathbf{F}$ and $\vec{\mathbf{b}}$ lies in $\mathbb{Z}_{2^\ell}$. Each support vector along with a scalar from the bias can classify the input $\vec{\mathbf{x}}$ into a specific category. More precisely, let $\mathbf{F}_i$ denote the i^{th} row of matrix $\mathbf{F}$. Then, the value $\mathbf{F}_i \cdot \vec{\mathbf{x}} + b_i$ specifies how likely $\vec{\mathbf{x}}$ is to be in category i. To find the most likely category, we compute argmax over these values, i.e. $\text{category}(\vec{\mathbf{x}}) = \text{argmax}_{i \in \{1,\ldots,q\}} \mathbf{F}_i \cdot \vec{\mathbf{x}} + b_i$.

Network architectures

We consider the following networks for benchmarking. These are chosen based on the different range of model parameters and layers used in the network. We refer readers to [134] for a detailed architecture of the neural networks.

1. *SVM:* This consists of 10 categories for classification [50].
2. *NN-1:* This is a fully connected network with 3 layers with ReLU activation after each layer. This network has around 118K parameters and is chosen from [101, 110].
3. *NN-2:* This is a convolutional neural network comprising of 2 hidden layers, with 100 and 10 nodes [120, 101, 38].

4. *NN-3:* This network, called LeNet [91], comprises of 2 convolutional layers and 2 fully connected layers with ReLU activation after each layer, additionally followed by maxpool for convolutional layers. This network has approximately 431K parameters.

5. *NN-4:* This network, called VGG16 [127], was the runner-up of ILSVRC-2014 competition. This network has 16 layers in total and contains fully-connected, convolutional, ReLU activation and maxpool layers. This network has about 138 million parameters.

Datasets.

To benchmark the machine learning algorithms, we use the following real-world datasets:

- MNIST [90] is a collection of 28×28 pixel, handwritten digit images with a label between 0 and 9 for each. It has $60,000$ and respectively, $10,000$ images in training and test set. We evaluate Linear Regression, Logistic Regression, NN-1, NN-3 and SVM on this dataset.

- CIFAR-10 [88] has 32×32 pixel images of 10 different classes such as dogs, horses, etc. It has $50,000$ images for training and $10,000$ for testing, with 6000 images in each class. We evaluate NN-2, NN-4 on this dataset.

Benchmarking Environment Details

The protocols are benchmarked over a Wide Area Network (WAN), instantiated using n1-standard-64 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 2.0 GHz Intel (R) Xeon (R) (Skylake) processors supporting hyper-threading, with 64 vCPUs, and 240 GB of RAM Memory. Parties are connected by pairwise authenticated bidirectional synchronous channels (eg. instantiated via TLS over TCP/IP). We use a limited bandwidth of 40 MBps between every pair of parties and the average round-trip time (rtt)[2] values among the parties are

P_0-P_1	P_0-P_2	P_0-P_3	P_1-P_2	P_1-P_3	P_2-P_3
$153.74ms$	$93.39ms$	$274.84ms$	$62.01ms$	$174.15ms$	$219.46ms$

[1] https://cloud.google.com/
[2] Time for communicating 1 KB of data between a pair of parties

For a fair comparison, we implemented and benchmarked all the protocols, including the protocols of SecureML [102] and ABY3 [101], building on the ENCRYPTO library [45] in C++17. Primitives such as maxpool, which SecureML and ABY3 do not support, have been run using our building blocks. We would like to clarify that our code is developed for benchmarking, is not optimized for industry-grade use, and optimizations like GPU support can enhance performance. Our protocols are instantiated over a 64-bit ring ($\mathbb{Z}_{2^{64}}$), and the collision-resistant hash function is instantiated using SHA-256. We use multi-threading, and our machines are capable of handling a total of 64 threads. Each experiment is run 10 times, and the average values are reported. We use 1 KB = 8192 bits and use a batch size of $B = 128$ for training.

Notation	Description
$\mathsf{T}_{\mathsf{on,i}}$	Online runtime of party P_i.
$\mathsf{T}_{\mathsf{tot,i}}$	Total runtime of party P_i.
$\mathsf{PT}_{\mathsf{on}}$	Protocol online runtime; $\mathsf{max_i}\{\mathsf{T}_{\mathsf{on,i}}\}$.
$\mathsf{PT}_{\mathsf{tot}}$	Protocol total runtime; $\mathsf{max_i}\{\mathsf{T}_{\mathsf{tot,i}}\}$.
$\mathsf{CT}_{\mathsf{on}}$	Cumulative online runtime; $\Sigma_i \mathsf{T}_{\mathsf{on,i}}$.
$\mathsf{CT}_{\mathsf{tot}}$	Cumulative total runtime; $\Sigma_i \mathsf{T}_{\mathsf{tot,i}}$.
$\mathsf{Comm}_{\mathsf{on}}$	Online communication.
$\mathsf{Comm}_{\mathsf{tot}}$	Total communication.
Cost	Total monetary cost.
TP	Online throughput; higher = better (#iterations / #queries per minute in online)

Table 10.2: Benchmarking parameters (lower is better, except for TP)

Benchmarking Parameters

We evaluate the protocols across a variety of parameters as given in Table 10.2. In addition to parameters such as runtime, communication, and *online throughput* (TP) [7, 8, 101, 38], the cumulative runtime (sum of the up-time of all the hired servers) is also reported. This is because when deployed over third-party cloud servers, one pays for them by the communication and the uptime of the hired servers. To analyze the cost of deployment of the framework, *monetary cost* (Cost) [99] is reported. This is done using the pricing of Google Cloud Platform[1], where for 1 GB and 1 hour of usage, the costs are USD 0.08 and USD 3.04, respectively. For protocols with an asymmetric communication graph, communication load is unevenly distributed among

[1]See https://cloud.google.com/vpc/network-pricing for network cost and https://cloud.google.com/compute/vm-instance-pricing for computation cost.

all the servers, leaving several communication channels underutilized. Load balancing improves the performance by running several execution threads in parallel, each with the roles of the servers changed. Load balancing has been performed in all the protocols benchmarked.

Discussion

Broadly speaking, we consider two deployment scenarios – optimized for time (T), and for cost (C). In the first one, participants want the result of the output as soon as possible while maximizing the online throughput. In the second one, they want the overall monetary cost of the system to be minimal and are willing to tolerate an overhead in the execution time. Using multi-input multiplication gates and the 2 GC variant of the garbled makes the online phase faster but incur an increase in monetary cost. This is because they cause an overhead in communication in the preprocessing phase, and communication affects monetary cost more than uptime (in our setting).

Chapter 11

ASTRA: 3PC Semi-honest Applications

$\mathsf{ASTRA}_\mathsf{T}$ uses multi-input multiplication gates and the 2 GC variant of the garbled world and is the fastest variant of the framework. On the other hand, $\mathsf{ASTRA}_\mathsf{C}$ is the variant with a minimal monetary cost. We benchmark our protocols against the 3PC semi-honest framework of ABY3 [101].

11.1 ML Training

We begin with analyzing the benchmarks for linear and logistic regression. Starting with the time-optimized variant, $\mathsf{ASTRA}_\mathsf{T}$ is $2.5-4\times$ faster than ABY3 [101] in online runtime for training. For linear regression, this reduction is observed due to the different rtts among the three parties. This difference vanishes if rtt between every pair of parties is the same. However, the reduction in the online run time for the case of logistic regression is primarily due to the round-optimized bit extraction circuit. Specifically, we use the depth-optimized bit extraction circuit while instantiating the sigmoid activation function using multi-input AND gates. We observe a reduction of up to $2\times$ in communication ($\mathsf{Comm}_\mathsf{tot}$) in $\mathsf{ASTRA}_\mathsf{T}$ over ABY3. This is due to the extra cost required for performing truncation in ABY3. These reductions in communication and run time, coupled with the requirement of one less party in the online phase, directly impact the monetary cost of the system, where $\mathsf{ASTRA}_\mathsf{T}$ brings in a saving of up to 78% over ABY3. On the other hand, the cost-optimized variant $\mathsf{ASTRA}_\mathsf{C}$ is around $1.5\times$ slower in the online phase than $\mathsf{ASTRA}_\mathsf{T}$. However, it is still faster than ABY3 due to the reason discussed above. Further, this variant has $1.3\times$ lesser communication cost compared to $\mathsf{ASTRA}_\mathsf{T}$.

For neural networks, $\mathsf{ASTRA}_\mathsf{T}$ is up to $3.6\times$ faster than ABY3 in the online phase, similar to

Algorithm	Parameter[a]	Training[b]			Inference[c]		
		ABY3	$\mathsf{ASTRA}_\mathsf{T}$	$\mathsf{ASTRA}_\mathsf{C}$	ABY3	$\mathsf{ASTRA}_\mathsf{T}$	$\mathsf{ASTRA}_\mathsf{C}$
Linear Regression	$\mathsf{PT}_{\mathsf{on}}$	0.31	0.12	0.12	0.15	0.06	0.06
	$\mathsf{PT}_{\mathsf{tot}}$	0.32	0.12	0.12	0.15	0.06	0.06
	$\mathsf{CT}_{\mathsf{tot}}$	0.72	0.25	0.25	0.34	0.12	0.12
	$\mathsf{Comm}_{\mathsf{tot}}$	57.12	27.5	27.5	0.05	0.02	0.02
	Cost	0.62	0.21	0.21	0.29	0.1	0.1
	TP	24977.23	37465.85	37465.85	49957.72	74936.58	74936.58
Logistic Regression	$\mathsf{PT}_{\mathsf{on}}$	1.54	0.37	0.56	1.38	0.3	0.48
	$\mathsf{PT}_{\mathsf{tot}}$	1.55	0.37	0.56	1.38	0.3	0.48
	$\mathsf{CT}_{\mathsf{tot}}$	3.48	0.74	1.12	3.08	0.6	0.96
	$\mathsf{Comm}_{\mathsf{tot}}$	76.93	63.5	47.31	0.2	0.3	0.18
	Cost	2.95	0.64	0.9	2.61	0.5	0.81
	TP	4995.45	12488.62	8325.74	5550.82	14987.32	9367.07

[a]Time (in seconds) and communication (in KB) are reported. [b]For training, batch size is 128 and the monetary cost (USD) is reported for 1000 iterations. [c]For inference, cost is reported for 1000 queries.

