Secure Computation Protocol Suite for Privacy-Conscious Applications

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Koti Nishat Saipanmuluk

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

June, 2023

Declaration of Originality

I, **Koti Nishat Saipanmuluk**, with SR No. **04-04-00-10-12-17-1-14611** hereby declare that the material presented in the thesis titled

Secure Computation Protocol Suite for Privacy-Conscious Applications

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2017-2023**.
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: 21 June, 2023 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 beloved

Nana Abba

for being a great source of motivation and inspiration,

along with my loving

Parents & Naushad

whose affection, support and words of encouragement have helped me sail through this journey.

Acknowledgements

I would like to begin by thanking the Almighty for giving me the strength to successfully complete this thesis. Alhamdulillah!

I would like to express my deepest gratitude to my PhD advisor, Prof. Arpita Patra, for her support and guidance throughout my journey. Her invaluable insights and feedback have been instrumental in shaping me as a researcher. I am especially grateful for her patience and encouragement, which helped me stay motivated throughout the process and helped in completing my thesis. Working with her has been an exceptional experience, and I am honoured to have had this opportunity.

I also take this opportunity to express warm gratitude to all my colleagues at the CrIS lab, whose presence has been a great source of inspiration and joy. Their understanding, emotional support and encouragement have been invaluable in maintaining my emotional well-being throughout this endeavour. I would like to extend my sincere thanks to Ajith Suresh, who played a crucial role in launching my research journey. We have had many engaging discussions solving research problems which not only helped me build new insights but also boosted my confidence. I would also like to express my heartfelt gratitude to Varsha Bhat Kukkala, who has been one of my strong support systems, both academically and personally. I am immensely grateful for all the learnings I had from her and will treasure them forever. I cannot thank her enough for her unwavering support during the most critical stages of my PhD. A special mention to Bhavish Raj Gopal, with whom I have had the privilege of collaborating on multiple projects alongside Varsha Bhat Kukkala. Although I initially met them both as colleagues, our shared experiences and camaraderie have fostered a strong friendship, and I consider myself fortunate to have found in them some of my closest friends through this journey. I would like to take a moment to also thank Protik Paul and Shravani Patil whose support has been vital in helping me navigate through my PhD journey. Be it academic discussions or having fun conversations, it has been a pleasure knowing them and working with them. I also take this opportunity to thank all my co-authors for their invaluable support.

I would like to express my gratitude to Ms Padmavathi, Ms Kushael and the other staff at

the CSA office for their help on the administrative front. I would like to acknowledge financial support from the Centre for Networked Intelligence (a Cisco CSR initiative), National Security Council, India, Indian Urban Data Exchange, and the Google Cloud Platform. I also extend my sincere thanks to Prof. Purushothama B.R. (NIT Goa), who played a pivotal role in inspiring me to pursue a PhD, without whose encouragement, this thesis would not have been possible.

Last but not least, I am deeply grateful to my parents and grandparents, who have been a constant source of support and inspiration for me. I would like to extend special thanks to my brother, Saqib Koti, who has always been there whenever I needed him, whether it was to take a leisurely walk or simply have a chat over a cup of coffee. My heartfelt thanks to my husband, Naushad Ahmed, for being my pillar of strength and support. I am indebted for his presence in my life and his unwavering belief in me, even during the most challenging times of my academic journey. Thank you for being my rock throughout. I would like to thank my entire family for their understanding and constant support along the way. My efforts are as much theirs as they are mine, and I dedicate this thesis to them. Thank you!

Abstract

As an alternative to performing analytics in the clear, there is an increasing demand for developing privacy-preserving solutions that aim to protect sensitive data while still allowing for its efficient analysis. Among the various privacy-enhancing technologies, secure multiparty computation (MPC) is a promising approach that enables multiple parties (n) to jointly process their private inputs while ensuring that no coalition of at most $t < n$ parties, under the control of an adversary, learns any information other than the intended output. In this thesis, we identify various such real-world applications that demand privacy-preserving solutions and address these via MPC. We consider a broad range of applications that span across healthcare, finance and even social sectors. For each application under consideration, we identify the desirable MPC setting (e.g., number of computing parties n) and security notion to be achieved when designing the solution. Based on this, we either design new MPC frameworks that provide improved security guarantees and efficiency or enhance the existing frameworks.

Although we make application-specific design choices, the common theme while designing secure protocols for all is to design as efficient a solution as possible. In this regard, we make the following common design choices across all applications. First, we consider an honest majority among the computing parties (i.e., $t < n/2$), which is known to render efficient protocols in comparison to the dishonest majority (i.e., $t < n$). Second, we focus on designing secure protocols in the preprocessing paradigm, where expensive input-independent computations are pushed onto a preprocessing phase, thereby making way for a fast and efficient input-dependent online phase. Finally, our protocols are designed to operate on the ring algebraic structure to capitalize on the efficiency gains obtained from utilizing the CPU architecture. We next elaborate on the specific applications considered in the thesis and the contributions therein.

Secure computation over graphs via traditional security notion Operating on graph-structured data is ubiquitous due to the modelling capabilities of graphs, and this finds use in analysing various systems like social networks, biological networks, transportation networks, etc. However, privacy concerns arise when analysing graphs that model sensitive data. To

address this, we design privacy-preserving solutions for two popular graph algorithms—local clustering and graph convolutional networks.

– *Secure local clustering:* Identifying a cluster around a target node in a graph, termed local clustering, finds use in several applications, including fraud detection, targeted advertising, community detection, etc. We design solutions for privacy-preserving local clustering, which is done for the first time in the literature. Keeping efficiency in mind for large graphs, we build over the best-known honest-majority 3-party framework of SWIFT (USENIX'21) and enhance it with some of the necessary yet missing primitives. To further enhance efficiency, we design the protocols using the GraphSC paradigm, which provides a generic secure framework for efficiently evaluating graph algorithms. Since this paradigm relies on a secure shuffle primitive, we also design an efficient secure 3-party shuffle protocol.

We note that secure shuffle is a versatile primitive that finds widespread use in various other applications as well (which may not involve computations over a graph), such as electronic voting, oblivious RAM, and anonymous broadcast, to name a few. Hence, as a by-product of our shuffle protocol, we are also able to securely realise an *anonymous broadcast* system. As the name suggests, anonymous broadcast enables a set of N clients to anonymously broadcast their messages while guaranteeing that none learns about the association between a message and the identity of its sender. Hence, while anonymous broadcast may not be inherently associated with graph computations, we diverge slightly to demonstrate how our shuffle protocol can be employed to realize anonymous broadcast in the 3-party setting, as considered in prior works. In the process, not only do we design a more efficient anonymous broadcast system compared to the state-of-the-art, but our system also provides improved security guarantees and properties such as censorship resistance that were missing in the prior solution.

– *Secure graph convolutional networks:* Graph convolutional networks (GCNs) are gaining popularity due to their ability to effectively model and learn from complex graph-structured data. We put forth Entrada, a framework for securely evaluating GCNs. For efficiency and accuracy reasons, Entrada builds over the 4-party framework of Tetrad (NDSS'22) and enhances the same by providing the necessary primitives. Moreover, Entrada leverages the GraphSC paradigm to further enhance efficiency and entails designing a secure and efficient shuffle protocol specifically in the 4-party setting. This, to the best of our knowledge, is done for the first time and may be of independent interest.

Stepping beyond traditional security for financially and socially relevant problems Most protocols in the small-party setting that are designed to attain the strongest security notion of guaranteed output delivery (GOD), rely on entrusting an honest party, identified as

the trusted third party (TTP), with inputs in the clear to carry out the computation. However, this may not be desirable for certain applications that deal with highly sensitive data. Another drawback of traditional MPC protocols is the view leakage attack, where a malicious adversary may send its view to an honest party, thereby enabling the latter to obtain the underlying secret information. To address these drawbacks in the traditional MPC definition, Alon et al.(CRYPTO'20) propose the notion of MPC with Friends and Foes (FaF). Thus, departing from the traditional MPC model, we identify the need to design FaF-secure MPC protocols for applications that deal with highly sensitive information, where information leakage must be prevented even against quorums of honest parties. Specifically, we consider the applications of secure dark pools and secure allegation escrow systems. Keeping efficiency at the centre stage, we design FaF-secure 5-party computation protocols (5PC) that consider one malicious and one semi-honest corruption and constitute the optimal setting for attaining an honest majority.

– *Secure dark pools:* Dark pools are private security exchanges that allow investors to trade financial instruments outside of the prying eyes of the public and ensure the trade remains unexposed until it is completed. Dark pools are traditionally operated by centralized trusted brokers, who, in the past, have been known to misuse insider information. This necessitates designing solutions that guarantee privacy even against the dark pool operator. Hence, given the sensitive nature of financial data that is involved in the computation and the drawbacks present in the traditional MPC solutions, we design FaF secure solutions for the same in the 5PC setting. We design improved solutions for the continuous double auction (CDA) and volume-based matching algorithms that are used in dark pools. We benchmark the performance of these secure matching algorithms and observe improvements in comparison to the prior works.

– *Secure allegation escrow system:* The rising issues of malpractices have led victims to seek comfort by acting in unison against common perpetrators (e.g., the #MeToo movement). To increase trust in the system, cryptographic solutions are being designed to realize secure allegation escrow (SAE) systems. In this regard, we identify privacy issues present in prior works and put forth an SAE system to arrest all these breaches. Given the highly sensitive nature of allegation data, we choose to realise the system under FaF security as opposed to traditional security notions. We also provide additional features which were absent in the state of the art. We benchmark the proposed system with the FaF secure 5PC protocols to showcase the practicality of our solution.

Secure computation with a constant number of parties Unlike the applications considered above that demanded operating with a specific number of parties, the latter may vary depending on the application. Hence, we provide a generalization which allows instantiating the

Abstract

MPC protocol with an arbitrary (constant) number of parties (n). Our generalized protocols continue to operate in the honest majority setting to capitalize on the efficiency benefits that this setting provides over the dishonest majority, which thereby facilitates attaining an efficient solution for the end application. We design two different protocols that are secure against a semi-honest and a malicious adversary, respectively. We also design a wide range of building blocks that facilitate the secure realization of various applications, including but not limited to genome sequence matching, biometric matching, and even deep neural networks, and showcase the practicality of the designed protocols by benchmarking these applications.

In this way, we design a range of building blocks in various MPC settings that can facilitate secure realizations for the above-mentioned privacy-conscious applications.

Publications based on this thesis

The results of this dissertation have led to the following publications. The author names in the publications are ordered lexicographically based on the surname. Workshops/poster acceptances are marked in this colour.

Papers accepted

1. **Nishat Koti**, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal. *Shield: Secure Allegation Escrow System with Stronger Guarantees* [139]. The Web Conference, 2023 (CORE A*). IEEE S&P 2022, CANS 2022.

2. **Nishat Koti**, Shravani Patil, Arpita Patra, Ajith Suresh. *MPClan: Protocol Suite for Privacy-Conscious Computations* [140]. Journal of Cryptology, 2023 (CORE A*). NDSS 2022.

3. Pranav Shriram A, **Nishat Koti**, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, Somya Sangal. *Ruffle: Rapid 3-Party Shuffle Protocols* [212]. PoPETs, 2023 (CORE A).

4. Pranav Shriram A, **Nishat Koti**, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal. *Find Thy Neighbourhood: Privacy-Preserving Local Clustering* [211]. PoPETs, 2023 (CORE A).

5. **Nishat Koti**, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal. *PentaGOD: Stepping beyond Traditional GOD with Five Parties* [137]. ACM CCS, 2022 (CORE A*). CANS 2022.

Papers under submission

1. **Nishat Koti**, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal. *Entrada to Secure Graph Convolutional Networks.*

Publications outside this thesis

Papers accepted

1. Banashri Karmakar, **Nishat Koti**, Arpita Patra, Sikhar Patranabis, Protik Paul, Divya Ravi. *Asterisk: Super-fast MPC with a Friend* [121]. IEEE S&P 2024 (CORE A*).

2. Pranav Jangir, **Nishat Koti**, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, Somya Sangal. *Vogue: Faster Computation of Private Heavy Hitters* [117]. TDSC, 2023 (CORE A). CCS 2022 (Poster).

3. **Nishat Koti**, Arpita Patra, Rahul Rachuri and Ajith Suresh. *Tetrad: Actively Secure 4PC for Secure Training and Inference* [138]. NDSS, 2022 (CORE A*). PPML 2021 (CCS).

4. Aditya Hegde, **Nishat Koti**, Varsha Bhat Kukkala, Shravani Patil, Arpita Patra, Protik Paul. *Attaining GOD Beyond Honest Majority with Friends and Foes* [104]. Asiacrypt, 2022 (CORE A).

5. **Nishat Koti**, Mahak Pancholi, Arpita Patra and Ajith Suresh. *SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning* [136]. USENIX Security, 2021 (CORE A*). PRIML/PPML 2020 (NeurIPS), DPML 2021 (ICLR).

Papers under submission

1. **Nishat Koti**, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal. *Mosaic: Fast and Secure Pattern Matching.*

2. Yongqin Wang, Pratik Sarkar, **Nishat Koti**, Arpita Patra, Murali Annavaram. *CompactTag: Minimizing Computation Overheads in Actively-Secure MPC for Deep Neural Networks.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's world is seeing a visible transition from offline services to a heavy dependency on online platforms for banking, socializing, healthcare, etc. This is leading to an increased user presence online, which leaves a trail of online activity and personal data over the Internet. The widespread sharing of such personal information and its collection has led to several high-profile data breaches and scandals [96, 112]. As a result, individuals and organizations are becoming more conscious of the potential risks associated with the collection, use, and storage of personal data. Several data privacy laws are being implemented, such as the European Union's General Data Protection Regulation (GDPR) and California's Consumer Privacy Act (CCPA), which are helping raise awareness about data privacy. With the steady incline in the awareness of data privacy, we are witnessing a paradigm shift in healthcare, finance, and various other sectors involved in processing a large amount of sensitive user data. Various privacy-preserving practices are being adopted to reassure users and provide them with the highest level of security guarantees while still allowing for useful insights to be derived from private data. Given the ease of accommodating multiple users and its computational efficiency, many real-world applications are preferring the use of secure multiparty computation (MPC) to perform privacy-preserving computations.

Informally, MPC enables n mutually distrusting parties to compute a function over their private inputs while ensuring privacy against a coalition of at most t parties. The distrust among the parties is captured via the notion of a *centralized* adversary which controls up to t parties. These t parties are said to be *corrupt* while the rest are assumed to be *honest*. Based on the number of parties t that the adversary corrupts among the total n parties, the setting is categorized as *honest majority* or *dishonest majority*. In the honest majority setting, the majority of the parties are honest and $t < n/2$. On the other hand, in the dishonest majority setting, the majority of the parties are corrupt (dishonest), i.e. $t < n$. Further, depending on its

behaviour, the adversary can be categorized as either *semi-honest* or *malicious* [89]. Semi-honest adversary models the corruption scenario where the corrupt parties are restricted to follow the protocol and cannot deviate arbitrarily. On the other hand, in the stronger notion of malicious corruption, the adversary can arbitrarily deviate from protocol specification. Moreover, in the malicious adversarial model, various security notions can be attained. These are as follows.

1. *Security with abort*: This security notion allows the adversary alone to learn the output of the computation and abort the computation at will.

2. *Security with fairness:* This security notion ensures that either all parties learn the output of the computation or none do.

3. *Security with guaranteed output delivery (GOD) or robustness:* This security notion guarantees that regardless of the adversary's misbehaviour, all honest parties learn the output of the computation.

There are various applications for which privacy-preserving solutions have been designed via MPC under the different settings described above. These include secure auctions [28], privacy-preserving machine learning [174, 173, 49, 193, 50, 136, 138, 38, 220, 61], secure recommendation systems [210, 109], real-world deployments such as the Estonian study on the correlation between tax data and educational records [27], and the study of salary inequities across various employees in the city of Boston [24], to name a few. In this thesis, we explore various other applications that are of interest and design privacy-preserving solutions for the same via MPC. Broadly, these applications range from secure computations on graph-structured data, secure computation for applications of social relevance, and secure computation in the financial sector and the healthcare domain. In designing the privacy-preserving solutions for each of these applications, we identify the desirable MPC setting (e.g., number of computing parties n) and security notion to be achieved for the application under consideration. Based on this, for each application, we design new MPC frameworks or improve upon the existing ones. Note, however, that the secure protocols for the applications are designed to make black-box use of the underlying MPC. This not only allows the application to inherit the latter's security guarantees and efficiency but also opens up the possibility of utilizing the future advancements of MPC in a seamless way. Before we elaborate on the considered applications, we note that although we make application-specific design choices, the common theme while designing secure protocols for all is to design as efficient a solution as possible. In this regard, we make the following common design choices across all applications in an attempt to obtain a practically efficient solution.