Table 11.1: Benchmarking of Linear Regression and Logistic Regression algorithms.

the observation in logistic regression. Concerning the communication, $\mathsf{ASTRA}_\mathsf{T}$ has a slightly higher communication than ABY3 for smaller NNs. However, the gap closes for larger NNs. This phenomenon is observed because of the trade-off in the increase in communication due to the use of multi-input multiplication versus the reduction in communication due to the free truncation operation. However, the cost-optimized variant, $\mathsf{ASTRA}_\mathsf{C}$, has a better communication cost than ABY3. Further, $\mathsf{ASTRA}_\mathsf{C}$ is up to 1.4× slower than $\mathsf{ASTRA}_\mathsf{T}$ in terms of online run time, while it is better than ABY3. Note that the requirement of one less party in the online phase coupled with the improvements in communication and run time results in saving up to 87% in the monetary cost of $\mathsf{ASTRA}_\mathsf{C}$ over ABY3, and up to 18% over $\mathsf{ASTRA}_\mathsf{T}$. As the depth increases, we observe that the gap in the monetary cost of $\mathsf{ASTRA}_\mathsf{C}$ and $\mathsf{ASTRA}_\mathsf{T}$ closes in.

These trends can be better captured with a pictorial representation as given in Figure 11.1.

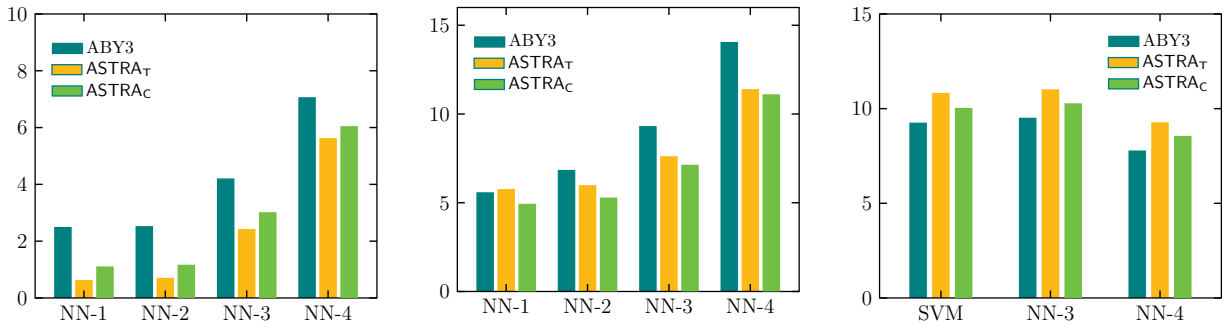
11.2 ML Inference

A similar trend for linear and logistic regression inference is observed for training, where both $\mathsf{ASTRA}_\mathsf{T}$ and $\mathsf{ASTRA}_\mathsf{C}$ outperform ABY3. The exception concerns the slightly higher communication of $\mathsf{ASTRA}_\mathsf{T}$ compared to ABY3 due to the higher communication cost required for multi-input multiplication gates. This difference, however, vanishes for larger circuits, as will be evident from Table 11.2. For neural networks, the time-optimized variant $\mathsf{ASTRA}_\mathsf{T}$ is faster

Algorithm	Parameter[a]	Training[b]			Inference[c]		
		ABY3	$\mathsf{ASTRA_T}$	$\mathsf{ASTRA_C}$	ABY3	$\mathsf{ASTRA_T}$	$\mathsf{ASTRA_C}$
NN-1	$\mathsf{PT_{on}}$	5.66	1.55	2.17	4.15	0.93	1.49
	$\mathsf{PT_{tot}}$	11.36	4.11	4.4	4.16	0.93	1.49
	$\mathsf{CT_{tot}}$	26.97	10.12	8.81	9.29	1.86	2.98
	$\mathsf{Comm_{tot}}$	0.15	0.29	0.15	0.03	0.04	0.03
	Cost	48.49	54.61	30.9	7.81	1.54	2.5
	TP	1160.7	2844.48	2139.75	1850.17	4995.45	3122.15
NN-2	$\mathsf{PT_{on}}$	5.78	1.64	2.26	4.15	0.93	1.49
	$\mathsf{PT_{tot}}$	30.64	4.35	4.98	4.17	0.93	1.49
	$\mathsf{CT_{tot}}$	84.81	11.26	9.96	9.33	1.86	2.98
	$\mathsf{Comm_{tot}}$	0.23	0.34	0.19	0.13	0.18	0.12
	Cost	115.65	63.85	39.15	7.85	1.55	2.51
	TP	225.59	489.56	483.51	1850.17	4995.45	3122.15
NN-3	$\mathsf{PT_{on}}$	18.58	5.42	8.15	10.45	2.23	3.72
	$\mathsf{PT_{tot}}$	157.39	10.88	13.61	10.7	2.24	3.73
	$\mathsf{CT_{tot}}$	458	24.32	27.23	24.12	4.48	7.46
	$\mathsf{Comm_{tot}}$	0.87	1.11	0.74	2.72	4.16	2.53
	Cost	642.07	198	141.1	21.11	4.38	6.69
	TP	14.03	41.78	40.6	734.62	2081.44	1248.86
NN-4	$\mathsf{PT_{on}}$	134.63	49.72	66.54	34.51	7.45	12.29
	$\mathsf{PT_{tot}}$	4753.2	133.31	150.12	39.09	7.59	12.43
	$\mathsf{CT_{tot}}$	14201.97	269.18	300.25	90.91	15.18	24.87
	$\mathsf{Comm_{tot}}$	18.23	15.57	12.27	42.4	61.53	38.1
	Cost	17134.85	2718	2215.6	88.41	22.3	26.9
	TP	0.79	1.96	1.92	222.52	623.45	377.87

[a]Time is reported in seconds [b]For training, communication is reported in GB. Monetary cost (USD) is reported for 1000 iterations and batch size is 128. [c]For inference, communication is reported in MB and the cost is reported for 1000 queries.

Table 11.2: Benchmarking of Neural Networks.



(a) Training: $\mathsf{PT_{on}}$ (b) Training: Cost (c) Inference: TP

Figure 11.1: Analysis of protocols in terms of $\mathsf{PT_{on}}$, Cost and TP. All the values are reported in the $\log_2()$ scale.

when it comes to online run time ($\mathsf{PT}_{\mathsf{on}}$), by 4.4× over ABY3. This is also reflected in the TP, where the improvement is up to 2.8×, as evident from Figure 11.1c. For inference, the communication is in the order of a few megabytes, while run time is in the order of a few seconds. The key observation is that communication is well suited for the bandwidth used (40 MBps). So unlike training, the monetary cost in inference depends more on run time rather than on communication. This is evident from Table 11.2 which shows that $\mathsf{ASTRA}_{\mathsf{T}}$ saves on monetary cost up to a factor of 4 over ABY3. A similar trend is observed in the case of Support Vector Machines.

Algorithm	Parameter[a]	Inference[b]		
		ABY3	$\mathsf{ASTRA}_{\mathsf{T}}$	$\mathsf{ASTRA}_{\mathsf{C}}$
Support Vector Machines	$\mathsf{PT}_{\mathsf{on}}$	12.45	2.53	4.39
	$\mathsf{PT}_{\mathsf{tot}}$	12.45	2.54	4.39
	$\mathsf{CT}_{\mathsf{tot}}$	27.86	5.07	8.78
	$\mathsf{Comm}_{\mathsf{tot}}$	604.93	1161.63	666.46
	Cost	23.71	4.43	7.45
	TP	616.73	1827.61	1055.38

[a]Time (in seconds) and communication (in KB) are reported.
[b]Cost is reported for 1000 queries.

Table 11.3: Benchmarking of the inference phase of Support Vector Machines.

Note that the cost-optimized variant underperforms in terms of monetary cost compared to $\mathsf{ASTRA}_{\mathsf{T}}$. This is because run time plays a more significant role in monetary cost than communication. Hence for inference, the time-optimized variant becomes the optimal choice.

11.3 Additional Benchmarking

11.3.1 Varying batch sizes and feature sizes

Table 11.4 shows the online throughput (TP) of neural network (NN-1) training over varying batch sizes and feature sizes using synthetic datasets.

We find that both $\mathsf{ASTRA}_{\mathsf{T}}, \mathsf{ASTRA}_{\mathsf{C}}$ are up to 2.9× higher in TP. However, as the batch size and feature size increase, ABY3 and ASTRA experience a bandwidth bottleneck.