1. *Corruption threshold:* It is well known that an honest majority among the parties enables designing efficient protocols in comparison to a dishonest majority. Moreover, an honest majority among the parties is necessary to achieve the strongest security notion of GOD [55]. Hence, to design efficient secure solutions for the applications under consideration, we focus on designing maliciously secure protocols in the honest majority setting.

2. *Preprocessing paradigm:* An essential factor to be considered when designing a practically efficient protocol is its response time, which accounts for the time taken from submission of the input, its processing, to delivery of the output. To minimize response time, we focus on designing secure protocols in the preprocessing paradigm [64, 65, 127, 128, 21, 66, 58, 200, 129, 193, 50]. Here, expensive *data-independent* computations are carried out in a preprocessing phase, thereby making way for a fast *data-dependent* online phase.

3. *Algebraic structure:* To further enhance efficiency by utilizing the underlying CPU architecture, several protocols work over rings [173, 136, 50, 220, 138, 169]. We follow this approach and design secure protocols operating over the ring $\mathbb{Z}_{2^\ell}$.

1.1 Overview of the applications considered in the thesis

1.1.1 Secure computation over graphs via traditional security notion

Many real-world applications, such as communication networks, traffic networks, social networks, etc., generate an enormous amount of unstructured data. A natural and conventional approach to model such data is via graphs [218] owing to their highly expressive capabilities and ease of processing. Specifically, modelling a system as a graph involves representing each entity as a node and capturing their interactions as edges. Further, the nodes and edges may also be associated with data components depending on the underlying system. In a simplistic example of a friendship network, each node denotes an individual, an edge denotes friendship between two individuals, and each node may additionally store data associated with the individual, such as name, age, gender, etc.

There exist various techniques that enable deriving meaningful information about the system modelled as a graph. Some of these include clustering the nodes of the graph, predicting new edges to identify relations between nodes, and computing various centrality measures to determine the importance of the nodes, to name a few. While running these graph-based algorithms, most of them assume that the topology of the graph is available in its entirety [133, 214, 160, 215, 86]. This may not always be the case. For example, consider the COVID-

19 contact network, where users are modelled as nodes, and their interactions are modelled as edges. Here, the contact network may be held in a distributed fashion where each node is aware only of its neighbouring nodes. In general, the graph may be distributed across multiple data owners, such that each of them is aware of only a subset of the edges in the overall graph. Performing computations on such distributed graphs, thus, requires data owners to disclose their view of the graph, which may not only comprise the topology of the graph they hold but also sensitive data associated with the corresponding nodes/edges. However, data privacy concerns prevent them from doing so. Hence, the distributed nature of the global graph, clubbed with privacy concerns, makes it challenging to perform computation when accounting for the entire graph topology. This motivates the need for designing privacy-preserving solutions that allow computing on the global graph without requiring data owners to reveal their view of the graph.

Having motivated the need to perform privacy-preserving computations on graph-structured data, we consider two popular graph algorithms which demand designing privacy-preserving solutions—local clustering and graph convolutional networks. We next discuss these applications, the need for privacy in these applications and provide an overview of the MPC setting we consider while designing a secure solution for the same.

1.1.1.1 Secure local clustering

One useful technique to analyse graph-based data is that of clustering, which allows analyzing the topology of a graph to identify entities that are related to each other [46, 209, 182, 120, 153]. At a high level, clustering is the process of grouping together similar nodes in a graph and finds use in several applications such as community detection in social networks [134, 225, 181, 149], behavioural analysis [165, 68], structural characterization of chemical networks [151, 131, 76, 201], etc. Most clustering algorithms are designed to categorize every node into its specific cluster, termed *global* clustering. However, more often than not, one may be interested in identifying a *local* cluster around a specific node. For example, consider the COVID-19 contact network. Identifying the close-knit cluster around a user who has recently contracted the virus is important and enables the implementation of preventive measures. A global clustering algorithm may not correctly identify such a local cluster. Other examples where identifying a local cluster is of importance include targeted advertising [20], fraud detection [166], etc. Graph-based local clustering algorithms in the literature [214, 160, 215, 86] assume that the topology of the graph is available in its entirety. However, as described earlier, the graph may be distributed, and together with the privacy concerns, introduces challenges. Hence, our goal is to perform local clustering in a privacy-preserving manner. This will facilitate multiple data

owners to perform local clustering on a graph that is held in a distributed fashion while ensuring that none of them is required to disclose their data in the clear. We realize this goal via MPC, where the private input is the graph (held distributedly) and a target seed node, while the output comprises a local cluster around the seed node.

We focus on designing an efficient solution by relying on the threshold-optimal setting of 3-party computation (3PC) with honest majority [136], and aim to attain the strongest security of GOD. In the process, we enhance the framework of [136] by incorporating the missing primitives. Further, we use the GraphSC paradigm [179, 13], which provides a generic framework for evaluating graph algorithms securely, to design efficient solutions. The GraphSC paradigm [13] heavily relies on a secure shuffle primitive. This primitive allows randomly permuting the elements of the ordered set while ensuring that the permutation, as well as the elements in the ordered set, are not known on clear. Hence, we also design an efficient secure 3-party shuffle protocol. The use of our shuffle protocol helps to further improve the efficiency of the designed system.

Anonymous broadcast The secure shuffle protocol, as mentioned above, finds wide-spread use as a primitive in various other applications as well (which may not involve computations over a graph) such as electronic voting [180, 98], oblivious RAM [47, 17], anonymous broadcast [80], to name a few. Anonymous broadcast is one application where a secure shuffle protocol forms an integral part. Elaborately, anonymous broadcast, as the name suggests, enables a set of N clients to anonymously broadcast their messages while guaranteeing that none learns about the association between a message and the identity of its sender. A simple solution to realize this is for the clients to send their messages to a centralized server, which can output the randomly shuffled messages back to the clients. However, to guarantee the privacy of messages against the server and ensure that the server, too, does not learn about the association between the client and its message, the shuffle can be realized via MPC. Elaborately, the clients secret-share their messages to a set of three servers (the three parties in the 3PC setting considered above while designing the secure shuffle protocol), such that no single server can derive any information about the client messages based on the shares that it receives. The servers then invoke a secure shuffle protocol on the received secret-shared messages, and reconstruct the shuffled output towards the clients. In this way, an anonymous broadcast system essentially relies on a secure shuffle protocol. Hence, although anonymous broadcast does not inherently compute over graphs, we digress slightly and showcase how our shuffle protocol can be used to realize the application of anonymous broadcast in the 3-party setting, as considered in prior works [80]. In the process, not only do we design a more efficient solution to anonymous broadcast

compared to the state-of-the-art solution of [80], but our system also provides improved security guarantees and properties such as censorship resistance that were missing in the prior solution.

1.1.1.2 Secure graph convolutional networks

Resuming the discussion on secure computation over graphs, we next study a powerful machine learning technique for leveraging graph-structured data—graph convolutional networks (GCN). GCNs are a type of convolutional neural network designed to operate on graph-structured data such as social networks, citation networks, etc. GCNs find use in a diverse set of applications, including traffic prediction, rumour detection, targeted advertising and recommendation systems, to name a few. The massive size of the input graph and multiple layers of the neural network results in GCNs being computationally intensive. Hence, various platforms, such as Neptune ML by Amazon, offer GCN training and inference as a service, where a data owner (client) provides its input data in clear to the hired servers that carry out the computations on behalf of the client. However, the input may comprise private information, such as the graph topology, node/edge features, etc., that the client may not wish to disclose to the servers. Such threats to privacy are evident when dealing with data with respect to health records [157], social networks [156], financial transactions [119], etc. This necessitates designing techniques that allow clients to outsource the computation such that their inputs remain private while enabling servers to operate on the private inputs. Although paradoxical, such privacy-preserving evaluation of GCNs can be achieved via MPC. We specifically consider the task of node classification via GCNs, where the graph comprises nodes, some of which are labelled, and the task is to assign labels to the unlabelled nodes.

MPC protocols are known to have an efficiency overhead in comparison to cleartext computation. Hence, for compute-intensive GCNs, it is imperative to design efficient MPC protocols. We note that works in the literature look at securely computing neural networks (NN) via MPC [173, 136, 138]; however, GCNs have not been well explored. Prior works that explore GCNs only consider performing secure inference over relatively older GCN models [208]. Although one may consider extending these works to securely realize GCNs, they either lack the necessary primitives or the desired level of security, efficiency and accuracy. Elaborately, several works trade off accuracy for efficiency by relying on MPC-friendly alternatives for non-linear functions [138, 50, 173]. Instead, we strive to design *accurate* protocols for GCN evaluation while *not* compromising on *efficiency*. To achieve an efficient solution, we build Entrada, a secure framework for evaluating GCNs, in the 4-party computation (4PC) setting with an honest majority. To further enhance the efficiency, we also use the GraphSC paradigm [179, 13].

1.1.2 Stepping beyond traditional security for financially and socially relevant problems

While designing privacy-preserving solutions for various applications, it is desirable to achieve the strongest security notion of robustness or GOD. Recall that robustness ensures that all parties obtain the output regardless of the adversary's misbehaviour. Absence of robustness leads to denial of service and economic losses. Moreover, a robust solution increases the users' trust in the system and encourages higher user participation. Hence, guaranteeing robustness is crucial for the seamless adoption of a privacy-preserving solution. With that said, it is worth noting that most GOD protocols in the literature [36, 30, 31, 136] rely on an *honest* party identified as the trusted third party (TTP) to carry out the computation if misbehaviour by a malicious adversary is detected. Elaborately, the parties entrust the TTP with their inputs, which carries out the computation and delivers the output to all. According to the standard security definition, this leakage of inputs towards a TTP is not considered a privacy breach. This is because the TTP is deemed to be honest, and the goal is to protect against information leakage towards an adversary. However, entrusting a TTP with all the inputs may not be acceptable for certain real-world applications that deal with highly sensitive data.

Another drawback of traditional MPC protocols is the view leakage attack. While executing an MPC protocol, nothing prevents a malicious adversary from sending its view (messages exchanged during the protocol run), which consists of the view of t corrupt parties, to an honest party. This is not treated as an attack in the traditional security definition since an honest party is expected to discard non-protocol messages, unlike a semi-honest one. However, if this honest party turns rogue in the future, the party can obtain all the information about the private inputs and the intermediate values generated during the computation. This holds because it would now possess information with respect to $t+1$ parties (t views received from the adversary and its own view), which suffices to obtain the underlying secret information. This, too, goes against the goal of providing privacy in a system.

To address these drawbacks of the traditional MPC security definition, Alon et. al. [5] proposes a new definition called MPC with Friends and Foes (FaF). This definition requires honest parties' inputs to be protected against not only the adversary (foes) but also from quorums of other honest parties (friends). This is modelled by a decentralized adversary which comprises two different *non-colluding* adversaries—(i) a malicious adversary that corrupts any subset of at most t out of n parties, (ii) a semi-honest adversary that corrupts any subset of at most $h^\star$ out of the remaining $n-t$ parties. A protocol secure against such an adversary is said to be $(t, h^\star)$-FaF secure. Further, the FaF model requires security to hold even when an

adversary sends its view to other parties. Hence, departing from the traditional MPC model, we identify the need to design FaF-secure MPC protocols for applications that deal with highly sensitive information which needs protection from all forms of misuse. Specifically, we consider the applications of dark pools and allegation escrow systems. Before we elaborate on the applications, we briefly discuss the number of parties and corruption threshold considered in our design of FaF-secure protocols.

Small-party honest majority FaF model Alon et. al. [5] show that GOD can be achieved in the $(t, h^\star)$-FaF model iff $2t + h^\star < n$. Thus, obtaining GOD requires $n \geq 4$ for non-zero values of t and $h^\star$. Focusing on MPC with a small number of parties, observe that instantiating $n = 4$ and $t = h^\star = 1$ provides the optimal threshold for 4PC to achieve GOD. However, two corruptions result in a dishonest majority setting, which renders less efficient protocols than their honest majority counterparts. Hence, to design efficient protocols, we augment this setting with one additional honest party and design 5-party computation (5PC) protocols which are $(1, 1)$-FaF secure. We remark that while $(t, h^\star)$ can be instantiated with a varied range of values to attain GOD such that $2t + h^\star < 5$, we set $t = h^\star = 1$ because of the following reasons: (i) this results in an honest majority setting; (ii) we believe that $h^\star = 1$ suffices for most practical applications since honest parties (friends) are unlikely to collude with each other (note that when $h^\star = 1$, the only possible value of t is 1). We note that in the current setting of $n = 5$ and $t = h^\star = 1$, one could alternatively avoid the aforementioned weaknesses by deploying a traditional (5,2) malicious secure protocol since the latter protects against view leakage and also avoids reliance on a TTP when deployed in the presence of a single malicious party. However, since the traditional protocol is designed to cater to two malicious parties as opposed to one in our setting, it may lose out on performance. Hence, keeping efficiency for real-world applications at centre stage, the objective is to leverage the presence of a semi-honest party to design customised efficient $(1, 1)$-FaF secure protocols. We note that a traditional (n, t) malicious protocol is capable of protecting against view leakage attacks and avoids the reliance on a TTP as long as at most $t - 1$ parties are malicious.

1.1.2.1 Secure dark pools

Dark pools are private security exchanges that allow investors to trade (buy and/or sell) financial instruments such as securities (stocks, bonds etc.) outside of the prying eyes of the public and ensure the trade remains unexposed until it is completed. This allows investors to trade large blocks of securities privately and ensures the market is not impacted by the knowledge of such potential large-scale trade. For example, public knowledge of an institution trying to sell a

large portion of its shares would cause a sudden depreciation of its share value even before the transaction is completed. On the other hand, the market impact is known to be much smaller when the trade is reported after it is executed. This is the working principle underlying dark pools, which makes them a popular choice for trading. Dark pools are traditionally operated by trusted brokers who are made aware of the trade interests of the clients. They are then expected to find matching counter-parties within their network of private clients. The clients, in the process, place complete trust in the broker to not misuse the trade interests disclosed on clear. However, several instances have showcased misuse of insider information where dark pool operators have been fined for the same [186, 188, 189, 190, 184, 185, 187].

To guarantee complete privacy, the interest to trade must never be disclosed in the open, not even to the broker operating the dark pool. Ideally, matches between sellers and buyers must be found without disclosing this sensitive information. Thus, the problem can be modelled as an instance of MPC, where the private input is the data related to the trade, and the clients are interested in securely matching the possible trades. In this setting, rather than the dark pool being operated by a central trusted broker, it is emulated by an MPC protocol run among a set of parties. Clients secret-share their trade data to these parties in such a way that no subset, of at most t of these parties, learns any information. These parties are responsible for running the MPC protocol designed to identify matching trades securely.

The applicability of MPC for securely operating dark pools has been shown previously [40, 60, 41, 16]. Although MPC is befitting to the addressed problem, the current solutions are far from complete. All the proposed protocols only offer malicious security with abort. This could cause denial-of-service attacks and result in the protocol terminating even before the matched trades are disclosed. Further, such a setting allows an adversary to cause repeated failures. Since time is of essence in applications such as dark pools, this not only results in the wastage of valuable compute resources but may also hamper the functionality of the system. Hence, any security notion that empowers the adversary to abort does not fit the bill. Instead, a security notion that guarantees the delivery of output regardless of the adversary's misbehaviour is desirable. Hence it is imperative to realize robust, secure dark pools. However, observe that an MPC-based solution for a dark pool that achieves GOD by relying on a TTP, is equivalent to having a central broker who learns all the inputs and is trusted to perform the matching. This defeats the purpose of employing an MPC protocol, as one of the goals of a secure dark pool system is to hide the trade from *every single party* since it contains highly sensitive information of the client. Moreover, as described earlier, existing lawsuits against dark pool operators showcase the temptation to misuse profitable information. Hence, departing from the traditional approach, we design $(1, 1)$-FaF secure 5PC solutions for the same.

1.1.2.2 Secure allegation escrow systems

Another application where FaF-security is desirable is in the allegation escrow system. An allegation escrow is a system that allows victims of crimes such as abuse, corruption, exploitation, harassment, etc., to file a confidential report about the crime so that necessary actions can be taken to provide justice to the victim. For instance, institutions are mandated to appoint an organizational ombudsperson or a Chief Vigilance Officer (CVO) responsible for the prevention, detection, and punishment of malpractices. The victims are expected to report the inflicted crime to the CVO. Observe that the report contains the identities of the accused and victim, details of the inflicted crime, etc. Hence, it is regarded to be highly sensitive. The profound harm that can be inflicted on victims if the CVO leaks this sensitive data to the perpetrator, which is likely when the latter is a person of influence, instils great fear in victims and prevents many from coming forward. Thus, such a system requires the victims to place enormous trust in the integrity of the CVO. Instead, a secure platform for reporting crimes is a more reliable solution. Further, the victims may be more comfortable reporting the crime to a digital platform rather than to a human counterpart [113], and such a platform is more accessible and scalable. Thus, we aim to design a secure allegation escrow system that empowers victims to securely report allegations and seek justice.

Desirable properties of secure allegation escrow Having a system that merely records allegations and reveals them to the concerned authorities (for further action) may not suffice. Instead, the system should reveal allegations to the concerned authorities only when a sufficient number of allegations are recorded against a common perpetrator. This is because victims often find it effective and comforting to come out as a group. Further, acting against a common perpetrator in unison reduces the fear of retribution discussed previously. Some noteworthy examples of acting in unison are the #MeToo movement [1], and Project Callisto [198], which was deployed to help report sexual assaults on university campuses. To facilitate the reporting and processing of *collective* allegations, an allegation escrow system should have the following properties– (i) each victim must be able to independently file an allegation against a perpetrator, (ii) the system must be capable of matching allegations filed against a common perpetrator, (iii) these matched allegations should be revealed to the concerned authorities only once a predetermined condition for disclosure is met (e.g., Project Callisto requires at least two allegations against the same perpetrator before these can be revealed), (iv) the identity of the accuser, accused, and the details of the allegation must remain hidden until the allegation is revealed as a part of a collection. Additionally, a centralized solution (i.e., one escrow) for

the same is a misfit since it forms a single point of failure. Hence, similar to the CVO-based solution, one may compromise the escrow and learn the sensitive allegation data. Thus, it is desirable to have several independent escrows which collectively effectuate a secure allegation escrow (SAE) system with the above-mentioned properties and guarantee that *none* of the escrows can individually learn allegations on clear.

The condition for disclosure is one of the most crucial features of an SAE system. It defines the system's sensitivity towards handling an alleger's discomfort. This condition is calibrated using a parameter called *reveal threshold*. The parameter captures the minimum size of the unison the alleger wishes to be a part of (excluding the alleger) when its allegation is revealed in clear to the concerned authorities. In the literature, the reveal threshold has evolved from being a parameter that is globally fixed (i.e., common to all allegers) and public (i.e., known on clear to all the escrows) to an alleger-defined (variable) public parameter. Project Callisto [198] uses a globally-fixed public reveal threshold of one. The work of [144] extends support for a reveal threshold of more than one, yet it is globally fixed and public as before. However, *not every* alleger may be comfortable in coming out against a common perpetrator with just one other alleger (or even a system-defined threshold number of allegers). That is, setting a low (high) system-defined threshold will not allow the participation of victims who prefer more (few) supporters, making the system non-inclusive. The work of [14] that forms state of the art recognizes this pressing requirement and allows an alleger the flexibility of deciding its reveal threshold. Elaborately, each alleger can decide a reveal threshold, $\mathtt{t}$, for its allegation, which indicates that the allegation can be revealed if there exist at least $\mathtt{t}$ other *matching* allegations (i.e., those that allege the same perpetrator) which can be revealed. Thus, a subset $\mathcal{S}$ of matching allegations can be revealed if and only if the threshold of each allegation in $\mathcal{S}$ is $< |\mathcal{S}|$ (size of $\mathcal{S}$). This is referred to as the *reveal criteria* of the set $\mathcal{S}$. For example, if there exists a set of matching allegations with reveal thresholds $2, 3, 3, 4$, then no allegation is revealed because there does not exist any subset $\mathcal{S}$ of allegations that satisfies the reveal criteria. However, if another matching allegation with a reveal threshold of 3 is filed in the system, all the allegations with thresholds $2, 3, 3, 3, 4$ can be revealed. The system must thus allow secure identification of such a set of revealable matching allegations. We note that allowing a variable threshold is not the end of the road. Although [14] provides this key feature, it fails to do so while guaranteeing complete privacy to the victims. We explain our concern with one example below. Consider the scenario described above when a system has matching allegations with reveal thresholds $2, 3, 3, 4$, none of which can be revealed. Observe that if an alleger among these files another copy of its allegation, [14] treats the copy as a new allegation. Thus, these set of 5 allegations will satisfy the reveal criteria despite having an insufficient number of *distinct* allegers. Note

that this results in prematurely revealing the genuine (unique) allegations and compromises the privacy of the allegers. Similarly, there arise other privacy issues owing to the reveal threshold being public, which are detailed later (§6.2.1).

The user-defined reveal threshold captures the vulnerability of an alleger, and hence, it must be regarded as highly sensitive information. Thus, we develop the *first* SAE system that offers not only a flexible user-defined threshold, but also guarantees to keep thresholds private, and thereby arrests all concerns raised above. Additionally, we consider the possibility of allegation modification and deletion. Our MPC-based SAE system is realized via a set of (untrusted) escrows (acting as parties inside the MPC) who carry out the necessary computations of SAE via an MPC protocol on the submitted allegations. MPC guarantees the privacy of computation so that nothing beyond allowed outcomes of the SAE system (a bunch of matched allegations when reveal criteria are met) is leaked. Moreover, due to the drawbacks present in traditional MPC, which cannot be tolerated by a system such as that of SAE, our goal is to design $(1, 1)$-FaF secure solutions for the same in the 5PC setting.

1.1.3 Secure computation with a constant number of parties

The applications considered so far demanded to be operated with a specific number of parties to attain the desired level of security and efficiency. For instance, while 3PC provides the threshold-optimal setting in the honest majority and suffices for applications such as anonymous broadcast, 4PC is known to outperform 3PC in terms of efficiency and hence is beneficial for compute-intensive applications such as GCNs. Further, for applications such as dark pools and SAE, which deal with highly sensitive user inputs, it is desirable to provide FaF security where protocols are designed in the 5PC setting tolerating at most 1 malicious and at most 1 semi-honest corruption. In this way, the number of parties to be chosen to securely realize an application may vary across the applications. Hence, we next provide a generalization which allows instantiating the MPC protocol with an arbitrary (constant) number of parties (n) that can tolerate up to $t < n/2$ corruptions. Our generalized protocols continue to operate in the honest majority setting to capitalize on the efficiency benefits that this setting provides over the dishonest majority, which thereby facilitates attaining an efficient solution for the end application. In such a *multiparty* setting, for a higher n, the corruption threshold t is also higher. Hence, this setting is a better fit for applications that demand resiliency to a higher number of corruptions. The higher tolerance to corruption also increases trust in the system. Moreover, the multiparty setting allows performing privacy-preserving computations even in a non-outsourced deployment scenario when outsourcing the computation is not fea-

sible/preferable. Such scenarios arise a lot in practice, such as when multiple (more than 5) healthcare institutions want to pool their databases on genome sequences and subsequently perform analysis on their combined database among themselves. Hence, switching gears, we focus on designing efficient protocols in the multiparty computation setting having an honest majority. In contrast to preceding applications wherein the MPC was tailor-made to the specific application at hand, our multiparty protocols are generic. We also design a wide range of building blocks that facilitate the secure realization of various applications, including but not limited to genome sequence matching, biometric matching, and even deep neural networks.

1.2 Summary of the results in the thesis

Consequent to the above discussion on the applications, we now summarize our contributions.

1. **Secure local clustering:** We design solutions for privacy-preserving local clustering, which is addressed for the first time in the literature. Our local clustering algorithm is based on the heat kernel PageRank (HKPR) metric, which produces the best-known cluster quality. En route to our final solution, we have two important steps: (i) designing the data-oblivious equivalent of the state-of-the-art algorithms for computing local clustering and HKPR values (a data-oblivious algorithm is one whose control flow does not depend on the input data), and (ii) compiling the data-oblivious algorithms into its secure realisation via an MPC framework that supports operations over fixed-point arithmetic representation such as multiplication and division. Keeping efficiency in mind for large graphs, we choose the best-known honest majority 3PC framework of [136] and enhance it with some of the necessary yet missing primitives before using it for our purpose. Moreover, we operate in the GraphSC paradigm [13] to design efficient solutions. This additionally entails designing a secure shuffle protocol in the 3PC setting. We benchmark the performance of our secure protocols, and the reported run time showcases the practicality of the same. Further, we perform extensive experiments to evaluate the accuracy loss of our protocols. Compared to their cleartext counterparts, we observe that the results are comparable and, thus, showcase the practicality of the designed protocols.

   Anonymous broadcast: Given our improved secure shuffle protocol, we showcase its use for the application of anonymous broadcast where shuffle forms an integral part. In the process, we also provide improved properties of robustness and censorship resistance, which were missing in the state-of-the-art anonymous broadcast system of [80]. Our benchmarks showcase that our solution outperforms [80] in every aspect.

2. **Secure graph convolutional networks:** We design Entrada, a framework for securely evaluating GCNs. For efficiency and accuracy reasons, Entrada builds over the 4PC framework of [138] and enhances the same by providing the necessary primitives. Moreover, Entrada leverages the GraphSC paradigm of [13] to further enhance efficiency. This entails designing a secure and efficient shuffle protocol, specifically in 4PC, such that the random permutation used for shuffling is not leaked to anyone. This, to the best of our knowledge, is done for the first time and may be of independent interest. Through extensive experiments, we showcase that the accuracy of secure GCN evaluated via Entrada is on par with its cleartext counterpart. We also benchmark the efficiency of Entrada and showcase its practicality by benchmarking the GCN-based fraud detection application.

3. **Secure dark pools:** Given the shortcomings of traditional MPC, the work of [5] defined a Friends-and-Foes (FaF) security notion to address the same. We showcase the need for FaF security in applications such as dark pools. This subsequently necessitates designing concretely efficient FaF-secure protocols. Towards this, keeping efficiency at the centre stage, we design FaF-secure MPC protocols in the small-party honest-majority setting that operate on the ring algebraic structure. Specifically, we provide (1,1)-FaF secure 5PC protocols that consider one malicious and one semi-honest corruption and constitutes the optimal setting for attaining an honest majority. To facilitate having FaF-secure variants for the applications such as dark pools, we design a variety of building blocks optimized for our FaF setting. The practicality of the designed framework is showcased by benchmarking dark pools. In the process, we also improve the efficiency and security of the dark pool protocols over the existing traditionally secure ones provided in [40].

4. **Secure allegation escrow system:** In the work of [14], which presents the state-of-the-art solution, we identify attacks that can leak sensitive information and compromise victim privacy. We also report issues present in prior works that were left unidentified. To arrest all these breaches, we put forth an SAE system, Shield, that prevents the identified attacks and retains the salient features from all prior works. At the heart of our system lies a new duplicity check protocol and an improved matching protocol. We also provide additional features such as allegation modification and deletion, which were absent in the state of the art. To demonstrate feasibility, we benchmark the proposed system with our FaF-secure protocols, and the reported values showcase the practicality of our solution.

5. **Secure multiparty computation:** To address privacy concerns and give practical solutions, recent literature on the ring algebraic structure has mostly focused on the small-

party honest majority setting tolerating a single corruption, noting efficiency concerns. We extend the strategies to support higher resiliency in an honest majority setting keeping the efficiency of the online phase at the centre stage. Our semi-honest protocol improves over the state-of-the-art protocol in [62, 29] in terms of online communication without inflating the overall communication. It also allows shutting down almost half of the parties in the online phase, thereby saving up to 50% in the system's operational costs. Our maliciously secure protocol also enjoys similar benefits and requires only half of the parties, except for one-time verification towards the end, and provides security with fairness. Finally, we showcase the practicality of the designed protocols by benchmarking popular applications of genome sequence matching via edit distance, biometric matching via Euclidean distance, and deep neural networks.

1.3 Organization of the thesis

The thesis is structured into multiple chapters, each dedicated to an application outlined previously, with the penultimate chapter discussing the protocols for the multiparty setting. We finally conclude with the open questions. The chapters are organized as follows.

- Chapter 3: Secure local clustering
- Chapter 4: Secure graph convolutional networks
- Chapter 5: Secure dark pools
- Chapter 6: Secure allegation escrow system
- Chapter 7: Secure computation with a constant number of parties
- Chapter 8: Conclusion and open questions

Chapter 2

Preliminaries

This chapter discusses some of the common prerequisites to be used in the subsequent chapters. This includes notation, definitions, security model, and overview of the 3PC of SWIFT [136], the 4PC of Tetrad [138], and the GraphSC paradigm [179, 13]. The application-specific prerequisites are discussed within the respective chapters.

2.1 System model and primitives

System and threat model We let $\mathcal{P}$ denote the set of parties that carry out the computations in the MPC protocols. We assume that these parties are connected by pair-wise private and authentic channels in a synchronous network. There exists a probabilistic polynomial time adversary $\mathcal{A}$ that corrupts some of the parties in $\mathcal{P}$, where the corruption threshold (i.e., the number of parties corrupted by $\mathcal{A}$), as well as the type of corruption (i.e., semi-honest or malicious), varies depending on the application under consideration and is discussed within the specific chapter. Our protocols are cast in the preprocessing paradigm, which comprises an *input-independent* preprocessing phase and an *input-dependent* online phase. The protocols are designed to work over an ℓ-bit ring denoted by $\mathbb{Z}_{2^\ell}$. We consider both, the arithmetic ring $\mathbb{Z}_{2^\ell}$, as well as the Boolean ring $\mathbb{Z}_{2^1}$. For a bit $\mathsf{b} \in \{0, 1\}$, we use b^{R} to denote the representation of the bit b over the arithmetic ring $\mathbb{Z}_{2^\ell}$. Specifically, b^{R} has the least significant bit set to b, with all other bits being 0.

Dealing with decimal values For applications where the inputs are decimal numbers, we use the fixed-point arithmetic (FPA) [173, 49, 193, 50, 38, 138, 136, 220, 61] representation to encode the value in the underlying ring $\mathbb{Z}_{2^\ell}$. A decimal value is represented as an ℓ-bit number in signed 2's complement notation, where the most significant bit (msb) is the sign bit,

f least significant bits denote the fractional part, and the input is k bits long. Unless otherwise specified, we set $\ell = 64$, $\mathsf{k} = 32$ and $\mathsf{f} = 16$. Operations are then performed on the ℓ-bit integer, treated as an element of $\mathbb{Z}_{2^\ell}$, modulo 2^ℓ. We use $(\mathsf{x})_\mathsf{f}$ to denote that $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ has f bit precision.

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

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

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 x given just its commitment $\mathsf{Com}(x)$, while the latter prevents a corrupt party from opening the commitment to a different value $x' \neq x$. The practical realization of a commitment scheme is via a hash function $\mathsf{H}(\cdot)$ given below, whose security can be proved in the random-oracle model (ROM)—for $(c, o) = (\mathsf{H}(x||r), x||r) = \mathsf{Com}(x; r)$.

Threshold secret sharing A t-out-of-n secret sharing scheme allows a dealer with a secret v to distribute shares of v among a set of n parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ such that—(i) no subset of t shares provides any information about the underlying secret v, (ii) any set of $t+1$ shares or more allows successful reconstruction of v. Replicated secret sharing (RSS) [114] is one instantiation of threshold secret sharing which works as follows. A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be RSS-shared with threshold t if for every subset $\mathcal{T} \subset \mathcal{P}$ of $n-t$ parties there exists a share $\mathsf{v}_\mathcal{T} \in \mathbb{Z}_{2^\ell}$ possessed by all $P_i \in \mathcal{T}$ such that $\mathsf{v} = \sum_\mathcal{T} \mathsf{v}_\mathcal{T}$. Observe that RSS satisfies condition (i) stated above since every set of t parties will miss one share of v, and it satisfies condition (ii) since every set of $t+1$ parties possesses all the shares of v.

2.2 Security model

We prove the security of our protocols using the real-world/ ideal-word simulation paradigm [88, 155]. Informally, here the security is proved 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, where the

latter is considered to be secure by definition. In an ideal-world execution, there exists a trusted third party (TTP), to which the parties send their inputs over perfectly secure channels. The TTP carries out the computation and sends the output to the parties. Informally, a protocol is said to be secure if whatever an adversary can do in the real-world execution, it can also be done in the ideal-world execution. We refer the readers to [88, 155] for further details regarding the security model.