11.3.2 Comparison operations

Table 11.5 compares the performance of the frameworks for circuits of varying depth. At each layer of the circuits, we perform 128 comparisons where the comparison results are generated

Batch Size	Features	ABY3	ASTRA$_T$	ASTRA$_C$
128	10	1314.59	2997.27	2140.91
	100	1314.59	2997.27	2140.91
	1000	1104.37	2625.67	2139.75
256	10	725.66	2113.79	2058.65
	100	716.15	2060.63	2008.18
	1000	633.13	1646.47	1612.81

Table 11.4: Online throughput (TP) of NN-1 training (iterations per minute) over various batch sizes and features.

in arithmetic shared form. The idea is that each layer emulates a comparison layer in an NN with a batch size of 128.

Depth	Parameter	ABY3	ASTRA$_T$	ASTRA$_C$
128	PT$_{on}$	2.62	0.53	0.93
	CT$_{tot}$	5.87	1.06	1.85
	Cost	0.3	0.05	0.09
1024	PT$_{on}$	20.99	4.23	7.41
	CT$_{tot}$	46.99	8.47	14.82
	Cost	2.38	0.43	0.75
8192	PT$_{on}$	167.93	33.87	59.27
	CT$_{tot}$	375.89	67.74	118.54
	Cost	19.06	3.45	6.02

Table 11.5: Benchmarking of comparisons over various depths. Each of the layer has 128 comparisons. Time is reported in minutes, and monetary cost in USD.

To summarise the experimental results, beyond a depth of roughly 100, the time-optimized variant (ASTRA$_T$) starts outperforming in every metric, especially monetary cost, over the cost-optimized one (ASTRA$_C$). This is because as the depth increases, runtime (CT) grows at a much higher rate than the total communication. What we can infer from Table 11.5 is that if one were to use a DNN with a depth of over 100, ASTRA$_T$ becomes the optimal choice.

Chapter 12

SWIFT: 3PC Fair and Robust Applications

$\mathsf{SWIFT}_\mathsf{T}$ uses multi-input multiplication gates and the 2 GC variant of the garbled world and is the fastest variant of the framework. On the other hand, $\mathsf{SWIFT}_\mathsf{C}$ is the variant with a minimal monetary cost. We report only the numbers for the fair variant of SWIFT and not the robust variant since the overhead of robust over its fair counterpart is very minimal for the algorithms considered in this thesis.

12.1 ML Training

We begin with analyzing the benchmarks for linear and logistic regression. The improvements observed in the three-party semi-honest case carry forward to SWIFT as well. Both $\mathsf{SWIFT}_\mathsf{T}$ and $\mathsf{SWIFT}_\mathsf{C}$ showcase an improvement over ABY3 in terms of communication and run time. This also improves the monetary cost over ABY3, where the saving is up to 70%. One of the primary reasons for the improvement is the reduction in communication. This is attributed to an improved dot product protocol whose communication cost is independent of the vector dimension and a method for truncation which does not incur any overhead in the online phase. Moreover, our multiplication protocol has around 3.5× improvement in terms of communication over ABY3.

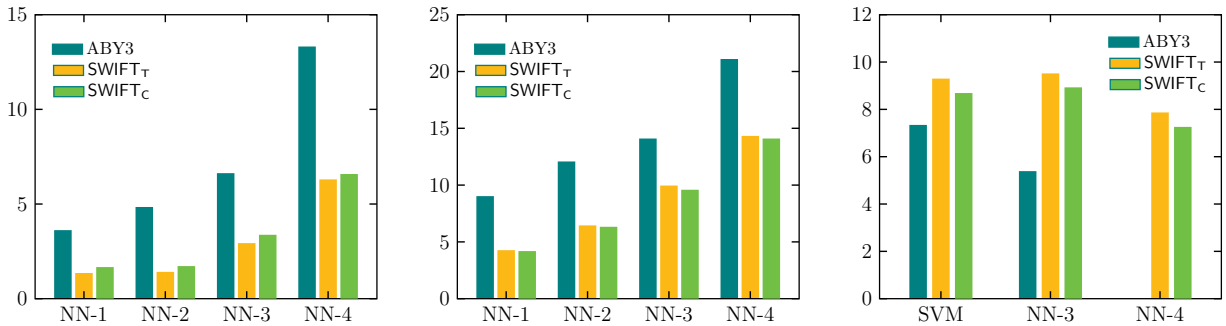
The improvements are more evident in the case of neural networks. Here, $\mathsf{SWIFT}_\mathsf{T}$ is up to two orders of magnitude faster than ABY3 in the online phase. The same trend holds true for communication costs. Like ASTRA, the cost-optimized variant, $\mathsf{SWIFT}_\mathsf{C}$, saves 15% in monetary cost over $\mathsf{SWIFT}_\mathsf{T}$, while incurring the overhead of 1.2× in the online run time.

Algorithm	Parameter[a]	Training[b]			Inference[c]		
		ABY3	SWIFT$_\mathsf{T}$	SWIFT$_\mathsf{C}$	ABY3	SWIFT$_\mathsf{T}$	SWIFT$_\mathsf{C}$
Linear Regression	PT$_{\mathsf{on}}$	1.09	0.94	0.94	0.97	0.88	0.88
	PT$_{\mathsf{tot}}$	1.16	1.61	1.61	1.43	1.54	1.54
	CT$_{\mathsf{tot}}$	1.69	3.57	3.57	0.66	3.41	3.41
	Comm$_{\mathsf{tot}}$	33225.81	97.91	97.91	128.94	0.25	0.25
	Cost	6.5	3.03	3.03	0.58	2.88	2.88
	TP	521.32	6111.54	6111.54	17497.49	7404.84	7404.84
Logistic Regression	PT$_{\mathsf{on}}$	2.64	1.25	1.44	2.41	1.18	1.36
	PT$_{\mathsf{tot}}$	2.76	1.92	2.11	2.49	1.84	2.02
	CT$_{\mathsf{tot}}$	8.28	4.2	4.57	7.24	4.01	4.38
	Comm$_{\mathsf{tot}}$	33494.5	204.13	154.53	131.04	1.08	0.69
	Cost	12.1	3.58	3.88	6.14	3.39	3.69
	TP	517.39	3262.62	2549.53	1590.75	3598.18	2749.97

[a]Time (in seconds) and communication (in KB) are reported. [b]For training, batch size is 128 and the monetary cost (USD) is reported for 1000 iterations. [c]For inference, cost is reported for 1000 queries.

Table 12.1: Benchmarking of Linear Regression and Logistic Regression algorithms.

These trends can be better captured with a pictorial representation as given in Figure 12.1.



(a) Training: PT$_{\mathsf{on}}$ (b) Training: Cost (c) Inference: TP

Figure 12.1: Analysis of protocols in terms of PT$_{\mathsf{on}}$, Cost and TP. All the values are reported in the $\log_2()$ scale.

12.2 ML Inference

For linear regression, logistic regression and support vector machines, we observe a similar trend as in the inference of ASTRA, where SWIFT$_\mathsf{T}$ outperforms ABY3 and SWIFT$_\mathsf{C}$ in terms of run time and monetary cost. For neural networks, the time-optimized variant SWIFT$_\mathsf{T}$ is faster when it comes to online run time (PT$_{\mathsf{on}}$), by 9.7× over ABY3. This is also reflected in the

Algorithm	Parameter[a]	Training[b]			Inference[c]		
		ABY3	SWIFT$_\mathsf{T}$	SWIFT$_\mathsf{C}$	ABY3	SWIFT$_\mathsf{T}$	SWIFT$_\mathsf{C}$
NN-1	PT$_{\text{on}}$	12.31	2.56	3.18	7.24	1.93	2.49
	PT$_{\text{tot}}$	28.89	5.91	6.53	7.31	2.6	3.16
	CT$_{\text{tot}}$	80.89	14.82	16.06	21.92	5.54	6.66
	Comm$_{\text{tot}}$	2.98	0.31	0.17	11.85	0.14	0.09
	Cost	522.74	19.31	18.3	21.44	4.71	5.64
	TP	5.86	1102.34	838.28	530.13	1610.74	1138.94
NN-2	PT$_{\text{on}}$	28.74	2.67	3.3	7.27	1.93	2.49
	PT$_{\text{tot}}$	146.24	20.92	21.55	7.82	2.62	3.17
	CT$_{\text{tot}}$	432.93	59.5	60.76	23.47	5.6	6.71
	Comm$_{\text{tot}}$	25.02	0.49	0.32	178.09	0.55	0.35
	Cost	4341.79	87.52	80.22	47.63	4.85	5.73
	TP	0.67	304.35	300.84	94.52	1610.74	1138.94
NN-3	PT$_{\text{on}}$	99.1	7.65	10.38	18.51	3.55	5.03
	PT$_{\text{tot}}$	643.69	174.87	177.6	20.08	4.65	6.14
	CT$_{\text{tot}}$	1925.28	514.31	519.77	60.25	10.07	13.05
	Comm$_{\text{tot}}$	100.52	2.9	2.05	398.96	12.34	7.37
	Cost	17607.31	996.09	771.68	112.93	11.41	12.36
	TP	0.17	23.81	23.23	42.71	733.25	487.9
NN-4	PT$_{\text{on}}$	10222	79.22	96.03	96.52	9.9	14.75
	PT$_{\text{tot}}$	58428.87	4195.38	4212.19	264.58	17.66	22.5
	CT$_{\text{tot}}$	175280.84	12456.7	12490.33	793.73	42.65	52.34
	Comm$_{\text{tot}}$	13225.44	51.67	41.82	54335.94	184.21	112.44
	Cost	2262986.61	20697.42	17586.31	9156.18	78.92	64.44
	TP	0	1.18	1.16	0.31	233.45	153.53

[a]Time is reported in seconds [b]For training, communication is reported in GB. Monetary cost (USD) is reported for 1000 iterations and batch size is 128. [c]For inference, communication is reported in MB and the cost is reported for 1000 queries.