Let $\mathcal{A}$ denote the probabilistic polynomial time real-world adversary corrupting at most t out of n parties in $\mathcal{P}$, let $\mathcal{S}$ denote the corresponding ideal-world adversary, and let $\mathcal{F}$ denote the ideal functionality. Let $\text{VIEW}^{\text{IDEAL}}_{\mathcal{S},\mathcal{F}}(1^\kappa, z_\mathcal{A})$ be the malicious adversary's simulated view and $\text{OUT}^{\text{IDEAL}}_{\mathcal{S},\mathcal{F}}(1^\kappa, z_\mathcal{A})$ denote the output of the uncorrupted parties during a random execution of ideal-world functionality $\mathcal{F}$ with respect to the security parameter κ and auxiliary input $z_\mathcal{A}$. Similarly, let $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_\mathcal{A})$ be $\mathcal{A}$'s view and $\text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_\mathcal{A})$ denote the output of the uncorrupted parties during a random execution of a protocol Π. 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 the joint distribution of the view of $\mathcal{S}$ and the output of the honest parties from the ideal world execution, and the joint distribution of the view of $\mathcal{A}$ and the output of the honest parties from the real-world execution, are computationally indistinguishable.

Definition 2.1 *For $n \in \mathbb{N}$, let $\mathcal{F}$ be an ideal-world functionality and let Π be the real-world n-party protocol that operates over private and authenticated point-to-point channels among the parties. 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 subset of parties as $\mathcal{A}$, such that the real-world view and the ideal-world view is indistinguishable, i.e.,*

$$\{\text{VIEW}^{\text{IDEAL}}_{\mathcal{S},\mathcal{F}}(1^\kappa, z_\mathcal{A}), \text{OUT}^{\text{IDEAL}}_{\mathcal{S},\mathcal{F}}(1^\kappa, z_\mathcal{A})\} \overset{c}{\approx} \{\text{VIEW}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_\mathcal{A}), \text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_\mathcal{A})\}$$

We analyze the security guarantees of correctness and privacy separately in all our security proofs since we consider deterministic functionalities alone in this thesis [155].

Security notions We next describe the security notions that a protocol can attain (considered in this thesis) and the corresponding ideal functionalities for the same.

Security with fairness: Informally, a protocol that attains the security notion of fairness ensures that either all parties learn the output of the computation or none do. This is captured by the ideal functionality in Fig. 2.1 when considering $\mathcal{P} = \{P_1, \ldots, P_n\}$.

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

$\mathcal{F}_{\mathsf{Fair}}$ interacts with the parties in $\mathcal{P} = \{P_1, \ldots, P_n\}$, and the adversary $\mathcal{S}$ that corrupts a subset of parties in $\mathcal{P}$. Let f denote the function to be computed. Let x_s be the input of party $P_s \in \mathcal{P}$, and y_s be the corresponding output, i.e $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$. Corrupted parties may send arbitrary inputs as instructed by $\mathcal{S}$. Further, $\mathcal{S}$ is allowed to send a special command, `abort`, which indicates that none of the parties should receive the output.

Step 1: $\mathcal{F}_{\mathsf{Fair}}$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$. If $(\mathsf{Input}, *)$ is already received from P_s, then ignore the current message. Otherwise, record $\mathsf{x}'_s = \mathsf{x}_s$ internally. If x'_s lies outside the input domain of P_s, then record $\mathsf{x}'_s = \mathtt{abort}$.

Step 2: If there exists $s \in \{1, \ldots, n\}$ such that $\mathsf{x}'_s = \mathtt{abort}$, then set $\mathsf{y}_s = \mathtt{abort}$ for $s \in \{1, \ldots, n\}$. Else, compute $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$.

Step 3: Send $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$. Here, $\mathsf{y}_s = \mathtt{abort}$ if $\mathcal{S}$ sent $(\mathsf{Signal}, \mathtt{abort})$.

Figure 2.1: Ideal functionality for evaluating function f with fairness.

Security with guaranteed output delivery (GOD): Informally, a protocol that attains the security notion of guaranteed output delivery (or robustness) guarantees that regardless of the adversary's misbehaviour, all honest parties learn the output of the computation. This is captured by the ideal functionality in Fig. 2.2.

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

$\mathcal{F}_{\mathsf{GOD}}$ interacts with the parties in $\mathcal{P} = \{P_1, \ldots, P_n\}$, and the adversary $\mathcal{S}$ that corrupts a subset of parties in $\mathcal{P}$. Let f denote the function to be computed. Let x_s be the input of party $P_s \in \mathcal{P}$, and y_s be the corresponding output, i.e $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$. Corrupted parties may send arbitrary inputs as instructed by $\mathcal{S}$.

Step 1: $\mathcal{F}_{\mathsf{GOD}}$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$. If $(\mathsf{Input}, *)$ is already received from P_s, then ignore the current message. Otherwise, record $\mathsf{x}'_s = \mathsf{x}_s$ internally. If x'_s lies outside the input domain of P_s, then record x'_s as some predetermined default value.

Step 2: Compute $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$ and send $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$.

Figure 2.2: Ideal functionality for evaluating function f with GOD.

Friends-and-foes (FaF) security model [5] In the traditional definition, as discussed above, security is captured by designing an ideal-world adversary (simulator) which can simulate the view of the real-world adversary corrupting a subset of the parties in $\mathcal{P}$. However, the FaF security model assumes the presence of decentralized adversaries—a malicious $\mathcal{A}$ that corrupts t out of the n parties maliciously, and a semi-honest adversary $\mathcal{A}_{\mathcal{H}}$ that corrupts $h^\star$ out of the

remaining $n-t$ parties in a semi-honest manner. Hence, in the FaF-security model, there exists the additional requirement of simulating the view of the semi-honest parties. This necessitates the use of two simulators. Thus, to prove the security, two simulators are constructed in the ideal-world execution, one for the malicious adversary and one for the semi-honest adversary. Further, the malicious adversary is allowed to send its entire view to the semi-honest adversary in the ideal-world execution (to capture the behaviour where the malicious adversary may send non-protocol messages to uncorrupted parties in the real-world execution).

Let $\mathcal{A}$ denote the PPT real-world malicious adversary corrupting t parties in $\mathcal{I} \subset \mathcal{P}$, and $\mathcal{S}_{\mathcal{A}}$ denote the corresponding ideal-world simulator. Similarly, let $\mathcal{A}_{\mathcal{H}}$ denote the PPT real-world semi-honest adversary corrupting $h^\star$ parties in $\mathcal{H} \subset \mathcal{P} \backslash \mathcal{I}$, and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$, be the ideal-world simulator. Let $\mathcal{F}$ be the ideal-world functionality. Let $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_{\mathcal{A}})$ be $\mathcal{A}$'s view and $\text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_{\mathcal{A}})$ denote the output of the uncorrupted parties (in $\mathcal{P} \setminus \mathcal{I}$) during a random execution of a protocol Π, where $z_{\mathcal{A}}$ is $\mathcal{A}$'s auxiliary input. Correspondingly, let $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\Pi}(1^\kappa, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}})$ be $\mathcal{A}_{\mathcal{H}}$'s view during an execution of protocol Π running alongside $\mathcal{A}$, where $z_{\mathcal{A}_{\mathcal{H}}}$ is the $\mathcal{A}_{\mathcal{H}}$'s auxiliary input. Note that $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\Pi}(1^\kappa, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}})$ consists of the non-protocol messages sent by the $\mathcal{A}$ to $\mathcal{A}_{\mathcal{H}}$. Similarly, let $\text{VIEW}^{\text{IDEAL}}_{\mathcal{S}_{\mathcal{A}},\mathcal{F}}(1^\kappa, z_{\mathcal{A}})$ be the malicious adversary's simulated view and $\text{OUT}^{\text{IDEAL}}_{\mathcal{S}_{\mathcal{A}},\mathcal{F}}(1^\kappa, z_{\mathcal{A}})$ denote the output of the uncorrupted parties during a random execution of ideal-world functionality $\mathcal{F}$. Further, let $\text{VIEW}^{\text{IDEAL}}_{\mathcal{S}_{\mathcal{A}},\mathcal{S}_{\mathcal{A},\mathcal{H}},\mathcal{F}}(1^\kappa, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}})$ be the semi-honest adversary's simulated view during an execution of $\mathcal{F}$ running alongside $\mathcal{A}$.

A protocol Π is said to compute $\mathcal{F}$ with (weak) computational $(t, h^\star)$-FaF security if

$$\{\text{VIEW}^{\text{IDEAL}}_{\mathcal{S}_{\mathcal{A}},\mathcal{F}}(1^\kappa, z_{\mathcal{A}}), \text{OUT}^{\text{IDEAL}}_{\mathcal{S}_{\mathcal{A}},\mathcal{F}}(1^\kappa, z_{\mathcal{A}})\} \overset{c}{\approx} \{\text{VIEW}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_{\mathcal{A}}), \text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_{\mathcal{A}})\}$$
$$\{\text{VIEW}^{\text{IDEAL}}_{\mathcal{S}_{\mathcal{A}},\mathcal{S}_{\mathcal{A},\mathcal{H}},\mathcal{F}}(1^\kappa, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}}), \text{OUT}^{\text{IDEAL}}_{\mathcal{S}_{\mathcal{A}},\mathcal{F}}(1^\kappa, z_{\mathcal{A}})\} \overset{c}{\approx} \{\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\Pi}(1^\kappa, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}}), \text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\kappa, z_{\mathcal{A}})\}$$

2.3 3PC of SWIFT

SWIFT [136] is a robust 3-party computation (3PC) framework with $\mathcal{P} = \{P_0, P_1, P_2\}$ (or equivalently $\mathcal{P} = \{P_i, P_j, P_k\}$) tolerating at most 1 malicious corruption. We first explain the 3PC sharing semantics of SWIFT, followed by its protocols that the thesis relies on.

Sharing semantics SWIFT performs computation via masked evaluation and relies on a variant of replicated secret sharing (RSS) among 3 parties with threshold 1. The following are the sharing semantics used therein.

– $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be (3, 1) replicated secret shared (RSS) or $[\cdot]$-shared, if there exists $[\mathsf{v}]_{01}, [\mathsf{v}]_{02}, [\mathsf{v}]_{12} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = [\mathsf{v}]_{01} + [\mathsf{v}]_{02} + [\mathsf{v}]_{12}$, and each $[\mathsf{v}]_{ij} \in \{[\mathsf{v}]_{01}, [\mathsf{v}]_{02},$

$[\mathsf{v}]_{12}\}$ is held by $P_i, P_j \in \mathcal{P}$.

– $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}$, if there exists $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared, and there exists $\beta_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ such that $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ which is held by all parties in $\mathcal{P}$.

Linear operations such as $[\![\mathsf{c}_1\mathsf{x} + \mathsf{c}_2\mathsf{y}]\!]$ given $[\![\mathsf{x}]\!], [\![\mathsf{y}]\!]$ where $\mathsf{c}_1, \mathsf{c}_2 \in \mathbb{Z}_{2^\ell}$ are public constants can be performed non-interactively by computing $\mathsf{c}_1[\![\mathsf{x}]\!] + \mathsf{c}_2[\![\mathsf{y}]\!]$. The linearity property extends to $[\cdot]$-sharing as well. Sharing over $\mathbb{Z}_{2^\ell}$ is referred to as arithmetic sharing ($[\![\cdot]\!]$) while over $\mathbb{Z}_2$ is referred to as Boolean sharing ($[\![\cdot]\!]^{\mathbf{B}}$), which is similar to $[\![\cdot]\!]$ except that addition operation is replaced with XOR. In general, the Boolean world is analogous to the arithmetic world, with arithmetic addition and multiplication operations being replaced with Boolean XOR and AND.

Shared key setup Parties can non-interactively generate common random values among themselves by relying on the common PRF keys established among themselves during a one-time setup phase. This is abstracted via the ideal functionality $\mathcal{F}_{\mathsf{Setup}}$ (Fig. 2.3). Several works [11, 12, 173, 49, 193, 30, 50, 38, 136] rely on such a setup. Thus, the computation starts with such a setup phase which is done once and for all. Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function (PRF), with co-domain X being $\mathbb{Z}_{2^\ell}$. The set of keys established between the parties for 3PC is as follows:

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

– One shared key known to all the parties—$k_\mathcal{P}$.

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

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

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

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

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

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 2.3: Ideal functionality for shared-key setup in SWIFT [136].

Suppose P_0, P_1 wish to sample a random value $r \in \mathbb{Z}_{2^\ell}$ non-interactively. To do so they invoke $F_{k_{01}}(id_{01})$ and obtain r. Here, id_{01} denotes a counter maintained by the parties and is updated after every PRF invocation. When the appropriate key used to sample the random

values is implicit from the context, from the identities of the pair that sample or from the fact that it is sampled by all, the key and the counter will be omitted from the description.

Non-interactively generating $[\cdot]$-shares of a common $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ held by $P_l, P_m \in \mathcal{P}$ To generate $[\mathsf{v}]$, parties need to define three shares $[\mathsf{v}]_{01}, [\mathsf{v}]_{02}, [\mathsf{v}]_{12} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = [\mathsf{v}]_{01} + [\mathsf{v}]_{02} + [\mathsf{v}]_{12}$, where each $[\mathsf{v}]_{ij}$ is held by parties $P_i, P_j \in \mathcal{P}$. Observe that this can be done non-interactively by setting the share $[\mathsf{v}]_{lm} = \mathsf{v}$ and the other two $[\cdot]$-shares of v as 0.

Joint message passing (Jmp) primitive Protocols in SWIFT heavily rely on the joint message passing primitive to ensure robust computation. This primitive allows two parties to deliver a common message to a third party, where one sender sends the message while the other sends its hash to the receiver. In the process, either the recipient receives the correct message or, if there is an inconsistency in the received messages, parties instead proceed to identify a trusted third party (TTP)[1]. The TTP is then responsible for performing the required computation on clear and guarantee delivery of output. Several works [136, 138, 61] rely on this primitive or its variation to ensure GOD. We let "P_i, P_j $\mathsf{Jmp\text{-}send}$ v to P_k" denote invocation of Jmp with P_i, P_j as senders, P_k as receiver, and v being the message to be sent.

The robust 3PC protocol for Π_{Jmp} appears in Figure 2.4. The instantiation of Π_{Jmp} 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 is executed only once, at the end of the computation. The protocol also relies on a collision-resistant hash function $\mathsf{H}(\cdot)$. We refer the reader to [136, 61] for further details of Jmp.

We note that at any point during the run of the protocol, if the invocation of Π_{Jmp} results in identifying a TTP, parties do not execute the rest of the protocol steps, and the remainder computation is carried out by the TTP who guarantees the delivery of the correct function output to all parties. Elaborately, on identifying a TTP, all parties send their inputs on clear to the TTP, who evaluates the necessary function on these clear inputs. It then sends the computed output to all the parties. In this way, reliance on Π_{Jmp} to send protocol messages ensures that if any malicious party misbehaves to prevent the parties from obtaining the output, a TTP is identified, and the output is now obtained via the TTP. We remark that most protocols in the 3-party setting that guarantee output delivery follow the TTP-based approach [30, 36, 136].

[1] In the case of fair protocol, parties `abort`.

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

Send Phase: P_i sends $\mathsf{msg}_i = \mathsf{v}$ to P_k.
Verify Phase: P_j sends $\mathsf{msg}_j = \mathsf{H}(\mathsf{v})$ to P_k, who checks if the hash is consistent with the value sent by P_i. If the values are not consistent, parties proceed as follows to identify a TTP.

– P_k broadcasts $(Accuse, P_i, P_j, \mathsf{msg}_i, \mathsf{msg}_j)$

- If $\mathsf{H}(\mathsf{msg}_i) = \mathsf{msg}_j$ parties set $\mathsf{TTP} = P_i$
- If msg_i is different from the value sent by P_i, then P_i broadcasts $(Accuse, P_k)$ and parties set $\mathsf{TTP} = P_j$.
- Similarly if msg_j is different from the value sent by P_j, then P_j broadcasts $(Accuse, P_k)$ and parties set $\mathsf{TTP} = P_i$.
- If both parties P_i and P_j broadcasts $(Accuse, P_k)$ and parties set $\mathsf{TTP} = P_i$.
- If none of the parties P_i or P_j accuse, then parties set $\mathsf{TTP} = P_k$.

Figure 2.4: Joint message passing in 3PC.

Input sharing and consistency check Consider the outsourced setting, where a set of three servers are hired to carry out the 3PC. To enable the client to generate $[\![\mathsf{v}]\!]$ of its input $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ towards the servers, the input sharing protocol proceeds as follows. During the preprocessing phase, the servers generate $[\cdot]$-shares for a random $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\ell}$, non-interactively. Observe that each of $[\alpha_\mathsf{v}]_{01}, [\alpha_\mathsf{v}]_{02}, [\alpha_\mathsf{v}]_{12}$ is held by exactly two servers, at most one of which can be maliciously corrupt. Making $P_i, P_j \in \mathcal{P}$ send $[\alpha_\mathsf{v}]_{ij}$ to the client may lead to uncertainty at the client if P_i, P_j send different versions of $[\alpha_\mathsf{v}]_{ij}$ to it. Hence, to ensure the correct delivery of each $[\alpha_\mathsf{v}]_{ij}$ towards the client, servers rely on a commitment scheme. Elaborately, each pair of servers P_i, P_j generate a commitment $\mathsf{Com}([\alpha_\mathsf{v}]_{ij})$ of $[\alpha_\mathsf{v}]_{ij}$ and $\mathsf{Jmp\text{-}send}$ it to P_k. This ensures that all servers are in agreement with commitments generated for each $[\cdot]$-share of α_v.

During the online phase, once the client is ready to share its input, each server sends $\mathsf{Com}([\alpha_\mathsf{v}]_{01})$, $\mathsf{Com}([\alpha_\mathsf{v}]_{02})$ and $\mathsf{Com}([\alpha_\mathsf{v}]_{12})$ to the client. The client retains the values in majority for each $\mathsf{Com}([\alpha_\mathsf{v}]_{ij})$. At the same time, each P_i, P_j send the opening to $\mathsf{Com}([\alpha_\mathsf{v}]_{ij})$ towards the client. The client accepts the opening, which is consistent with the commitment that was in the majority (the correct opening can be identified owing to the property of the commitment scheme, which outputs a $\perp$ for incorrect ones). In this way, the client receives correct $[\alpha_\mathsf{v}]_{01}, [\alpha_\mathsf{v}]_{02}, [\alpha_\mathsf{v}]_{12}$. Upon receiving these values, the client generates and sends $\beta_\mathsf{v} = \mathsf{v} + [\alpha_\mathsf{v}]_{01} + [\alpha_\mathsf{v}]_{02} + [\alpha_\mathsf{v}]_{12}$ to the servers. To ensure that each server receives the same β_v and guarantee that a client has not misbehaved, each server broadcasts the β_v received from the client. If there is a majority among the broadcast values, the client's message is accepted, and each server sets its β_v to be the majority value. Else, the client's message is deemed as malformed and discarded.

Output reconstruction To enable robust reconstruction of a $[\![\cdot]\!]$-shared value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ among the computing parties (or equivalently, the servers hired to carry out the 3PC), they proceed similar to as described above. During the preprocessing phase, in addition to generating $[\alpha_\mathsf{v}]$, parties also generate commitments on each of the $[\cdot]$-shares of α_v. This results in all parties being in agreement with commitments for each $[\cdot]$-share of α_v. Next, in the online phase to reconstruct v, observe that each party P_k misses the share $[\alpha_\mathsf{v}]_{ij}$ which is held by the other two parties $P_i, P_j \in \mathcal{P} \setminus P_k$. Hence, P_i, P_j send the opening of $\mathsf{Com}([\alpha_\mathsf{v}]_{ij})$ to P_k. Since at most one party among P_i, P_j can be malicious, even if the malicious party sends an incorrect opening, P_k is guaranteed to receive the correct opening from the honest party (the correct opening can be identified owing to the property of the commitment scheme which outputs a $\perp$ for incorrect ones). Party P_k uses the correct opening to obtain the missing share $[\alpha_\mathsf{v}]_{ij}$ and reconstructs v as $\mathsf{v} = \beta_\mathsf{v} - [\alpha_\mathsf{v}]_{ij} - [\alpha_\mathsf{v}]_{ik} - [\alpha_\mathsf{v}]_{jk}$. Thus, reconstruction will not fail if a malicious party tries to disrupt it by sending an incorrect message, resulting in robust reconstruction.

Protocols from SWIFT The protocols from SWIFT that the thesis relies on, other than the ones described above, are listed in Table 2.1.

Building block	Notation	Description
Joint sharing	$[\![\mathsf{v}]\!] = \Pi_{\mathsf{JSh}}(P_i, P_j, \mathsf{v})$	Enables $P_i, P_j \in \mathcal{P}$ to generate $[\![\mathsf{v}]\!]$ where $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is held by P_i, P_j
Multiplication with truncation	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{Mul}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!], \mathsf{f})$	Multiplies x, y and outputs $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ by truncated by f bits
3-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{3\text{-}Mul}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 3 bits at once
4-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{4\text{-}Mul}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}}, [\![\mathsf{d}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 4 bits at once
Comparison	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} < \mathsf{y}$, else outputs $\mathsf{b} = 0$
Oblivious select	$[\![\mathsf{x_b}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{x_0}]\!], [\![\mathsf{x_1}]\!], [\![\mathsf{b}]\!]^{\mathbf{B}})$	Obliviously selects $\mathsf{x_b}$ among $\mathsf{x_0}, \mathsf{x_1}$
Bit2A	$[\![\mathsf{b}]\!] = \Pi_{\mathsf{Bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	Converts bit to its arithmetic equivalent
Arithmetic to Boolean	$[\![\mathsf{v}]\!]^{\mathbf{B}} = \Pi_{\mathsf{A2B}}([\![\mathsf{v}]\!])$	Converts arithmetic representation of a value to its Boolean equivalent
Boolean to arithmetic	$[\![\mathsf{v}]\!] = \Pi_{\mathsf{B2A}}([\![\mathsf{v}]\!]^{\mathbf{B}})$	Converts Boolean representation of a value to its arithmetic equivalent
Negation	$[\![\bar{\mathsf{b}}]\!]^{\mathbf{B}} = \Pi_{\mathsf{NOT}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	Outputs $\bar{\mathsf{b}} = 1 \oplus \mathsf{b}$ where $\mathsf{b} \in \mathbb{Z}_2$

Table 2.1: Description of other protocols from SWIFT [136].

On the security of the protocols from SWIFT While SWIFT provides the strongest security of robustness, we note that depending on the application scenario, one may choose the desired level of security. Specifically, robustness is attained by relying on a TTP to carry out the computation on the honest party's inputs (in the clear) if misbehaviour is detected. Hence, if the application under consideration cannot tolerate revealing the inputs to a TTP even though the TTP is known to be honest, the application can settle for the weaker security notion of fairness (which is stronger than abort security). The fair version of the protocols can

be derived from the robust version by making the following changes– (i) use of the fair version of Jmp instead of the robust version, (ii) terminating the protocol when a party `aborts` instead of proceeding with TTP identification and, (iii) relying on a fair reconstruction protocol. We remark that even for this weaker security notion of fairness, the protocols are on par with the robust protocols in terms of efficiency.

2.4 4PC of Tetrad

Tetrad [138] is a 4-party computation (4PC) framework with $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$ (or equivalently $\mathcal{P} = \{P_i, P_j, P_k, P_l\}$) tolerating at most 1 malicious corruption. We first explain the 4PC sharing semantics of Tetrad, followed by its protocols that the thesis relies on.

Sharing semantics Tetrad performs computation via masked evaluation and relies on a variant of replicated secret sharing (RSS) among 4 parties with threshold 1. The following are the sharing semantics used therein.

– $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be (3, 1) replicated secret shared (RSS) or $[\cdot]$-shared among parties in $\{P_1, P_2, P_3\}$, if there exists $\alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}2}, \alpha_{\mathsf{v}3}, \in \mathbb{Z}_{2^\ell}$ such that $\alpha_{\mathsf{v}} = \alpha_{\mathsf{v}1} + \alpha_{\mathsf{v}2} + \alpha_{\mathsf{v}3}$, distributed among the parties as follows: P_1 holds $\alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}3}$; P_2 holds $\alpha_{\mathsf{v}2}, \alpha_{\mathsf{v}3}$; P_3 holds $\alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}2}$.

– $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be $[\![\cdot]\!]$-shared if there exists mask $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ and masked value $\beta_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ is held by parties in $\{P_1, P_2, P_3\}$, and α_{v} is (3, 1) replicated secret-shared (or $[\cdot]$-shared) among parties in $\{P_1, P_2, P_3\}$, and all its shares are known on clear to P_0. Elaborately, there exist values $\alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}2}, \alpha_{\mathsf{v}3}, \in \mathbb{Z}_{2^\ell}$ such that $\alpha_{\mathsf{v}} = \alpha_{\mathsf{v}1} + \alpha_{\mathsf{v}2} + \alpha_{\mathsf{v}3}$, which together with β_{v} is distributed among the parties as follows: P_0 holds $\alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}2}, \alpha_{\mathsf{v}3}$; P_1 holds $\beta_{\mathsf{v}}, \alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}3}$; P_2 holds $\beta_{\mathsf{v}}, \alpha_{\mathsf{v}2}, \alpha_{\mathsf{v}3}$; and P_3 holds $\beta_{\mathsf{v}}, \alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}2}$.

As in the case of SWIFT, all the sharing schemes in Tetrad also satisfy the linearity property. Boolean secret-sharing over $\mathbb{Z}_2$, denoted as $[\![\cdot]\!]^{\mathbf{B}}$, is similar to $[\![\cdot]\!]$ except that the addition operation is replaced with XOR. In general, the Boolean world is analogous to the arithmetic world, with arithmetic addition and multiplication operations being replaced with Boolean XOR and AND. Arithmetic equivalent of a bit b in ring $\mathbb{Z}_{2^\ell}$ is denoted as b^{R}.

Shared key setup As in the case of SWIFT [136], Tetrad also enables the non-interactive generation of common random values among parties by relying on the common PRF keys established among the parties during a one-time setup phase. This is abstracted via the ideal functionality $\mathcal{F}_{\mathsf{Setup}}$ (Fig. 2.5). Elaborately, let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function (PRF), with $X = \mathbb{Z}_{2^\ell}$. The following keys are established between parties–

(i) k_{ij} for every pair of parties P_i, P_j, (ii) k_{ijk} for every triple of parties P_i, P_j, P_k, and (iii) $k_{\mathcal{P}}$ known to all parties in $\mathcal{P}$. If P_0, P_1 wish to sample a random value $r \in \mathbb{Z}_{2^\ell}$ non-interactively, they compute $F_{k_{01}}(id_{01})$ and obtain r. Here, id_{01} denotes a counter maintained by the parties and is updated after every PRF invocation.

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

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

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

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 2.5: Ideal functionality for shared-key setup in Tetrad [138].

Joint message passing Similar to SWIFT [136], protocols in Tetrad rely on a joint message passing primitive (Fig. 2.6) to guarantee robust computation. As before, this primitive allows two senders P_i, P_j to relay a common message, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, to a recipient P_k, either by ensuring successful delivery of v, or by establishing a conflicting pair of parties, one among which is guaranteed to be corrupt. This implies the residual two parties are honest, one of which is then entrusted to complete the computation by enacting the role of a trusted third party (TTP).

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

$P_s \in \mathcal{P}$ 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 2.6: Joint message passing in Tetrad.

Joint sharing Protocol Π_{JSh} enables parties P_i, P_j to generate $[\![\cdot]\!]$-shares of value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. During the preprocessing phase, shares of α_{v} are sampled such that both P_i, P_j will get the entire mask α_{v}. During the online phase, P_i, P_j compute and send $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to parties P_1, P_2, P_3 via Π_{Jmp}.

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

– (P_0, P_1): Parties in $\mathcal{P} \setminus \{P_2\}$ sample $\alpha_{\mathsf{v}1} \in \mathbb{Z}_{2^\ell}$. Parties set $\alpha_{\mathsf{v}2} = \beta_{\mathsf{v}} = 0$. P_0, P_1 send $\alpha_{\mathsf{v}3} = -\mathsf{v} - \alpha_{\mathsf{v}1}$ to P_2 via Π_{Jmp}.

– (P_0, P_2): Parties in $\mathcal{P} \setminus \{P_3\}$ sample $\alpha_{\mathsf{v}3} \in \mathbb{Z}_{2^\ell}$. Parties set $\alpha_{\mathsf{v}1} = \beta_{\mathsf{v}} = 0$. P_0, P_2 send $\alpha_{\mathsf{v}2} = -\mathsf{v} - \alpha_{\mathsf{v}3}$ to P_3 via Π_{Jmp}.

– (P_0, P_3): Parties in $\mathcal{P} \setminus \{P_1\}$ sample $\alpha_{\mathsf{v}2} \in \mathbb{Z}_{2^\ell}$. Parties set $\alpha_{\mathsf{v}3} = \beta_{\mathsf{v}} = 0$. P_0, P_3 send $\alpha_{\mathsf{v}1} = -\mathsf{v} - \alpha_{\mathsf{v}1}$ to P_1 via Π_{Jmp}.

Protocols from Tetrad The list of protocols that the thesis relies on from Tetrad [138], other than the ones described above, appears in Table 2.2.

Building block	**Notation**	**Description**
Multiplication	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{Mul}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!], \mathsf{f})$	Multiplies x, y and outputs $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ truncated by f bits
Matrix Multiplication	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{MatMul}}([\![\mathsf{A}]\!], [\![\mathsf{B}]\!], \mathsf{f})$	Multiplies matrices A, B and outputs $\mathsf{C} = \mathsf{A} \cdot \mathsf{B}$
3-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{3\text{-}Mul}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 3 inputs at once
4-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{4\text{-}Mul}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}}, [\![\mathsf{d}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 4 inputs at once
MulR	$[\mathsf{z}] = \Pi_{\mathsf{MulR}}([\mathsf{x}], [\mathsf{y}])$	Multiplies x, y and outputs $[\cdot]$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$
Arithmetic to Boolean	$[\![\mathsf{x}]\!]^{\mathbf{B}} = \Pi_{\mathsf{A2B}}([\![\mathsf{x}]\!])$	Converts arithmetic shares of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ to Boolean shares of each bit of x
Boolean to arithmetic	$[\![\mathsf{x}]\!] = \Pi_{\mathsf{B2A}}([\![\mathsf{x}]\!]^{\mathbf{B}})$	Converts Boolean shares of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ to arithmetic shares
Bit2A	$[\![\mathsf{b}]\!] = \Pi_{\mathsf{Bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	Converts bit to its arithmetic equivalent
Bit extraction	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Bitext}}([\![\mathsf{x}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} < 0$, else outputs $\mathsf{b} = 0$
Comparison	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} < \mathsf{y}$, else outputs $\mathsf{b} = 0$
Bit injection	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{BitInj}}([\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{x}]\!])$	Multiplies arithmetic equivalent $\mathsf{b}^{\mathsf{R}} \in \mathbb{Z}_{2^\ell}$ of $\mathsf{b} \in \mathbb{Z}_2$ with $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ and outputs $\mathsf{z} = \mathsf{b}^{\mathsf{R}} \cdot \mathsf{x}$
Oblivious select	$[\![\mathsf{x}_\mathsf{b}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{x}_0]\!], [\![\mathsf{x}_1]\!], [\![\mathsf{b}]\!]^{\mathbf{B}})$	Obliviously selects x_b among $\mathsf{x}_0, \mathsf{x}_1$
Negation	$[\![\bar{\mathsf{x}}]\!]^{\mathbf{B}} = \Pi_{\mathsf{NOT}}([\![\mathsf{x}]\!]^{\mathbf{B}})$	Outputs 1's complement of Boolean representation of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$

Table 2.2: Description of other protocols from Tetrad [138].

2.5 Secure shuffle

2.5.1 Random permutation

Let $\mathbb{N}$ denote the set of integers $\{1, 2, \ldots, \mathsf{N}\}$. A permutation is any bijective function $\pi : \mathbb{N} \to \mathbb{N}$. That is, a permutation π denotes a mapping of a rearrangement of the elements in $\mathbb{N}$. The set denoted as S_{N} consists of all possible (bijective functions) rearrangements of elements in $\mathbb{N}$ and hence comprises $\mathsf{N}!$ permutations. Note that permutations can be composed similarly to the composition of functions, and thus S_{N} forms a group with respect to composition ($\circ$) operation. S_{N} satisfies group properties of closure, associativity, and presence of identity. However, permutations are not commutative under composition but are invertible.

Sampling a random permutation denotes choosing a random $\pi \in S_{\mathsf{N}}$. We next describe how parties P_i, P_j can sample a common random permutation non-interactively using the shared key established via $\mathcal{F}_{\mathsf{Setup}}$. P_i, P_j non-interactively generate N common random values say $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_{\mathsf{N}} \in \mathbb{Z}_{2^\ell}$ where $\ell >> \log_2 \mathsf{N}$. The parties tag each of the values v_i with its index to obtain a list $S = \{(\mathsf{v}_i, \mathsf{x}_i)\}_{i=1}^{\mathsf{N}}$, where $\mathsf{x}_i = i$. Each party then locally sorts this list of tuples based on the first entry v_i of each tuple to obtain a sorted list $S' = \{(\mathsf{v}'_j, \mathsf{x}'_j)\}_{j=1}^{\mathsf{N}}$. The second element in each tuple of S' defines a random permutation where $\pi(\mathsf{x}_i) = \pi(i) = \mathsf{x}'_i$ for $i \in \{1, 2, \ldots, \mathsf{N}\}$.