Table 12.2: Benchmarking of Neural Networks.

TP, where the improvement is up to 753×, as evident from Table 12.2. Unlike the inference of ASTRA, the cost-optimized variant SWIFT$_\mathsf{C}$ outperforms the rest (ABY3 and SWIFT$_\mathsf{T}$) in terms of the monetary cost and communication as the network becomes deeper. The trends in throughput are captured in Figure 12.1c.

12.3 Additional Benchmarking

12.3.1 Varying batch sizes and feature sizes

Table 12.4 shows the online throughput (TP) of neural network (NN-1) training over varying batch sizes and feature sizes using synthetic datasets.

Algorithm	Parameter[a]	Inference[b]		
		ABY3	SWIFT$_T$	SWIFT$_C$
Support Vector Machines	PT$_{on}$	22.17	3.97	5.83
	PT$_{tot}$	22.18	4.71	6.57
	CT$_{tot}$	66.54	9.85	13.56
	Comm$_{tot}$	9497.64	3339.88	1822.93
	Cost	57.56	9.12	11.78
	TP	173	639.46	413.04

[a]Time (in seconds) and communication (in KB) are reported.
[b]Cost is reported for 1000 queries.

Table 12.3: Benchmarking of the inference phase of Support Vector Machines.

Batch Size	Features	ABY3	SWIFT$_T$	SWIFT$_C$
128	10	20.78	1102.82	838.56
	100	16.04	1102.82	838.56
	1000	4.88	1102.09	838.14
256	10	10.41	1102.56	838.41
	100	8.05	1102.56	838.41
	1000	2.46	981.09	836.42

Table 12.4: Online throughput (TP) of NN-1 training (iterations per minute) over various batch sizes and features.

12.3.2 Comparison operations

Table 12.5 compares the performance of the frameworks for circuits of varying depth. At each layer of the circuits, we perform 128 comparisons where the comparison results are generated in arithmetic shared form. The idea is that each layer emulates a comparison layer in an NN with a batch size of 128.

Depth	Parameter	ABY3	SWIFT$_T$	SWIFT$_C$
128	PT$_{on}$	4.21	0.66	1.06
	CT$_{tot}$	12.64	1.33	2.12
	Cost	0.64	0.07	0.11
1024	PT$_{on}$	33.71	5.29	8.47
	CT$_{tot}$	101.13	10.63	16.98
	Cost	5.14	0.55	0.87
8192	PT$_{on}$	269.67	42.33	67.73
	CT$_{tot}$	809.07	85.04	135.84
	Cost	41.12	4.41	6.92

Table 12.5: Benchmarking of comparisons over various depths. Each of the layer has 128 comparisons. Time is reported in minutes, and monetary cost in USD.

To summarize, SWIFT improves over ABY3 up to two orders of magnitude in terms of monetary cost. As observed from the Table 12.2, $\mathsf{SWIFT}_\mathsf{T}$ provides the best online time while $\mathsf{SWIFT}_\mathsf{C}$ attains the best monetary cost, corroborating our claims.

Chapter 13

Tetrad: 4PC Fair and Robust Applications

$\mathsf{Tetrad}_\mathsf{T}$ uses multi-input multiplication gates and the 2 GC variant of the garbled world and is the fastest variant of the framework. On the other hand, $\mathsf{Tetrad}_\mathsf{C}$ is the variant with a minimal monetary cost. We report only the numbers for the fair variant of Tetrad and not the robust variant since the overhead of robust over its fair counterpart is very minimal for the algorithms considered in this thesis.

For training, we benchmark against the fair 4PC framework of Trident [38]. For inference, in addition to Trident, we also benchmark against the 4PC robust protocol of SWIFT [85] since it supports NN inference. Note that the best case performance of Fantastic Four [46], when cast in the preprocessing model, resembles that of SWIFT. In contrast, their worst-case execution (3PC malicious) is an order of magnitude slower (cf. §5.2.6.1), as demonstrated in their paper (cf. Table 2 of [46]).

13.1 ML Training

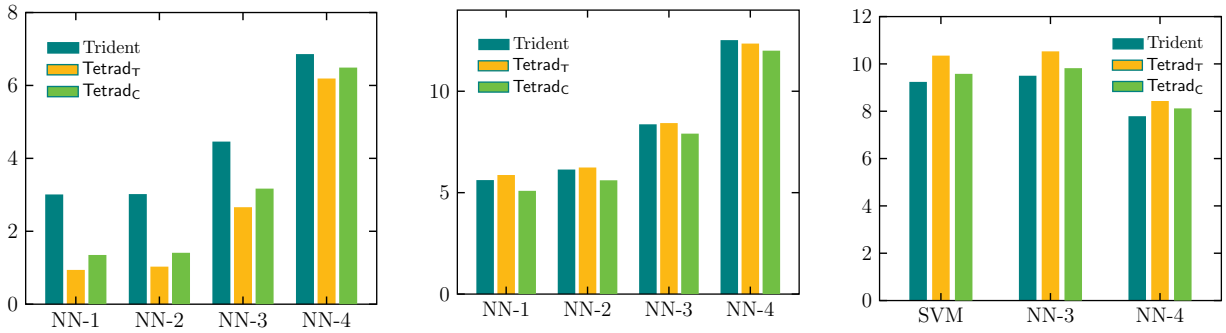
Starting with the time-optimized variant, $\mathsf{Tetrad}_\mathsf{T}$ is $3-4\times$ faster than Trident in online runtime. The primary factor is the reduction in online rounds of our protocol due to multi-input gates. More precisely, we use the depth-optimized bit extraction circuit while instantiating the ReLU activation function using multi-input AND gates (cf. §9.1.2). Looking at the total communication ($\mathsf{Comm}_{\mathsf{tot}}$) in Table 13.2, we observe that the gap in $\mathsf{Comm}_{\mathsf{tot}}$ between $\mathsf{Tetrad}_\mathsf{T}$ vs. Trident decreases as the networks get deeper. This is justified as the improvement in communication of our dot product with truncation outpaces the overhead in communication caused

by multi-input gates. The impact of this is more pronounced with NN-4, as observed by the lower monetary cost of $\mathsf{Tetrad_T}$ over Trident. Another reason is that there are two active parties (P_1, P_2) in our framework, whereas Trident has three. Given the allocation of servers, the best rtt Trident can get with three parties (P_0, P_1, P_2) is $153.74ms$, compared to $62.01ms$ of Tetrad, contributing to Tetrad being faster. However, if the rtt among all the parties were similar, this gap would be closed. Concretely, the online runtime ($\mathsf{PT_{on}}$) of Trident will be similar to that of $\mathsf{Tetrad_C}$.

Algorithm	Parameter[a]	Training[b]			Inference[c]			
		Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$	Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$	SWIFT
Linear Regression	$\mathsf{PT_{on}}$	0.83	0.5	0.5	0.44	0.44	0.44	0.99
	$\mathsf{PT_{tot}}$	1.11	0.78	0.78	0.71	0.71	0.71	1.81
	$\mathsf{CT_{tot}}$	2.99	2.15	2.15	2.02	2.02	2.02	5.8
	$\mathsf{Comm_{tot}}$	76.5	48.03	48.03	0.2	0.2	0.06	0.21
	Cost	2.53	1.83	1.83	1.71	1.71	1.71	4.89
	TP	13971.76	14780.03	14780.03	27944.03	20094.71	20094.71	11688.96
Logistic Regression	$\mathsf{PT_{on}}$	2.5	0.75	0.94	2.1	0.68	0.86	1.3
	$\mathsf{PT_{tot}}$	2.77	1.03	1.21	2.38	0.95	1.13	2.12
	$\mathsf{CT_{tot}}$	7.49	2.65	3.02	6.52	2.5	2.86	6.64
	$\mathsf{Comm_{tot}}$	119.16	123.25	86.75	0.53	0.78	0.5	0.54
	Cost	6.34	2.26	2.56	5.5	2.12	2.42	5.61
	TP	4299	7182.26	5183.72	5080.81	8241.66	5713.88	4743.86

[a]Time (in seconds) and communication (in KB) are reported. [b]For training, batch size is 128 and the monetary cost (USD) is reported for 1000 iterations. [c]For inference, cost is reported for 1000 queries.

Table 13.1: Benchmarking of Linear Regression and Logistic Regression algorithms.



(a) Training: $\mathsf{PT_{on}}$ (b) Training: Cost (c) Inference: TP

Figure 13.1: Analysis of protocols in terms of $\mathsf{PT_{on}}$, Cost and TP. All the values are reported in the $\log_2()$ scale.

On the other hand, the cost-optimized variant $\mathsf{Tetrad_C}$ is 1.5× slower in the online phase than $\mathsf{Tetrad_T}$. However, it is still faster than Trident owing to the rtt setup, as discussed above.

When it comes to monetary cost, this variant is up to $20-40\%$ cheaper than its time-optimized counterpart and cheaper by around 30% over Trident.

These trends can be better captured with a pictorial representation as given in Figure 13.1.