2.5.2 Shuffle protocol of [13]

Let a table T denote a set of ordered rows where each row consists of an ℓ-bit string. Let N denote the size of T or the number of rows in T. Here, the table is equivalent to considering a vector with N elements where each element is an ℓ-bit value. The work of [13] considers performing a 3PC shuffle protocol in the honest-majority setting. It takes as input $[\cdot]^{\mathbf{B}}$-shares of the table T, and outputs $[\cdot]^{\mathbf{B}}$-shares of the table shuffled using a random secret permutation π. It also outputs a flag that indicates the correctness of $[\cdot]^{\mathbf{B}}$-shares of the shuffled table. Note here that shuffling the table using a random π denotes the operation of rearranging the rows in T as per π.

In the semi-honest setting, [13] presents a shuffle protocol which is an adaption of the shuffle protocol of [145] to the 3-party setting. This semi-honest protocol requires three rounds of interaction and guarantees privacy against a malicious adversary. [13] contributes to making this semi-honest protocol secure in the presence of a malicious adversary by augmenting with a verification phase (Set-Equality protocol) to ensure the correctness of the semi-honest shuffle, which additionally requires $2 + \log_2 \kappa$ rounds. Elaborately, the semi-honest shuffle comprises three invocations of Shuffle-Pair protocol. In each instance of Shuffle-Pair, a random permutation

is applied to the input (of the Shuffle-Pair), where the permutation is known to a distinct pair of parties and is hidden from the third. The output of the current Shuffle-Pair is fed as input to the next Shuffle-Pair. The composition of all three permutations, thus, makes up the random secret permutation used to shuffle the input table. Since each party is aware of only two permutations, the final permutation remains private. Each invocation of Shuffle-Pair is followed by a Set-Equality protocol which outputs a flag $\in \{0, 1\}$ indicating whether the table output by the Shuffle-Pair is indeed a random permutation of the input to this Shuffle-Pair. In this way, the output of the shuffle protocol is guaranteed to be correct if all instances of Shuffle-Pair are verified to be correct. In the following, we first provide the Shuffle-Pair protocol followed by the Set-Equality protocol.

Shuffle-Pair Let table T be $[\cdot]^{\mathbf{B}}$-shared. Shuffle-Pair enables parties to generate $[\mathsf{T}']^{\mathbf{B}}$ where $\mathsf{T}' = \pi_{ij}(\mathsf{T})$ and π_{ij} is a random permutation held by $P_i, P_j \in \mathcal{P}$. Here, $\pi_{ij}(\mathsf{T})$ denotes the operation of permuting the elements in T as per π_{ij}. We describe the protocol with respect to P_0, P_1 who hold the permutation π_{01} in Fig. 2.7. The protocol for the other pair of parties can be worked out analogously.

Protocol Shuffle-Pair $\left([\mathsf{T}]^{\mathbf{B}}, P_0, P_1, \pi_{01}\right)$

Preprocessing:

1. P_0, P_2 randomly sample $[\mathsf{T}']^{\mathbf{B}}_{02} \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$, and P_1, P_2 randomly sample $[\mathsf{T}']^{\mathbf{B}}_{12} \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$, non-interactively using the common keys established via $\mathcal{F}_{\mathsf{Setup}}$.

Online:

1. P_0 computes $\rho_0 = \pi_{01}\left([\mathsf{T}]^{\mathbf{B}}_{02}\right) \oplus [\mathsf{T}']^{\mathbf{B}}_{02}$ and sends it to P_1. Similarly, P_1 computes $\rho_1 = \pi_{01}\left([\mathsf{T}]^{\mathbf{B}}_{12}\right) \oplus [\mathsf{T}']^{\mathbf{B}}_{12}$ and sends it to P_0.
2. P_0 and P_1 compute $[\mathsf{T}']^{\mathbf{B}}_{01} = \pi_{01}\left([\mathsf{T}]^{\mathbf{B}}_{01}\right) \oplus \rho_0 \oplus \rho_1$.

Figure 2.7: Shuffle-Pair [13, 145].

Set-Equality The input for the Set-Equality protocol is a pair of tables $(\mathsf{T}, \mathsf{T}')$, each comprising of n rows and an m-bit string in each row. Thus, the table comprises n rows and m columns. Here, T' is obtained by randomly shuffling T. The protocol returns as output a 1 if the two tables are different and a 0 otherwise. The protocol chooses random subsets of rows and columns and verifies that the bits in the intersection of the chosen rows and columns are the same for

both the underlying tables. This verification is performed κ times to ensure a low probability of cheating. In order to choose the subset of rows randomly, each row of T is extended by κ random and secret bits before the shuffle. Consequently, after the shuffle, every row in T' has the same κ-sized suffix which it had in T. The rows for the ℓth test are picked based on the bit in the ℓth column of the κ-sized extension, with the row being chosen if the corresponding bit is a 1. Let R_ℓ be the n bit vector that denotes this selection of rows (i.e., $m+\ell$th column of the table). Similarly, let C_ℓ be the $m' = m + \kappa$ bit public vector that denotes the choice of columns picked for the ℓth test. The verification test compares the XOR of the values in the intersection of the chosen rows and columns in T and T' to check for the correctness of the shuffle. This is captured by the operations performed as described in Equation (2.1), where $C_{j,\ell}$ denotes the jth component of the vector C_ℓ. The output of the check is a bit V_ℓ such that if the tables are different, some V_ℓ, for $\ell \in \{1, \ldots, \kappa\}$, is non-zero with high probability. To detect if any V_ℓ is non-zero, a flag is computed, which is the OR of all these V_ℓ's, followed by reconstructing flag. A non-zero flag indicates misbehaviour in the Shuffle-Pair instance for which Set-Equality is performed.

$$[V_\ell]^{\mathbf{B}} = \sum_{i=1}^{n} [\mathsf{T}'_{i,m+\ell}]^{\mathbf{B}} \cdot \sum_{j=1}^{m'} C_{j,\ell} \cdot [\mathsf{T}'_{i,j}]^{\mathbf{B}} - \sum_{i=1}^{n} [\mathsf{T}_{i,m+\ell}]^{\mathbf{B}} \cdot \sum_{j=1}^{m'} C_{j,\ell} \cdot [\mathsf{T}_{i,j}]^{\mathbf{B}} \tag{2.1}$$

We note that performing the steps of Set-Equality by relying on a robust MPC yields a robust Set-Equality protocol, as done in the work of [13].

Regarding the security of [13] As per the discussion above, observe that their protocol correctly determines the output shares of the shuffled table for each party, in case no misbehaviour occurs during any of the three Shuffle-Pairs. However, since Set-Equality is performed via robust MPC, it guarantees that any malicious activity in the corresponding Shuffle-Pair is detected since some V_ℓ, for $\ell \in \{1, \ldots, \kappa\}$, will be non-zero with high probability. Such misbehaviour in the Shuffle-Pair is indicated by a flag bit being set to a 1. In such an event, observe that parties will not possess the correct output shares, and the protocol cannot proceed further. Thus, the protocol only provides security with abort (considering the robust ideal functionality of shuffle, defined in Fig. 4.10). Moreover, observe that a malicious party can also misbehave such that it learns the output shares but deprives the honest parties of the correct shares. This is possible if a malicious party aborts in the last Shuffle-Pair invocation. Elaborately, consider the last Shuffle-Pair among the three invocations used for getting a random shuffle. Without loss of generality, the protocol proceeds exactly as given in Fig. 2.7 with P_0 and P_1 being the communicating parties. In case P_1 is corrupt, it may receive ρ_0 from P_0, obtain the output

shares of the shuffled table, and then abort. Since the honest party P_0 does not receive ρ_1 from P_1, it does not learn its output shares. Hence, we believe [13]'s protocol gives only security with abort.

Complexity of [13] Observe that the protocol of [13] requires three invocations of Shuffle-Pair, each of which is followed by a Set-Equality protocol that verifies the correctness of the semi-honest Shuffle-Pair. While Shuffle-Pair requires only one round of communication, the Set-Equality protocol requires $2 + \log_2 \kappa$ rounds (one round is required for performing multiplications as per (2.1), followed by $\log_2 \kappa$ rounds for computing OR of κ bits, followed by one round of reconstruction), where κ is the security parameter. Thus, the overall round complexity is $3 + 2 + \log_2 \kappa$. With respect to the communication complexity, it requires communicating $6\mathsf{N}\ell$ bits for the three Shuffle-Pair instances, and additional communication for evaluating the Set-Equality protocol that involves computing $2\mathsf{N}\kappa$ Boolean multiplications, computing OR of κ bits and a robust reconstruction.

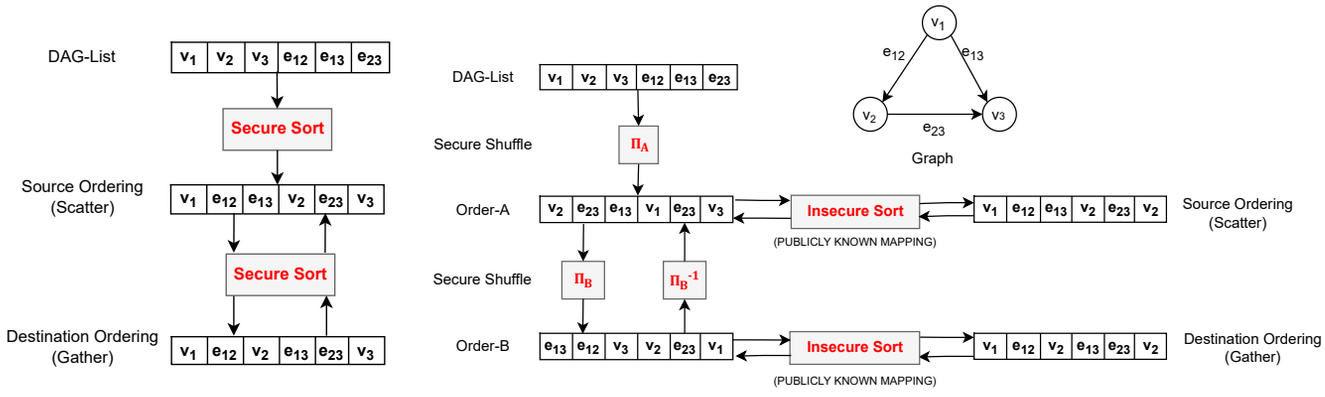
2.6 GraphSC paradigm

Real-world graphs are known to be sparse. Hence, naively using the adjacency matrix representation of the graph for computations would be expensive. Thus, designing an efficient solution to address the same involves—(i) designing an effective representation of the graph structure, (ii) ensuring the computation does not leak any private information, (iii) designing solutions that are highly parallelizable. The work by Nayak et al. [179] is the first to address the above problem and provides a framework for the same. The framework operates on a data-augmented directed graph $\mathsf{G}(\mathsf{V}, \mathsf{E}, \mathsf{Data})$ which consists of a directed graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$ where V is the set of vertices (or nodes, used interchangeably), E is the set of edges and Data is a set of user-defined data values associated with each vertex and edge of the graph. The data augmented graph is expressed as a list of vertices and edges where every vertex $\mathsf{v} \in \mathsf{V}$ is encoded as a tuple $(\mathsf{v}, \mathsf{v}, 1, \mathsf{data})$ and every edge $(\mathsf{u}, \mathsf{v}) \in \mathsf{E}$ is encoded as a tuple $(\mathsf{u}, \mathsf{v}, 0, \mathsf{data})$. The third entry in each tuple is a bit, isV, which equals 1 for a vertex and 0 otherwise, while data refers to the state information stored at each vertex and edge. These tuples constitute the data augmented graph list (DAG list). The DAG list representation of the graph is used to effectively represent the graph. Note that an undirected graph can be converted into a directed graph by accounting for each edge twice (incoming and outgoing edge). To perform secure computation over the graph while hiding its topology, each tuple in the data-augmented graph is secret-shared entry-wise between the computing parties. This ensures that parties cannot distinguish between shares of

a tuple corresponding to a vertex from that of an edge.

In principle, the framework of [179] enables securely evaluating message-passing graph algorithms. The latter are graph algorithms that operate in multiple rounds, where in each round, the nodes in the graph—(i) use their state information to send messages over their outgoing edges; (ii) receive messages along their incoming edges and aggregate these messages; (iii) use these messages to update their state. These three operations are abstracted into three primitives—Scatter, Gather, and Apply, respectively. Assuming that the graph algorithm can be expressed as a composition of the above primitives, the work of [179] enables its secure evaluation via MPC [2]. Since designing an MPC protocol naively may leak information regarding the graph topology, the framework first designs a data-oblivious algorithm for each of these primitives, followed by a translation of the same using the generic 2-party protocol of [231]. In general, an algorithm is said to be data-oblivious if the instructions executed and the memory accesses made during the run of the algorithm are independent of the input and hence leak no information about the input. To obtain a data-oblivious algorithm for the GraphSC primitives, it is important to ensure that each entry in the DAG list is visited when realizing these primitives to ensure no information about the association between the entries (such as an edge being incident on a node) is leaked. Observe that Apply can be computed obliviously by scanning through the DAG list representation and applying the user-defined function if the element is a vertex and performing a dummy operation otherwise. To compute Scatter and Gather obliviously, [179] relies on two different sorted orders of the DAG list representation. The *source order* requires the DAG list to be sorted such that every node in the graph is placed before all edges that originate from it. The *destination order* requires the DAG list to be sorted such that all edges that end at a particular node are placed before that node. Let $\Pi_{\mathsf{Sort}}(\llbracket \mathbf{x} \rrbracket, key)$ denote an oblivious sort protocol that outputs a sorted list of elements in a secret-shared vector $\mathbf{x}$ based on the key key. Given a data-oblivious sort protocol such as Bitonic sort [195], switching between the source order and destination order each time a Scatter or Gather is applied, ensures obliviousness as follows. Scatter can be accomplished obliviously by linearly scanning through the DAG list sorted in the source order. For this, if the current tuple in the list is a node, the data value at the node is picked up, and if the current tuple is an edge, then the value picked up at the most recent tuple is used to update the edges. Gather can also be done obliviously by a linear scan through the DAG list sorted in the destination order. During the scan, if the current tuple is an edge, its value is stored in an aggregate variable by applying an aggregation

[2]The operations within Scatter and Gather will vary across different graph algorithms. Hence, [179] provides a generic GraphSC framework, where the operations to be performed within Scatter and Gather are assumed as a black box.



(a) GraphSC paradigm of [179]. (b) GraphSC paradigm of [13].

Figure 2.8: Overview of operations involved in GraphSC.

operation, and if the current tuple is a node, then the aggregate variable is stored along with the node. This approach of performing Scatter and Gather by performing a linear scan over a sorted order is oblivious as every node and edge of the graph is operated on, without revealing the relationship between the nodes and edges. Given that the primitives can be performed obliviously, as described above, the graph computation can be performed securely using MPC protocols. To summarize, message-passing graph algorithms can be computed obliviously by using the following steps in every message-passing round—(i) sort based on source order, (ii) Scatter, (iii) sort based on destination order, (iv) Gather and (v) Apply. An illustration of the operations involved in the GraphSC paradigm of [179] is given in Figure 2.8. Further, the framework in [179] provides a parallel algorithm for each of the individual primitives—Scatter, Gather, and Apply. In a multiprocessor setting, the parallel variants allow the computations to be performed in sub-linear complexity rather than the linear complexity of $\mathcal{O}(|\mathsf{V}| + |\mathsf{E}|)$ in the size of the graph. We remark that this technique of obtaining a sub-linear solution in the multiprocessor setting also extends to our protocols. An overview of the sub-linear solution of [179] is provided in §2.6.1.

The work of [13] improves on the work of [179]. First, it combines Gather and Apply primitives such that both operations can be achieved in a single pass through the DAG list. Moreover, [13] observes that the approach of [179] has the drawback of requiring an oblivious sort each time a Scatter or Gather primitive is applied. This amounts to two calls to an oblivious sort in every message-passing round. Instead, [13] observes that a secret shuffle followed by an insecure sort (which reveals the result of the comparisons) can be used to realize an oblivious sort. Let $\Pi_{\mathsf{Shuffle}}([\![\mathbf{x}]\!])$ denote an oblivious shuffle protocol that outputs the elements of the secret-shared vector $\mathbf{x}$ in a random shuffled order. Observe that since the list is first shuffled, revealing the

result of comparisons during sort does not break the obliviousness property. On the other hand, the insecure sort is required to be performed only once in the beginning, subsequent to which, the public permutation (obtained as output from the insecure sort) can be applied to sort the DAG list, non-interactively. In summary, in the first message-passing round, a secret shuffle followed by an insecure sort (e.g., a comparison sort algorithm) is applied to get the required source or destination order. The permutations which map from the shuffled orders to the source and destination order are made public, as they do not reveal any information about the DAG list. In the subsequent rounds, the secret shuffle, followed by the public permutation, is applied to get the required sorted order. Since shuffle can be performed much more efficiently than a sort, this change brings in significant efficiency improvements. An illustration of the operations involved in the GraphSC paradigm of [13] is given in Figure 2.8. Finally, to perform secure computation, [13] considers a 3PC setting to further enhance efficiency. Although the framework of [179] requires an explicit "Apply" operation, we note that it can be performed along with the "Gather" operation and hence, is not explicitly depicted in Figure 2.8. Table 2.3 provides a list of most frequently used notations with respect to the GraphSC paradigm.