Algorithm	Parameter[a]	Training[b]			Inference[c]			
		Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$	Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$	SWIFT
NN-1	$\mathsf{PT_{on}}$	8.06	1.93	2.55	5.87	1.31	1.87	2.31
	$\mathsf{PT_{tot}}$	10.76	5.05	5.27	6.15	1.58	2.14	3.13
	$\mathsf{CT_{tot}}$	27.9	12.69	11.22	16.75	3.76	4.88	8.65
	$\mathsf{Comm_{tot}}$	0.16	0.3	0.16	0.06	0.09	0.05	0.06
	Cost	49.33	58.51	34.29	14.15	3.19	4.13	7.32
	TP	118.75	2083.68	1517.79	1802.8	3330.33	2167.73	2011.68
NN-2	$\mathsf{PT_{on}}$	8.13	2.05	2.67	5.87	1.31	1.87	2.31
	$\mathsf{PT_{tot}}$	11.47	5.79	6.14	6.15	1.58	2.14	3.13
	$\mathsf{CT_{tot}}$	30.88	14.86	13.4	16.75	3.77	4.88	8.66
	$\mathsf{Comm_{tot}}$	0.28	0.39	0.24	0.26	0.37	0.22	0.25
	Cost	70.84	75.67	49.16	14.19	3.24	4.16	7.35
	TP	428.16	652.75	644.69	1802.8	3330.32	2167.73	2011.68
NN-3	$\mathsf{PT_{on}}$	22.04	6.33	9.06	14.42	2.61	4.1	4.54
	$\mathsf{PT_{tot}}$	30.91	15.79	18.53	14.71	2.91	4.39	5.39
	$\mathsf{CT_{tot}}$	92.37	41.7	44.45	39.92	6.43	9.4	13.18
	$\mathsf{Comm_{tot}}$	1.59	1.94	1.28	5.62	8.42	4.76	5.39
	Cost	331.76	345.16	241.83	34.59	6.74	8.68	11.97
	TP	53.62	55.71	54.13	725.8	1479.22	904.6	876.23
NN-4	$\mathsf{PT_{on}}$	116.32	73.19	90.01	47.05	7.85	12.69	13.13
	$\mathsf{PT_{tot}}$	328.2	229.42	246.23	47.61	8.44	13.28	14.33
	$\mathsf{CT_{tot}}$	983.74	16866.48	643.06	129.41	17.77	27.46	31.35
	$\mathsf{Comm_{tot}}$	31.59	29.52	22.24	85.69	124.09	71.27	81.33
	Cost	5884.81	5240.81	4101.26	122.66	34.32	34.4	39.18
	TP	2.54	2.61	2.56	222.54	458.25	279.44	276.67

[a]Time is reported in seconds [b]For training, communication is reported in GB. Monetary cost (USD) is reported for 1000 iterations and batch size is 128. [c]For inference, communication is reported in MB and the cost is reported for 1000 queries.

Table 13.2: Benchmarking of Neural Networks.

13.2 ML Inference

Similar to training, the time-optimized variant for inference is faster when it comes to $\mathsf{PT_{on}}$, by $4-6\times$ over Trident. This is also reflected in the TP, where the improvement is about $2.8-5.5\times$, as evident from Figure 13.1c. In inference, the communication is in the order of megabytes,

while run time is in the order of a few seconds. The key observation is that communication is well suited for the bandwidth used (40 MBps). So unlike training, the monetary cost in inference depends more on run time rather than on communication. This is evident from Table 13.5 which shows that $\mathsf{Tetrad_T}$ saves on monetary cost up to a factor of 10 over Trident.

Algorithm	Parameter[a]	Inference[b]			
		Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$	SWIFT
Support Vector Machines	$\mathsf{PT_{on}}$	17.09	2.91	4.77	5.21
	$\mathsf{PT_{tot}}$	17.37	3.19	5.05	6.04
	$\mathsf{CT_{tot}}$	47.02	6.99	10.7	14.47
	$\mathsf{Comm_{tot}}$	1395.72	2391.47	1275.01	1395.59
	Cost	45.92	6.26	9.23	12.43
	TP	607.47	1306.34	767.87	747.34

[a]Time (in seconds) and communication (in KB) are reported. [b]Cost is reported for 1000 queries.

Table 13.3: Benchmarking of the inference phase of Support Vector Machines.

Note that the cost-optimized variant underperforms in terms of monetary cost compared to $\mathsf{Tetrad_T}$. This is because, as mentioned earlier, run time plays a more significant role in monetary cost than communication. Hence for inference, the time-optimized variant becomes the optimal choice.

13.3 Additional Benchmarking

13.3.1 Varying batch sizes and feature sizes

Table 13.4 shows the online throughput (TP) of neural network (NN-1) training over varying batch sizes and feature sizes using synthetic datasets.

Batch Size	Features	Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$
128	10	1189.08	2086.28	1519.17
	100	1189.08	2086.28	1519.17
	1000	1188.75	2083.68	1517.79
256	10	1189.08	2084.19	1518.06
	100	1189.08	2084.19	1518.06
	1000	1188.75	2077.69	1514.62

Table 13.4: Online throughput (TP) of NN-1 training (iterations per minute) over various batch sizes and features.

We find that both $\mathsf{Tetrad_T}$, $\mathsf{Tetrad_C}$ are up to 1.8× higher in TP. However, as the batch size and feature size increase, Trident and Tetrad experience a bandwidth bottleneck. The effect of the bandwidth limitation is higher for Tetrad; hence the gain in TP over Trident decreases a bit.

13.3.2 Comparison operations

Table 13.5 compares the performance of the frameworks for circuits of varying depth. At each layer of the circuits, we perform 128 comparisons where the comparison results are generated in arithmetic shared form. The idea is that each layer emulates a comparison layer in an NN with a batch size of 128.

Depth	Parameter	Trident	$\mathsf{Tetrad_T}$	$\mathsf{Tetrad_C}$
128	$\mathsf{PT_{on}}$	3.55	0.53	0.93
	$\mathsf{CT_{tot}}$	9.6	1.06	1.85
	Cost	0.49	0.05	0.09
1024	$\mathsf{PT_{on}}$	28.42	4.23	7.41
	$\mathsf{CT_{tot}}$	76.79	8.47	14.82
	Cost	3.89	0.43	0.75
8192	$\mathsf{PT_{on}}$	227.34	33.87	59.27
	$\mathsf{CT_{tot}}$	614.3	67.76	118.56
	Cost	31.15	3.48	6.03

Table 13.5: Benchmarking of comparisons over various depths. Each of the layer has 128 comparisons. Time is reported in minutes, and monetary cost in USD.

Interestingly, beyond a depth of roughly 100, the time-optimized variant ($\mathsf{Tetrad_T}$) starts outperforming in every metric, especially monetary cost, over the cost-optimized one ($\mathsf{Tetrad_C}$). This is because as the depth increases, runtime (CT) grows at a much higher rate than the total communication. What we can infer from Table 13.5 is that if one were to use a DNN with a depth of over 100, $\mathsf{Tetrad_T}$ becomes the optimal choice.

Chapter 14

ABY2.0: 2PC Semi-honest Applications

ABY2.0$_\mathsf{T}$ makes use of multi-input multiplication gates and is the fastest variant of the framework. On the other hand, ABY2.0$_\mathsf{C}$ is the variant with a minimal monetary cost. We benchmark our protocols against the 2PC semi-honest framework of SecureML [102]. The preprocessing phase of ABY2.0 is similar to SecureML except for the use of multi-input multiplication in ABY2.0$_\mathsf{T}$. The preprocessing can be performed either using oblivious transfer or via homomorphic encryption as discussed in Chapter 6. Note that the benchmarking is performed only for the online phase.

14.1 ML Training

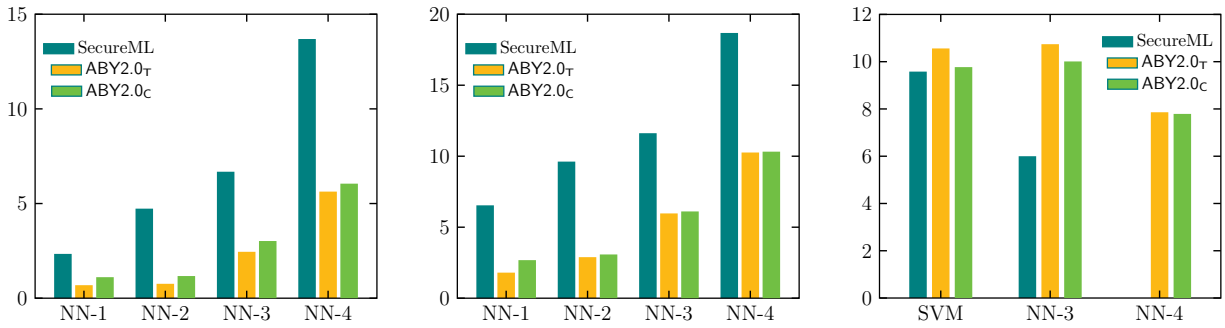
Starting with the time-optimized variant, ABY2.0$_\mathsf{T}$ is up to two orders of magnitude faster than SecureML [102] in run time as well as communication. The reduction is primarily due to the following: (i) the improved dot product protocol whose online phase communication is independent of the dimension of the vector, and (ii) improvements in online rounds due to multi-input multiplication. These reductions in communication and run time directly impact the monetary cost of the system, where ABY2.0$_\mathsf{T}$ brings in a saving of up to 342× over SecureML. On the other hand, the cost-optimized variant ABY2.0$_\mathsf{C}$ is around 1.3× slower than ABY2.0$_\mathsf{T}$. However, it is still faster than SecureML due to the reasons discussed above. Further, this variant has a slightly higher communication than ABY2.0$_\mathsf{T}$ due to the absence of multi-input multiplication.

These trends can be better captured with a pictorial representation as given in Figure 14.1.

Algorithm	Parameter[a]	Training[b]			Inference[c]		
		SecureML	ABY2.0$_T$	ABY2.0$_C$	SecureML	ABY2.0$_T$	ABY2.0$_C$
Linear Regression	PT$_{on}$	0.14	0.12	0.12	0.06	0.06	0.06
	CT$_{on}$	0.28	0.25	0.25	0.12	0.12	0.12
	Comm$_{on}$	6272	14.25	14.25	24.5	0.02	0.02
	Cost	1.2	0.21	0.21	0.11	0.1	0.1
	TP	783.67	30962.74	30962.74	61925.5	63872.25	63872.25
Logistic Regression	PT$_{on}$	0.64	0.37	0.56	0.55	0.3	0.48
	CT$_{on}$	1.28	0.74	1.12	1.11	0.6	0.96
	Comm$_{on}$	6295.75	23.44	24.13	24.68	0.09	0.09
	Cost	2.04	0.63	0.95	0.94	0.51	0.81
	TP	779.73	10320.91	6880.61	6927.53	12774.45	7984.05

[a]Time (in seconds) and communication (in KB) are reported. [b]For training, batch size is 128 and the monetary cost (USD) is reported for 1000 iterations. [c]For inference, cost is reported for 1000 queries.

Table 14.1: Benchmarking of Linear Regression and Logistic Regression algorithms.



(a) Training: PT$_{on}$ (b) Training: Cost (c) Inference: TP

Figure 14.1: Analysis of protocols in terms of PT$_{on}$, Cost and TP. All the values are reported in the $\log_2()$ scale.

14.2 ML Inference

Like training, the time-optimized variant for inference is faster when it comes to the performance in the online phase. For shallow NNs such as NN-1, we observe a 4× improvement in the online throughput over SecureML. However, the improvement changes drastically as the network grows bigger. Specifically, we observe a gain in throughput of up to 500× over SecureML for the case of NN-4. The poor performance of SecureML is due to the huge increase in communication costs for deeper networks, which forms the bottleneck.

Algorithm	Parameter[a]	Training[b]			Inference[c]		
		SecureML	ABY2.0$_T$	ABY2.0$_C$	SecureML	ABY2.0$_T$	ABY2.0$_C$
NN-1	PT_{on}	5.07	1.61	2.17	1.68	0.93	1.49
	CT_{on}	10.14	3.23	4.34	3.37	1.86	2.98
	$Comm_{on}$	540.22	4.94	5.03	3.63	0.02	0.02
	Cost	93.62	3.5	6.45	3.41	1.57	2.52
	TP	8.81	947.79	931.37	1315.06	4128.37	2580.22
NN-2	PT_{on}	26.7	1.7	2.26	1.78	0.93	1.49
	CT_{on}	53.41	3.4	4.53	3.57	1.86	2.98
	$Comm_{on}$	4752.91	29.29	29.66	33.78	0.06	0.07
	Cost	790.4	7.45	8.46	8.3	1.58	2.52
	TP	1.01	163.17	161.17	141.82	4128.37	2580.22
NN-3	PT_{on}	103.15	5.48	8.15	4.46	2.23	3.72
	CT_{on}	206.3	10.97	16.31	8.92	4.46	7.44
	$Comm_{on}$	18654	344.54	35.26	73.16	1.35	1.42
	Cost	3157.52	63.1	69.18	19.21	3.98	6.51
	TP	0.25	13.93	13.53	64.2	1720.15	1032.09
NN-4	PT_{on}	13254.9	49.79	66.54	64.76	7.45	12.29
	CT_{on}	26509.79	99.57	133.08	129.51	14.9	24.59
	$Comm_{on}$	2556821.6	7364.1	7501.93	10304.95	20.55	21.54
	Cost	422846.24	1234.73	1284.55	1723.13	15.79	24.13
	TP	0	0.65	0.64	0.46	233.58	222.79

[a]Time is reported in seconds and communication is reported in MB [b]For training, monetary cost (USD) is reported for 1000 iterations and batch size is 128. [c]For inference, the cost is reported for 1000 queries.

Table 14.2: Benchmarking of Neural Networks.

Algorithm	Parameter[a]	Inference[b]		
		SecureML	ABY2.0$_T$	ABY2.0$_C$
Support Vector Machines	PT_{on}	5.01	2.53	4.39
	CT_{on}	10.02	5.07	8.78
	$Comm_{on}$	1213.62	341.46	362.44
	Cost	8.72	4.33	7.47
	TP	766.82	1514.88	874.82

[a]Time (in seconds) and communication (in KB) are reported. [b]Cost is reported for 1000 queries.

Table 14.3: Benchmarking of the inference phase of Support Vector Machines.

14.3 Additional Benchmarking

14.3.1 Varying batch sizes and feature sizes

Table 14.4 shows the online throughput (TP) of neural network (NN-1) training over varying batch sizes and feature sizes using synthetic datasets.

Batch Size	Features	SecureML	ABY2.0$_T$	ABY2.0$_C$
128	10	31.02	1351.09	1317.96
	100	23.99	1287.39	1257.28
	1000	7.34	874.91	860.89
256	10	15.54	704.19	686.21
	100	12.02	686.49	669.39
	1000	3.68	548.57	537.6

Table 14.4: Online throughput (TP) of NN-1 training (iterations per minute) over various batch sizes and features.

14.3.2 Comparison operations

Table 14.5 compares the performance of the frameworks for circuits of varying depth. At each layer of the circuits, we perform 128 comparisons where the comparison results are generated in arithmetic shared form. The idea is that each layer emulates a comparison layer in an NN with a batch size of 128.

Depth	Parameter	SecureML	ABY2.0$_T$	ABY2.0$_C$
128	PT$_{on}$	0.93	0.53	0.93
	CT$_{on}$	1.85	1.06	1.85
	Cost	0.09	0.05	0.09
1024	PT$_{on}$	7.41	4.23	7.41
	CT$_{on}$	14.82	8.47	14.82
	Cost	0.75	0.43	0.75
8192	PT$_{on}$	59.27	33.87	59.27
	CT$_{on}$	118.53	67.73	118.53
	Cost	6.03	3.44	6.01

Table 14.5: Benchmarking of comparisons over various depths. Each of the layer has 128 comparisons. Time is reported in minutes, and monetary cost in USD.

Having benchmarked only the online phase, ABY2.0$_T$ is clearly the winner with respect to all the metrics. We believe a similar trend as observed in the prior frameworks will be followed here as well when considering the overall performance.

Chapter 15

Conclusion and Open Problems

This thesis designed **MPCLeague**, a robust MPC platform for privacy-preserving machine learning applications. The focus was on the small-party setting of two, three and four parties with at most one corruption under the control of a monolithic static adversary. A unified protocol design was presented, focusing on practical efficiency, which outperforms the state-of-the-art protocols by several orders of magnitude in the respective settings. On the way, several building blocks were identified for the PPML applications, and their efficient realizations were provided. Finally, the protocols were implemented by instantiating over Google Cloud instances and analyzed against various metrics such as run time, communication, throughput and monetary cost. The practicality of our platform was argued through improvements as observed in the benchmarks.

Open Problems We leave the following problems open for further explorations.

1. *Applications:* The platform was designed for PPML applications such as linear regression, logistic regression, neural networks and support vector machines. However, other PPML applications such as graph neural networks, decision trees and random forests, quantized neural networks have not been explored much in the literature. Extending our platform to provide support for these advanced applications is an interesting direction. This may require support for new building blocks in layer II, such as square-root, exponentiation, batch normalization, to name a few. While the platform discussed PPML applications, it is worthwhile to explore non-PPML applications such as private-set intersection, private-information retrieval, genome sequence matching.

2. *Adversarial setting:* The focus of the thesis was primarily on the honest majority setting. A step towards a dishonest majority was also taken, albeit in the semi-honest two-party

setting. It is an interesting question to explore the dishonest majority setting in the presence of a malicious adversary. While protocols were designed in the synchronous network model with static corruptions, designing protocols in the asynchronous network model and against a stronger adaptive adversary is left open.

The recent notion of Friends-and-Foes [4] (FaF) security resembles real-world corruption more closely, where the honest parties are instead considered to be semi-honest. Our protocols do not adhere to this security notion, and designing protocols for the same is an interesting future direction. Finally, our protocols, together with the above-mentioned adversarial settings, can be explored for the general n-party case.

3. *Federated Learning:* The advancements in PPML have paved the way for federated learning which allows collaborative training while ensuring the training data resides only with the data owners. Since the data does not leave its owner, it increases the trust in the system and has gained a lot of attention recently. The traditional approach of realizing PPML via MPC does not extend naively to the federated setting. We leave open the question of realizing our architecture in the federated setting as an open problem.