Notation	Description
V	Set of vertices
E	Set of edges
Data	Set of Data values
$\mathsf{G(V, E, Data)}$	Data augmented graph
DAG list	List representation of data augmented graph
$\mathsf{G}[i]$	i^{th} entry/tuple of the DAG list
$\mathsf{G}[i].\mathsf{isV}$	Denotes if tuple i corresponds to a vertex
$\mathsf{G}[i].\mathsf{dt}$	State information stored at tuple i
$\mathbf{A}$	Adjacency Matrix
$\mathbf{z}[\mathsf{v}]$	v^{th} component of a vector $\mathbf{z}$

Table 2.3: Table of notations pertaining to GraphSC paradigm.

2.6.1 Parallel variant of GraphSC paradigm [179]

We provide an overview of the steps involved in translating the linear round sequential protocols for computing Scatter and Gather to attain a logarithmic round parallel protocol, in the presence of multiple processors, as described in the work of [179]. Assuming that there are $M = |\mathsf{V}| + |\mathsf{E}|$ processors, we begin by describing how Gather can be performed in parallel. The parallel variant of Scatter can also be achieved similarly. We conclude by describing the approach to perform Scatter-Gather in parallel when there are only $P < M$ processors. We refer an interested reader to [179] for further details.

Performing Gather in parallel Let $\ominus$ represent the aggregate operation to be performed within Gather. Recall that when considering the destination sorted list, the aggregate operation updates the data associated with every vertex with the data present in the longest consecutive sequence of edges preceding it. Let $\mathsf{LPS}[i, j)$ for $1 \leq i < j \leq |\mathsf{V}| + |\mathsf{E}|$ denote the "sum" (aggregation with respect to $\ominus$ operator) of the data present in the longest consecutive sequence of edges before j beginning from (and including) i. Observe that computing $\mathsf{LPS}[1, j)$ for $1 \leq j \leq |\mathsf{V}| + |\mathsf{E}|$ and updating a vertex j with $\mathsf{LPS}[1, j)$, is equivalent to updating the data at the vertices as done in Gather. Thus, to obtain the parallel variant of Gather given $M = |\mathsf{V}|+|\mathsf{E}|$ processors, we assign the task of computing $\mathsf{LPS}[1, j)$ to the j^{th} processor.

To attain the logarithmic round complexity, in each time step τ, the parallel algorithm computes LPS values for all segments of length 2^τ. The LPS values of the consecutive 2^τ-length segments computed in time step τ are then used to compute LPS values of $2^{\tau+1}$-length segments in the next time step, $\tau + 1$. In this way, since the input is a segment of length $|\mathsf{V}| + |\mathsf{E}|$, the computation of the last LPS entry, $\mathsf{LPS}[1, |\mathsf{V}| + |\mathsf{E}|)$ can thus be accomplished in $\log(|\mathsf{V}| + |\mathsf{E}|)$ time steps. Concretely, in time step τ, a processor $j \in \{1, 2, \ldots, |\mathsf{V}| + |\mathsf{E}|\}$ computes $\mathsf{LPS}[j - 2^\tau, j)$ (a τ-length segment). Thus, in the next time step, it can compute $\mathsf{LPS}[j - 2^{\tau+1}, j)$ by combining $\mathsf{LPS}[j - 2^{\tau+1}, j - 2^\tau)$ and $\mathsf{LPS}[j - 2^\tau, j)$. A subtle thing to note here is that a segment can be aggregated with the immediately preceding segment of equal size *only if* a vertex has not been encountered in between. At the end of $\log(|\mathsf{V}| + \mathsf{E})$ time steps, vertex j's data for $j \in \{1, 2, \ldots, |\mathsf{V}| + |\mathsf{E}|\}$ is updated with $\mathsf{LPS}[1, j)$. The formal protocol for the same appears in Algorithm 1.

Algorithm 1: Parallel Gather(G)

Initialization: *Every processor j computes* ;

1 $$\mathsf{LPS}[j-1, j) = \begin{cases} G[j-1].\mathsf{dt}, & \text{if } G[j-1].\mathsf{isV} = 0 \\ 0, & \text{otherwise} \end{cases}$$

2 $$\mathsf{isV}[j-1, j) = \begin{cases} 1, & \text{if } G[j-1].\mathsf{isV} = 1 \\ 0, & \text{otherwise} \end{cases}$$

Main Algorithm:

3 **For** $\tau = 0$ *to* $\log(|\mathsf{V}| + |\mathsf{E}|) - 2$, *each processor j computes:*

4 $$\mathsf{LPS}[j - 2^{\tau+1}, j) = \begin{cases} \mathsf{LPS}[j - 2^{\tau+1}, j - 2^\tau) \ominus \mathsf{LPS}[j - 2^\tau, j), & \text{if } \mathsf{isV}[j - 2^\tau, j) = 0 \\ \mathsf{LPS}[j - 2^\tau, j), & \text{otherwise} \end{cases}$$

5 $$\mathsf{isV}[j - 2^{\tau+1}, j) = \mathsf{isV}(j - 2^{\tau+1}, j - 2^\tau) \vee \mathsf{isV}[j - 2^\tau, j)$$

end

*$\vee$ denotes the OR operation

Performing Scatter in parallel Recall that in the propagate operation performed as part of Scatter, each edge updates its data with the data of the nearest vertex preceding it. This propagate operation can also be performed in parallel, similar to as done in the aggregate operation. In fact, propagate can be represented as a special case of aggregate as follows. Initially, each edge either stores the value of the vertex if a vertex immediately precedes it, else it stores $-\infty$. Next, an aggregate operation can be performed where $\ominus$ is the max operator, i.e., $\mathsf{LPS}[j-2^{\tau+1}, j) = \max\{\mathsf{LPS}[j-2^{\tau+1}, j-2^{\tau}), \mathsf{LPS}[j-2^{\tau}, j)\}$. At the end of $\log(|\mathsf{V}|+\mathsf{E})$ time steps, the j^{th} processor computes $\mathsf{LPS}[1, j)$ which is the value of the nearest vertex preceding j. Thus, if j is an edge, its data can be updated with $\mathsf{LPS}[1, j)$.

Performing Gather in parallel with $P < M$ processors In such a scenario, instead of holding a processor responsible for computing a single cell of LPS, it is assigned a consecutive range of cells to be computed. Without loss of generality, assume that M is a multiple of P. Let processor j be assigned the range $[s_j, t_j]$ where $s_j = (j-1) \cdot \frac{M}{P} + 1$ and $t_j = j \cdot \frac{M}{P}$. Each processor first computes $\mathsf{LPS}[s_j, t_j + 1)$ sequentially in $\mathcal{O}(\frac{M}{P})$ time steps. Then, assuming that each $\mathsf{LPS}[s_j, t_j + 1)$ is a single value, the processors run the parallel algorithm for Gather on this segment of length P. Thus, the number of time steps required for this algorithm is $\mathcal{O}\left(\frac{M}{P} + \log P\right)$.

Chapter 3

Secure Local Clustering

In this chapter, we discuss our privacy-preserving solution for local clustering. In the process, we also discuss the additional primitives that are added to the 3PC framework of [136] that result in making the latter more comprehensive. The results in this chapter have led to two publications at PoPETs 2023 [211, 212].

3.1 Overview

Our primary objective is to provide a privacy-preserving solution for local clustering (a local cluster is a cluster of nodes that is centred around a given seed node), which is achieved for the first time. For this, we design a secure protocol for the state-of-the-art (cleartext) algorithm in [54]. This local clustering algorithm relies on the heat kernel PageRank (HKPR) metric to quantify the similarity of nodes and thereby identify the cluster. Thus, a secure protocol for computing local clustering demands a secure protocol for computing HKPR values, which is also designed in our work. The state-of-the-art algorithm of [222] forms the basis for our secure HKPR protocol.

To obtain as efficient a solution as possible, our protocols are designed in the GraphSC paradigm [179, 13] that provides a generic framework for evaluating message-passing graph algorithms securely while ensuring efficiency through parallel computations. We refer to §2.6 for details of this paradigm. Securely computing local clustering and HKPR metric via GraphSC entails the following steps:

① ***cleartext → message-passing algorithm:*** This involves identifying components of the relevant cleartext algorithm that can benefit from GraphSC and translating these to the

message-passing paradigm. This is non-trivial to achieve since the graph algorithm under consideration may not lend itself as a message-passing one.

② ***message-passing → secure protocol:*** This entails designing a data-oblivious variant of the message-passing algorithm by casting it in the GraphSC paradigm by defining the Scatter and Gather primitives. We emphasize that since GraphSC is a generic framework, it treats the operations within Scatter-Gather as a black box in the process of designing a data-oblivious solution. Hence, we define these specific to the considered graph algorithms. Note that our definitions of Scatter-Gather do not follow trivially and entail challenges as elaborated in §3.5.2, §3.6.2. Moreover, operations performed within Scatter-Gather should be defined carefully since these directly impact the efficiency of the resulting protocol. To design secure protocols for the data-oblivious variant we rely on the state-of-the-art robust 3PC of SWIFT [136]. However, the framework of SWIFT lacks several essential primitives, such as secure shuffle and division, required for evaluating the local clustering algorithm in the GraphSC paradigm. We design these primitives, making SWIFT more comprehensive. We elaborate on these results next.

Secure local clustering

Naively translating the local clustering algorithm in [54] via the above steps results in a data-oblivious algorithm (and thereby a secure protocol) that has $\mathcal{O}(|\mathsf{V}|(|\mathsf{V}|+|\mathsf{E}|))$ complexity. This complexity results from the $|\mathsf{V}|$ iterations of the clustering algorithm, each of which requires computing a cluster-specific parameter (as described in §3.6.2) via Scatter-Gather where the latter has $\mathcal{O}(|\mathsf{V}|+|\mathsf{E}|)$ complexity. Instead, we introduce a novel approach and design an algorithm (step ①) which can compute this cluster parameter incrementally, by reusing information from prior iterations without relying on Scatter-Gather. Elaborately, we augment the data associated with each vertex with a new component (to be computed via Scatter-Gather) that facilitates this incremental computation. Our reliance on a single call to Scatter-Gather to compute this new component, enables designing a significantly better data-oblivious algorithm having $\mathcal{O}(|\mathsf{V}|+|\mathsf{E}|)$ complexity (step ②). This complexity can further be improved to $\mathcal{O}(\log(|\mathsf{V}|+|\mathsf{E}|))$ relying on the parallel variant of GraphSC described in [179] (recalled in §2.6.1). Our new message-passing algorithm (step ①) coupled with efficient definitions of Scatter-Gather for it (step ②), aid in obtaining a more efficient secure protocol for local clustering.

Secure HKPR

Although obtaining the message-passing algorithm (step ①) for computing HKPR is relatively simpler compared to the one for local clustering, we note that the resulting algorithm requires keeping track of several parameters. Thus, keeping efficiency in mind, we define Scatter and Gather (step ②) such that we eliminate the overheads resulting from additional bookkeeping present in the message-passing variant. This results in obtaining an efficient realization of the secure protocol.

Note that our secure HKPR protocol can also be of independent interest since it finds use in other applications such as community detection [134], classification [170], etc. As in the case of the cleartext algorithm in [222], our secure protocol for computing HKPR is also versatile and allows computing several other graph propagation metrics such as L-hop transition probability, PageRank [192], personalized PageRank (PPR) [192], single-target PPR [159], Katz [122] measure, to name few.

Enhancing the 3PC framework of SWIFT

The GraphSC framework [179, 13] treats the underlying MPC as a black box, and hence does not require explicitly handling fixed-point arithmetic (FPA) operations. However, the algorithm of clustering requires operating over FPA and demands new primitives such as division, support for which is missing in SWIFT [136]. Hence, we design a secure protocol for division, which in turn requires designing a secure protocol for prefix OR. These primitives were originally missing in the framework of SWIFT. The addition of these makes SWIFT a more comprehensive framework. While SWIFT originally provided support for privacy-preserving machine learning (PPML) *inference* only (for neural networks), the inclusion of secure division protocol now also facilitates privacy-preserving *training* of neural networks. Further, since SWIFT was originally designed as an MPC framework for PPML, making it compliant with the GraphSC framework additionally requires a shuffle protocol. Keeping efficiency in mind, we design ring-based maliciously secure shuffle protocol, Ruffle, in the 3-party computation setting, whose highlight is the improved online efficiency in comparison to both [80] and [13]. In fact, Ruffle also has a better overall run time than the shuffle protocol of [80] owing to our better round as well as communication complexity. Finally, we note that Ruffle is designed to offer an improved security guarantee of robustness in comparison to prior works. Further, the existing secure shuffle protocols are designed for settings that require a single shuffle invocation. Thus, when requiring multiple sequential shuffles, as in the case of the GraphSC paradigm, the existing protocols must be invoked as many times as required. While our approach also requires

multiple invocations, we aim to minimize the overall efficiency while performing multiple sequential shuffles. In this regard, we identify two different scenarios, Independent-Shuffles and Composed-Shuffles, that arise while performing sequential shuffles and cater to both separately by designing $\mathsf{Ruffle}_{\mathsf{ind}}$ and $\mathsf{Ruffle}_{\mathsf{cmp}}$, respectively, as discussed in §3.4.3.

We note that SWIFT and its enhanced version provide the strongest security of GOD. Although GOD may be beneficial for several applications, it is worthwhile to note here that our protocols allow settling for the weaker security guarantee of fairness or even security with abort, depending on the application scenario. Moreover, we note that the added security of GOD is achieved at no extra (amortized) cost over the weakest guarantee of abort security.

Benchmarks

The designed secure protocols for clustering and HKPR operate on FPA, which has limited precision compared to its floating point counterpart. Further, probabilistic truncation and approximate division in the secure computation setting result in additional loss of accuracy. Hence, we perform extensive benchmarks to evaluate the accuracy loss of our secure clustering and HKPR protocols. Further, recall that the secure protocol for HKPR supports evaluating other graph propagation metrics. Hence, we also evaluate the accuracy loss involved in securely realizing all these metrics. We observe that the accuracy loss of our secure protocols is in the order of 10^{-5}, which makes them on par with their cleartext counterparts.

We also benchmark the secure clustering and HKPR protocols over [136], and report the run time for the same. Our implementation also accounts for parallelization that can be achieved in the multiprocessor setting using the techniques described in [179]. The parallel variants witness improvements of up to 23.4× and 14.4×, respectively, in the computation of HKPR and clustering, when considering a graph of size 10^6. The reported numbers showcase the practicality of the designed protocols. We also benchmark the performance of our shuffle protocols, Ruffle, $\mathsf{Ruffle}_{\mathsf{ind}}$ and $\mathsf{Ruffle}_{\mathsf{cmp}}$. We empirically establish that the protocols, $\mathsf{Ruffle}_{\mathsf{ind}}$, $\mathsf{Ruffle}_{\mathsf{cmp}}$ are apt for their respective scenarios involving multiple sequential shuffle invocations. Further, since the application of anonymous broadcast builds directly on top of $\mathsf{Ruffle}_{\mathsf{ind}}$, we showcase the efficiency improvements brought in by our shuffle protocol for anonymous broadcast in comparison to the state-of-the-art 3-party anonymous broadcast system of [80].

3.2 Related work

Local clustering as well as computing HKPR metric via MPC has not been explored. Hence, we discuss the related works that consider cleartext computation of the same first. Following this, for completeness, we also describe works that design privacy-preserving solutions for global clustering and PageRank, followed by the related works for the missing MPC primitives required for securely realizing local clustering.

Local clustering: Local clustering was initiated in the work of [214, 215], which identified clusters around a seed node by performing random walks over the graph. Since random walks starting from the seed node are more likely to visit nodes near the seed node, they help in identifying the local structure. The random walk-based method was improved in [9, 8] by relying on approximate PageRank vectors instead. When given a seed node, the PageRank vector provides a ranking of the nodes such that nodes that are more likely to be the endpoints of a random walk starting from the seed node get assigned a higher probability. Thus, the set of nodes with a higher rank constitutes a local cluster around the seed node. The state-of-the-art works on local clustering [54, 134] are based on heat kernel PageRank (HKPR) [53]. The advantage of HKPR over PageRank is that shorter random walks are more heavily weighted in HKPR, resulting in the walks being concentrated around the seed node. Hence, HKPR-based local clustering algorithms are known to provide a better-quality cluster. Since [54] provides the state-of-the-art solution in terms of cluster quality, we rely on the same while designing a secure protocol.