Bibliography

[1] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part I*, volume 11891 of *Lecture Notes in Computer Science*, pages 471–501, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-36030-6_19. 71

[2] Mark Abspoel, Anders 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, 49

[3] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 2018. URL https://doi.org/10.1145/3214303. 92

[4] Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. MPC with friends and foes. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 677–706, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-56880-1_24. 175

[5] Javier Alvarez-Valle, Pratik Bhatu, Nishanth Chandran, Divya Gupta, Aditya V. Nori, Aseem Rastogi, Mayank Rathee, Rahul Sharma, and Shubham Ugare. Secure medical image analysis with cryptflow. *CoRR*, abs/2012.05064, 2020. URL https://arxiv.org/abs/2012.05064. 1

[6] Yoshinori Aono, Takuya Hayashi, Le Trieu Phong, and Lihua Wang. Scalable and secure logistic regression via homomorphic encryption. Cryptology ePrint Archive, Report 2016/111, 2016. https://eprint.iacr.org/2016/111. 6

[7] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 805–817, Vienna, Austria, October 24–28, 2016. ACM Press. doi: 10.1145/2976749.2978331. 2, 5, 7, 153

[8] 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 *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press. doi: 10.1109/SP.2017.15. 2, 7, 153

[9] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 535–548, Berlin, Germany, November 4–8, 2013. ACM Press. doi: 10.1145/2508859.2516738. 8, 91, 92, 94, 95

[10] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany. doi: 10.1007/3-540-46766-1_34. xiv, 3, 7, 10, 14, 15

[11] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *Advances in Cryptology – CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109, Santa Barbara, CA, USA, August 27–31, 1995. Springer, Heidelberg, Germany. doi: 10.1007/3-540-44750-4_8. 92

[12] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, MD, USA, May 14–16, 1990. ACM Press. doi: 10.1145/100216.100287. 2, 22

[13] Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages

305–328, New York, NY, USA, March 4–7, 2006. Springer, Heidelberg, Germany. doi: 10.1007/11681878_16. 3

[14] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230, San Francisco, CA, USA, March 19–21, 2008. Springer, Heidelberg, Germany. doi: 10.1007/978-3-540-78524-8_13. 3

[15] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 784–796, Raleigh, NC, USA, October 16–18, 2012. ACM Press. doi: 10.1145/2382196.2382279. 22, 57

[16] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press. doi: 10.1109/SP.2013.39. 28

[17] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*, volume 11464 of *Lecture Notes in Computer Science*, pages 530–549, Bogota, Colombia, June 5–7, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-21568-2_26. x, xvii, 8, 26, 94, 98

[18] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press. doi: 10.1145/62212.62213. 2

[19] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany. doi: 10.1007/978-3-642-32009-5_39. 3

[20] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008:*

13th European Symposium on Research in Computer Security, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany. doi: 10.1007/978-3-540-88313-5_13. 4, 5

[21] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 2012. 104

[22] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In Angelos D. Keromytis, editor, *FC 2012: 16th International Conference on Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64, Kralendijk, Bonaire, February 27 – March 2, 2012. Springer, Heidelberg, Germany. 5

[23] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009: 13th International Conference on Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343, Accra Beach, Barbados, February 23–26, 2009. Springer, Heidelberg, Germany. 5

[24] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 67–97, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-26954-8_3. 7, 49, 117, 118, 119, 122

[25] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-96878-0_17. 6

[26] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure com-

putation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 291–308. ACM Press, November 11–15, 2019. doi: 10.1145/3319535.3354255. 95

[27] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 869–886. ACM Press, November 11–15, 2019. doi: 10.1145/3319535.3363227. 7, 49, 50, 54, 71, 117, 118, 119, 121, 122

[28] Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007: 14th Conference on Computer and Communications Security*, pages 486–497, Alexandria, Virginia, USA, October 28–31, 2007. ACM Press. doi: 10.1145/1315245.1315306. 5

[29] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 847–861, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi: 10.1145/3243734.3243786. 8

[30] Megha Byali, Arun Joseph, Arpita Patra, and Divya Ravi. Fast secure computation for small population over the internet. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 677–694, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi: 10.1145/3243734.3243784. 2, 5, 7, 81

[31] Megha Byali, Carmit Hazay, Arpita Patra, and Swati Singla. Fast actively secure five-party computation with security beyond abort. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1573–1590. ACM Press, November 11–15, 2019. doi: 10.1145/3319535.3345657. 2

[32] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. *Proceedings on Privacy Enhancing*

Technologies, 2020(2):459–480, April 2020. doi: 10.2478/popets-2020-0036. vii, 2, 5, 6, 7, 16, 20, 27

[33] José Cabrero-Holgueras and Sergio Pastrana. Sok: Privacy-preserving computation techniques for deep learning. *Proc. Priv. Enhancing Technol.*, 2021(4):139–162, 2021. 6

[34] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000. doi: 10.1007/s001459910006. 18

[35] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint Archive, Report 2017/035, 2017. https://eprint.iacr.org/2017/035. 6

[36] Nishanth Chandran, Juan A. Garay, Payman Mohassel, and Satyanarayana Vusirikala. Efficient, constant-round and actively secure MPC: Beyond the three-party case. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 277–294, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. doi: 10.1145/3133956.3134100. 5

[37] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK*, 2019. https://eprint.iacr.org/2019/429. vi, 2, 5, 6, 7, 9, 11, 16, 20, 29, 149

[38] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *ISOC Network and Distributed System Security Symposium – NDSS 2020*, San Diego, CA, USA, February 23-26, 2020. The Internet Society. vi, xviii, 2, 3, 5, 6, 7, 8, 9, 11, 16, 20, 27, 61, 62, 108, 137, 149, 151, 153, 165

[39] A. Choudhury and A. Patra. An Efficient Framework for Unconditionally Secure Multiparty Computation. *IEEE Trans. Information Theory*, 2017. 3

[40] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th Annual ACM Symposium on Theory of Computing*, pages 364–369, Berkeley, CA, USA, May 28–30, 1986. ACM Press. doi: 10.1145/12130.12168. 9

[41] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 466–485, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany. doi: 10.1007/978-3-662-45608-8_25. 18

[42] Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 596–613, Warsaw, Poland, May 4–8, 2003. Springer, Heidelberg, Germany. doi: 10.1007/3-540-39200-9_37. 4

[43] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany. doi: 10.1007/978-3-540-30576-7_19. ix, 21

[44] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-96881-0_26. 4, 7

[45] Cryptography and Privacy Engineering Group at TU Darmstadt. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils, 2017. 153

[46] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. Cryptology ePrint Archive, Report 2020/1330, 2020. https://eprint.iacr.org/2020/1330. xvii, 6, 7, 11, 61, 80, 165

[47] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Homomorphic encryption and secure comparison. *Int. J. Appl. Cryptogr.*, 2008. URL https://doi.org/10.1504/IJACT.2008.017048. 92

[48] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany. doi: 10.1007/978-3-642-32009-5_38. 3, 7, 94

[49] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 799–829, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-96881-0_27. 3, 4

[50] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press. doi: 10.1109/SP.2019.00078. 4, 106, 149, 151

[51] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *ISOC Network and Distributed System Security Symposium – NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society. xvii, xviii, 3, 4, 5, 8, 94, 95, 98, 145

[52] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society. xvii, 8, 97, 104

[53] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative scientific computations. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 273–294. IEEE Computer Society, 2001. 5

[54] Wenliang Du, Yunghsiang S. Han, and Shigang Chen. Privacy-preserving multivariate statistical analysis: Linear regression and classification. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 222–233. SIAM, 2004. 5

[55] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852, Santa Bar-

bara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-56880-1_29. 3, 4, 8, 139

[56] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 1322–1333, Denver, CO, USA, October 12–16, 2015. ACM Press. doi: 10.1145/2810103.2813677. 16

[57] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-56614-6_8. 2, 7

[58] Adria Gascon, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Secure linear regression on vertically partitioned datasets. Cryptology ePrint Archive, Report 2016/892, 2016. https://eprint.iacr.org/2016/892. 5

[59] M. Geisler. Viff: Virtual ideal functionality framework, 2007. 5

[60] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 243–261, Leuven, Belgium, July 2–4, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-93387-0_13. 5

[61] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016. 6

[62] Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001. ISBN 0-521-79172-3 (hardback). 18

[63] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004. ISBN ISBN 0-521-83084-2 (hardback). 18

[64] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press. doi: 10.1145/28395.28420. 2, 8

[65] S. Dov Gordon, Feng-Hao Liu, and Elaine Shi. Constant-round MPC with fairness and guarantee of output delivery. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 63–82, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany. doi: 10.1007/978-3-662-48000-7_4. 18

[66] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 59–85, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-03332-3_3. 7

[67] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. *Journal of Cryptology*, 31(3):798–844, July 2018. doi: 10.1007/s00145-017-9271-y. 28

[68] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010: 17th Conference on Computer and Communications Security*, pages 451–462, Chicago, Illinois, USA, October 4–8, 2010. ACM Press. doi: 10.1145/1866307.1866358. 8, 94

[69] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *CoRR*, abs/1711.05189, 2017. 6

[70] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *21st Annual ACM Symposium on Theory of Computing*, pages 44–61, Seattle, WA, USA, May 15–17, 1989. ACM Press. doi: 10.1145/73007.73012. 91

[71] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume

2729 of *Lecture Notes in Computer Science*, pages 145–161, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany. doi: 10.1007/978-3-540-45146-4_9. 91

[72] Yuval Ishai, Ranjit Kumaresan, Eyal Kushilevitz, and Anat Paskin-Cherniavsky. Secure computation with minimal interaction, revisited. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 359–378, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany. doi: 10.1007/978-3-662-48000-7_18. 7, 81

[73] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1539–1556. ACM Press, November 11–15, 2019. doi: 10.1145/3319535.3339818. 8

[74] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 593–599. ACM, 2005. 5

[75] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi: 10.1145/3243734.3243805. 26

[76] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1575–1590, Virtual Event, USA, November 9–13, 2020. ACM Press. doi: 10.1145/3372297.3417872. 4

[77] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 549–560, Berlin, Germany, November 4–8, 2013. ACM Press. doi: 10.1145/2508859.2516744. 7

[78] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press. doi: 10.1145/2976749.2978357. 3, 4, 7, 94, 95

[79] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-78372-7_6. 3, 7

[80] Joe Kilian. Founding cryptography on oblivious transfer. In *20th Annual ACM Symposium on Theory of Computing*, pages 20–31, Chicago, IL, USA, May 2–4, 1988. ACM Press. doi: 10.1145/62212.62215. 91

[81] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 54–70, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany. doi: 10.1007/978-3-642-40084-1_4. 8, 91, 95

[82] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498, Reykjavik, Iceland, July 7–11, 2008. Springer, Heidelberg, Germany. doi: 10.1007/978-3-540-70583-3_40. 2, 28, 38, 55, 82, 99

[83] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 2013. 8

[84] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Com-*

puter Science, pages 440–457, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany. doi: 10.1007/978-3-662-44381-1_25. 28, 38, 55, 82, 99

[85] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security 2021: 30th USENIX Security Symposium*, 2021. https://eprint.iacr.org/2020/592. vi, 2, 6, 7, 9, 11, 20, 27, 41, 61, 73, 103, 149, 165

[86] Nishat Koti, Arpita Patra, and Ajith Suresh. MPCLeague: Robust and Efficient Mixed-protocol Framework for 4-party Computation. In *DPML'21 at ICLR Workshop*, 2021. https://dp-ml.github.io/2021-workshop-ICLR/files/9.pdf. vi

[87] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *ISOC Network and Distributed System Security Symposium – NDSS 2022 (To Appear)*. The Internet Society, 2022. https://eprint.iacr.org/2021/755. vi, 6, 8, 11, 61

[88] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. 2014. https://www.cs.toronto.edu/~kriz/cifar.html. 122, 152

[89] J. Launchbury, D. Archer, T. DuBuisson, and E. Mertens. Application-scale secure multiparty computation. In *ESOP*, 2014. 5

[90] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/. 152

[91] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, pages 2278–2324, 1998. 6, 12, 152

[92] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. https://eprint.iacr.org/2016/046. 18

[93] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In Mihir Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 36–54, Santa Barbara, CA, USA, August 20–24, 2000. Springer, Heidelberg, Germany. doi: 10.1007/3-540-44598-6_3. 5

[94] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 319–338, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany. doi: 10.1007/978-3-662-48000-7_16. 26

[95] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 619–631, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. doi: 10.1145/3133956.3134056. 6

[96] Ximeng Liu, Lehui Xie, Yaopeng Wang, Jian Zou, Jinbo Xiong, Zuobin Ying, and Athanasios V. Vasilakos. Privacy and security issues in deep learning: A survey. *IEEE Access*, 9:4566–4593, 2021. 6

[97] Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren. EPIC: Efficient private image classification (or: Learning from the masters). In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 473–492, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-12612-4_24. 2, 5

[98] Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. Secure parallel computation on national scale volumes of data. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020: 29th USENIX Security Symposium*, pages 2487–2504. USENIX Association, August 12–14, 2020. 4, 7, 11, 61

[99] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 3–33, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-56877-1_1. 153

[100] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Praneeth Vepakomma, Abhishek Singh, Ramesh Raskar, and Hadi Esmaeilzadeh. Privacy in deep learning: A survey. *CoRR*, abs/2004.12254, 2020. 6

[101] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 35–52, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi: 10.1145/3243734.3243760. xvii, xviii, 2, 3, 4, 5, 6, 8, 11, 16, 20, 27, 29, 41, 103, 105, 108, 110, 113, 122, 126, 127, 128, 129, 133, 141, 149, 151, 153, 155

[102] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press. doi: 10.1109/SP.2017.12. 2, 3, 5, 6, 11, 27, 90, 96, 105, 149, 150, 153, 170

[103] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 591–602, Denver, CO, USA, October 12–16, 2015. ACM Press. doi: 10.1145/2810103.2813705. 2, 5, 7

[104] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1):1–35, January 2005. doi: 10.1007/s00145-004-0102-6. 91

[105] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy*, pages 334–348, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press. doi: 10.1109/SP.2013.30. 5

[106] Satsuya Ohata and Koji Nuida. Communication-efficient (client-aided) secure two-party protocols and its application. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020: 24th International Conference on Financial Cryptography and Data Security*, volume 12059 of *Lecture Notes in Computer Science*, pages 369–385, Kota Kinabalu, Malaysia, February 10–14, 2020. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-51280-4_20. x, 8, 10, 98, 99, 104

[107] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages 254–283, San Francisco, CA, USA, February 24–28, 2020. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-40186-3_12. 7

[108] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany. doi: 10.1007/3-540-48910-X_16. 92

[109] Arpita Patra and Divya Ravi. On the exact round complexity of secure three-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 425–458, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-96881-0_15. 2

[110] Arpita Patra and Ajith Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *ISOC Network and Distributed System Security Symposium – NDSS 2020*, San Diego, CA, USA, February 23-26, 2020. The Internet Society. vi, 2, 3, 5, 6, 7, 9, 11, 16, 20, 27, 41, 103, 149, 151

[111] Arpita Patra, Pratik Sarkar, and Ajith Suresh. Fast actively secure OT extension for short secrets. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society. 91, 95

[112] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation. In *14. IEEE International Workshop on Hardware-Oriented Security and Trust (HOST'21) (To Appear)*, 2021. https://eprint.iacr.org/2021/1153. vii

[113] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security 2021: 30th USENIX Security Symposium*, 2021. https://eprint.iacr.org/2020/1225. vi, 2, 3, 5, 8, 9, 10, 11, 20, 27, 90, 103, 110, 113, 114, 122, 128, 129, 133, 137, 138, 141, 145, 146

[114] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 515–530, Washington, DC, USA, August 12–14, 2015. USENIX Association. 104

[115] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes*

in Computer Science, pages 125–157, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-78372-7_5. 104

[116] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 401–431, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-26954-8_13. 12

[117] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-17659-4_5. 104

[118] Pille Pullonen and Sander Siim. Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *Lecture Notes in Computer Science*, pages 172–183, San Juan, Puerto Rico, January 30, 2015. Springer, Heidelberg, Germany. doi: 10.1007/978-3-662-48051-9_13. 150

[119] Deevashwer Rathee, Thomas Schneider, and K. K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In Yi Mu, Robert H. Deng, and Xinyi Huang, editors, *CANS 19: 18th International Conference on Cryptology and Network Security*, volume 11829 of *Lecture Notes in Computer Science*, pages 347–359, Fuzhou, China, October 25–27, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-31578-8_19. 92, 94, 95

[120] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18: 13th ACM Symposium on Information, Computer and Communications Security*, pages 707–721, Incheon, Republic of Korea, April 2–6, 2018. ACM Press. 2, 3, 5, 151

[121] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India*,

volume 11898 of *Lecture Notes in Computer Science*, pages 227–249, Hyderabad, India, December 15–18, 2019. Springer, Heidelberg, Germany. doi: 10.1007/978-3-030-35423-7_12. 3, 8

[122] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. DeepSecure: Scalable provably-secure deep learning. Cryptology ePrint Archive, Report 2017/502, 2017. https://eprint.iacr.org/2017/502. 6

[123] Ashish P. Sanil, Alan F. Karr, Xiaodong Lin, and Jerome P. Reiter. Privacy preserving regression modelling via distributed computation. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004*, pages 677–682. ACM, 2004. 5

[124] Shreya Sharma, Chaoping Xing, and Yang Liu. Privacy-preserving deep learning with SPDZ. In *The AAAI Workshop on Privacy-Preserving Artificial Intelligence*, 2019. 4

[125] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 1310–1321, Denver, CO, USA, October 12–16, 2015. ACM Press. doi: 10.1145/2810103.2813687. 6

[126] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy*, pages 3–18, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press. doi: 10.1109/SP.2017.41. 16

[127] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 6, 12, 152

[128] Aleksandra B. Slavkovic, Yuval Nardi, and Matthew M. Tibbits. Secure logistic regression of horizontally and vertically partitioned distributed databases. In *Workshops Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), October 28-31, 2007, Omaha, Nebraska, USA*, pages 723–728. IEEE Computer Society, 2007. 5

[129] Lushan Song, Haoqi Wu, Wenqiang Ruan, and Weili Han. Sok: Training machine learning models over multiple sources with privacy preservation. *CoRR*, abs/2012.03386, 2020. 6

[130] Stanford. CS231n: Convolutional Neural Networks for Visual Recognition. URL https://cs231n.github.io/convolutional-networks/. 103

[131] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 601–618, Austin, TX, USA, August 10–12, 2016. USENIX Association. 16

[132] Jaideep Vaidya, Hwanjo Yu, and Xiaoqian Jiang. Privacy-preserving SVM classification. *Knowl. Inf. Syst.*, 14(2):161–178, 2008. 5

[133] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019 (3):26–49, July 2019. doi: 10.2478/popets-2019-0035. 2, 4, 5, 6, 7, 103, 149

[134] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proceedings on Privacy Enhancing Technologies*, 2021(1):188–208, January 2021. doi: 10.2478/popets-2021-0011. 6, 7, 149, 151

[135] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 21–37, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. doi: 10.1145/3133956.3134053. 26

[136] Shuang Wu, Tadanori Teruya, and Junpei Kawamoto. Privacy-preservation for stochastic gradient descent application to secure logistic regression. 2013. 6

[137] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press. doi: 10.1109/SFCS.1982.38. 2, 21, 22

[138] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press. doi: 10.1109/SFCS.1986.25. 2

[139] Hwanjo Yu, Jaideep Vaidya, and Xiaoqian Jiang. Privacy-preserving SVM classification on vertically partitioned data. In *Advances in Knowledge Discovery and Data Mining,*

10th Pacific-Asia Conference, PAKDD 2006, Singapore, April 9-12, 2006, Proceedings, volume 3918 of *Lecture Notes in Computer Science*, pages 647–656. Springer, 2006. 5

[140] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. doi: 10.1007/978-3-662-46803-6_8. 2, 28