HKPR: Computing the HKPR vector, as required for clustering, has been considered in a series of works [230, 164, 134, 54, 222] that aim to optimize the computation. Among these, [222] forms the state-of-the-art, which outputs the most accurate HKPR vector compared to the prior works while also performing better in terms of efficiency. Since our goal is to identify the most accurate local cluster around a given seed node, we rely on the approach of [222] for computing the HKPR vector.

Privacy-preserving global clustering: Various works [226, 176, 124, 32, 10, 197, 99, 154, 228, 115, 111] have studied privacy-preserving solutions for global clustering with respect to four different types—(i) partitioning-based, (ii) distribution-based, (iii) density-based, and (iv) hierarchical clustering. In partition-based clustering, the goal is to split the input into K non-overlapping clusters. Popular examples of such clustering algorithms are K-means [217] and affinity propagation [83], which have been studied in the privacy-preserving setting in [226, 176] and [124], respectively. Distribution-based clustering algorithms assume that clusters are drawn from an unknown mixture of distributions and aim at approximating the orig-

inal distributions as well as the number of different distributions [103, 116]. A well-known example of distribution-based clustering algorithm is Gaussian Mixture Model (GMM) using the Expectation-Maximization (EM) algorithm [72], which has been studied in the privacy-preserving setting in [99, 154]. In density-based clustering, a density-based neighbourhood notion is used such that input records that lie together in a dense area are said to constitute a cluster. A popular example of such clustering algorithm is DBSCAN [81], which has been studied in the privacy-preserving setting in [32, 10, 197]. Hierarchical clustering algorithms [228] represent a data set as a binary tree of data points and iteratively merge or divide clusters based on the derived tree structure and have been studied in the privacy-preserving context in [69, 115, 111]. We refer an interested reader to [103] for a systematic analysis of privacy-preserving global clustering algorithms.

Privacy-preserving PageRank computation: Computation of the graph propagation metric of PageRank has been looked at in the privacy-preserving setting in various works [179, 142, 203, 56]. The work of [179] provides a GraphSC-based solution for the same via garbled circuits. The work of [142] provides a solution to securely realize PageRank via MPC, whereas the work of [203] uses additive homomorphic encryption for computing PageRank via the power method. The efficiency of [203] was further improved in the work of [56] by relying on their analysis of termination conditions for the power method.

MPC primitives: Note that other than the MPC primitives provided by [136] (see Table 2.1), we additionally require a secure realization for division and shuffle for local clustering. We next describe the relevant literature pertaining to these. With respect to secure division, note that this has been studied in various works such as [45, 43, 126, 6, 220, 50, 138, 174]. While some of them [174, 50, 138] rely on evaluating a division circuit via garbled circuits, others [45, 43, 126, 220] rely on an iterative method such as the Goldschmidt's division algorithm [168] which are more communication-efficient than garbled circuit based approaches. Due to efficiency reasons, we adapt the protocol of [43], which relies on Goldschmidt's division algorithm, over the 3PC setting. Further, performing secure division via Goldschmidt's algorithm additionally relies on prefixOR computation [44, 43, 132]. It is worthwhile to note that while computing prefixOR over the field algebraic structure can be accomplished in constant rounds by leveraging the presence of inverses over a field as done in the work of [43, 44], performing it over the ring algebraic structure requires a linear number of rounds in the number of bits for which prefixOR has to be computed [132]. We showcase how to achieve this with logarithmic round complexity by leveraging the multi-input multiplication gates.

With respect to secure shuffle, several works explore MPC-based techniques for the same [145, 178, 125, 167, 90, 177]. Some of these solutions rely on securely performing sort [145, 178],

while some others consider securely evaluating a permutation network [125, 167, 90, 177]. These techniques require at least $O(\log n)$ rounds for shuffling n elements, which proves to be expensive for time-sensitive applications. The concurrent works of [80, 13] consider a 3PC honest majority setting. In the semi-honest 3PC honest-majority setting, [13] presents a shuffle protocol which is an adaption of the shuffle protocol of [145] to the 3-party setting. This semi-honest protocol requires three rounds of interaction. Note that, [13] contributes to making this semi-honest protocol secure in the presence of a malicious adversary by augmenting with a verification phase to ensure the correctness of the semi-honest shuffle, which additionally requires $2+\log_2 \kappa$ rounds. Further, [13] also provides a 2 round semi-honest protocol but leaves open the question of attaining malicious security for the same. Clarion [80] also gives a 2-round 3PC honest-majority shuffle protocol which builds on the semi-honest 2-party protocol of [48]. To guarantee malicious security, they add integrity checks by having MACs appended to the elements to be shuffled. The resulting maliciously secure protocol requires 6 rounds overall. Clarion also extends its shuffle protocol to the n-party dishonest majority setting that guarantees malicious security, which additionally requires maliciously secure OTs (oblivious transfer) in the preprocessing phase. It improves over the protocol in [162] in terms of efficiency, however, it lacks in terms of security guarantees where the latter provides fairness in the preprocessing phase and GOD only in the online phase, for the setting of $t < n/3$.

3.2.1 On the choice of cleartext algorithms

Keeping the quality of the output cluster in mind, we rely on the state-of-the-art works in the literature for computing the HKPR metric as well as for performing clustering using the same. However, one could question the performance of these algorithms when translated to their secure variants via MPC, i.e., do there exist other algorithms that can provide better performance in MPC by trading off the output quality? For this, we note that our solution comes at no extra cost. For example, consider randomized alternatives to state-of-the-art (in cleartext) that trade off the quality of the output to obtain a more efficient solution. The improved efficiency can be attributed to operating only on a subset of the nodes (sampled based on nodes that satisfy some condition) rather than considering the entire graph. However, when translating the same to a secure variant, it is required that the computations are input-independent. Thus, if the secure computation algorithm works on a subset of nodes that are sampled based on some condition, this will leak information about the number of nodes that satisfy the condition by simply observing which nodes are operated on. The number of such nodes may thereby leak information about the structure of the input graph, which should otherwise be kept private.

Hence, to prevent such leakage, it is required that the secure algorithm operates on all nodes, albeit performing dummy operations on some nodes of the input graph (which ensures that computations are independent of the graph structure). Thus, the secure variants can no longer leverage the efficiency gains obtained by performing computations specific to a chosen subset of vertices. Hence, a secure variant of a graph algorithm is required to operate on the entire graph. This results in the asymptotic complexity of all alternatives being similar in the MPC domain. Hence, the designed secure protocols in no way trade-off output quality for efficiency.

With respect to the choice of [222] as the basis of our work, we note that the protocols in [222] outperform the prior protocols for HKPR-based clustering [230, 54] not only in terms of accuracy but also in terms of efficiency (see comparisons reported in [222]). Thus, [222] forms the state-of-the-art. We would further like to note that other solutions [230, 54, 134] are based on computing random walks, for which, to the best of our knowledge, there do not exist efficient MPC protocols. Naively performing the same would be highly expensive as it would require multiple scans of the edge list (to ensure obliviousness) when identifying each node of the random walk. Further, random walks only constitute one component of the algorithm, and hence additional overhead will be incurred due to other algorithmic components. For real-world graphs with the number of nodes and edges in the order of 10^6, this is highly inefficient. Hence, we conclude that the choice of our algorithm results in an efficient solution via MPC without compromising accuracy.

3.3 Preliminaries

3.3.1 System model

Let $\mathcal{P} = \{P_0, P_1, P_2\}$ denote the three compute parties connected via pairwise private and authentic channels in a synchronous network. We let $\mathcal{A}$ denote a static malicious probabilistic polynomial time adversary which corrupts at most one party in $\mathcal{P}$. We rely on the secret-sharing-based robust 3PC framework of SWIFT [136] for the underlying MPC. We refer to §2.3 for details of the secret-sharing semantics and for descriptions of the protocols from SWIFT used in this chapter.

3.3.2 GraphSC paradigm

The designed protocols leverage the GraphSC paradigm [179, 13] for efficiency reasons. Hence, we refer readers to §2.6 to familiarize themselves with the necessary background.

3.3.3 Notations

The notations used in this chapter are summarized in Table 3.1.

Notation	Description
$\boldsymbol{b}_{i,j}$	j^{th} bit in a block of bits $\boldsymbol{b}_i$
k	bit length of the input
f	number of precision bits when considering fixed point arithmetic
Independent-Shuffles	Scenario where multiple *independent* sequential shuffles are invoked
Composed-Shuffles	Scenario where multiple sequential shuffles are *composed*
T	Set of ordered elements
T_o	Output set after permuting the elements in T under a random permutation π
TTP	Trusted third party
HKPR	Heat kernel PageRank
$\mathsf{G} = (\mathsf{V}, \mathsf{E})$	Graph with set of vertices V and set of edges E
$\mathbf{A}$	Adjacency matrix of G
$\mathbf{D}$	Degree matrix of G
$\mathbf{v}[i]$	i^{th} element of vector $\mathbf{v}$
$\boldsymbol{\rho}[\mathsf{v}]$	HKPR value for vertex $\mathsf{v} \in \mathsf{V}$
w_i	Weight in the i^{th} iteration of the graph propagation metric computation
Y_i	Partial weight sum $\mathsf{Y}_i = \sum_{k=i}^{\infty} \mathsf{w}_k$
$\mathbf{r}$	Residue vector used in graph propagation metric computation
$\mathbf{q}$	Reserve vector used in graph propagation metric computation
Nb_u	Neighbouring vertices of vertex $\mathsf{u} \in \mathsf{V}$
deg_u	Degree of vertex $\mathsf{u} \in \mathsf{V}$
vol_S	Volume of S: sum of degrees of vertices in S
$\mathsf{val}, \mathsf{agg}$	Intermediate variables
Φ_s	Cheeger ratio of set of vertices S
∂	Number of edges that cross from set S to remaining vertices in $\mathsf{V} \backslash \mathsf{S}$ when computing Φ_s
ς	Target cluster volume
$\mathsf{G}[\mathsf{u}].\mathsf{GreaterCount}$	Number of neighbours v of vertex u that have a greater $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg}_\mathsf{v}}$ value than $\frac{\boldsymbol{\rho}[\mathsf{u}]}{\mathsf{deg}_\mathsf{u}}$.

Table 3.1: Notations used in this chapter.

3.4 Primitives for clustering

Most of the primitives required for clustering can be obtained from SWIFT [136] (described in Table 2.1). However, clustering additionally requires other primitives, which we detail in this section. These primitives make SWIFT a more comprehensive framework, facilitating applications beyond PPML. Since primitives such as prefix OR and division are known from the literature [220, 43, 44], we provide a high-level intuition of realizing them in SWIFT next.

3.4.1 Prefix OR

On input Boolean shared bits $\mathsf{x}_{\ell-1}, \ldots, \mathsf{x}_0$, Π_{PreOr} outputs Boolean shared bits $\mathsf{y}_{\ell-1}, \ldots, \mathsf{y}_0$ such that $\mathsf{y}_i = \vee_{j=i}^{\ell-1} \mathsf{x}_j$. We provide an efficient instantiation of prefix OR, which works over rings and leverages the presence of multi-input AND gates provided in [136], to yield a protocol that requires 3 rounds for 64-bit inputs.

The protocol proceeds as follows. We define an operator Ψ, which operates on a block of bits and outputs the prefix OR of the bits in the block. The Π_{PreOr} protocol is designed to recursively call the Ψ operator such that in each round, the size of the block increases exponentially, leading to a protocol with logarithmic rounds. Elaborately, our protocol proceeds in rounds such that after the j^{th} round, prefix OR of up to 4^j bits is computed. To achieve this, we define an operator Ψ which takes as input a block of bits $\boldsymbol{t}$ (where the number of bits in $\boldsymbol{t}$ is a power of 4), which is a concatenation of the bits in the four sub-blocks, $\boldsymbol{b}_3, \boldsymbol{b}_2, \boldsymbol{b}_1, \boldsymbol{b}_0$, i.e., $\boldsymbol{t} = \boldsymbol{b}_3||\boldsymbol{b}_2||\boldsymbol{b}_1||\boldsymbol{b}_0$. Here, each sub-block, $\boldsymbol{b}_3, \boldsymbol{b}_2, \boldsymbol{b}_1, \boldsymbol{b}_0$, is such that it is already the prefix OR of some input sub-block, i.e., there exist input sub-blocks $\boldsymbol{a}_3, \boldsymbol{a}_2, \boldsymbol{a}_1, \boldsymbol{a}_0$ such that $\boldsymbol{b}_i$ is the prefix OR of $\boldsymbol{a}_i$ for $i \in \{0, 1, 2, 3\}$. The operator Ψ is defined to output the prefix OR, $\boldsymbol{t}'$, of the bits in $\boldsymbol{t}$. Computation of Ψ proceeds as follows. Let the last bit of sub-block $\boldsymbol{b}_i$ be denoted as z_i and let $\boldsymbol{t}' = \boldsymbol{b}'_3||\boldsymbol{b}'_2||\boldsymbol{b}'_1||\boldsymbol{b}'_0$. Since each sub-block $\boldsymbol{b}_i$ already satisfies the prefix OR requirement, observe that we can set $\boldsymbol{b}'_3 = \boldsymbol{b}_3$. To compute the j^{th} bit in $\boldsymbol{b}'_2$, it suffices to compute the OR of z_3 with the j^{th} bit in $\boldsymbol{b}_2$, because the latter is already the prefix OR of the bits in $\boldsymbol{a}_2$. Similarly, j^{th} bit in $\boldsymbol{b}'_1$ can be computed as the OR of $\mathsf{z}_3, \mathsf{z}_2$ with the j^{th} bit in in $\boldsymbol{b}_1$. Finally, j^{th} bit in $\boldsymbol{b}'_0$ can be computed as the OR of $\mathsf{z}_3, \mathsf{z}_2, \mathsf{z}_1$ with the j^{th} bit in in $\boldsymbol{b}_0$. In this way, $\boldsymbol{t}' = \boldsymbol{b}'_3||\boldsymbol{b}'_2||\boldsymbol{b}'_1||\boldsymbol{b}'_0$ can be generated by performing 2/3/4-input OR, which can be reduced to a combination of NOT and multi-input AND. Formal details of Ψ appear in Fig. 3.1.

Having defined Ψ, we next describe how to compute prefix OR of $\{\mathsf{x}_i\}_{i=\ell-1}^{0}$ with the help of Ψ. The protocol proceeds in rounds, where in the j^{th} round, a processed version of $\{\mathsf{x}_i\}_{i=\ell-1}^{0}$ is split into $\ell/4^j$ blocks, $\boldsymbol{t}_i$, each of which consists of 4^j bits. Observe that when each $\boldsymbol{t}_i$ comprises four bits (in the initial round), each sub-block $\boldsymbol{b}_i$ consists of a single bit and already satisfies the prefix OR requirement. Thus, applying Ψ on $\boldsymbol{t}_i$ ensures that the invariant of each input sub-block to Ψ being a prefix OR is satisfied. At the end of round 1, the application of Ψ on each of the four-bit block $\boldsymbol{t}_i$, generates the prefix OR of the first four bits, as well as the prefix OR for each subsequent block of four bits. Thus, applying Ψ in the second round on each of the sixteen-bit blocks generates the prefix OR for the first 16 bits, as well as the prefix OR for each subsequent block of 16 bits. In this way, at the end of j rounds, the prefix OR of 4^j bits can be generated. The protocol for computing prefix OR appears in Fig. 3.2.

Operator $\Psi\left(\llbracket \boldsymbol{t} \rrbracket^{\mathbf{B}}\right)$

Input: $\llbracket \boldsymbol{t} \rrbracket^{\mathbf{B}} = \left(\llbracket \boldsymbol{b}_3 \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}_2 \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}_1 \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}_0 \rrbracket^{\mathbf{B}}\right)$ where each $\boldsymbol{b}_i$ is a block of d bits and constitutes the prefix OR of some sub-block of bits. Let the j^{th} bit in $\boldsymbol{b}_i$ be denoted as $\boldsymbol{b}_{i,j}$ and its last bit be denoted as z_i.

Output: $\llbracket \boldsymbol{t}' \rrbracket^{\mathbf{B}}$ such that bits in $\boldsymbol{t}'$ comprise the prefix OR of the bits in $\boldsymbol{t}$.

– For $i \in \{0, 1, \ldots, \mathsf{d}-1\}$, set

- $\llbracket \boldsymbol{b}'_{3,i} \rrbracket^{\mathbf{B}} = \llbracket \boldsymbol{b}_{3,i} \rrbracket^{\mathbf{B}}$
- $\llbracket \boldsymbol{b}'_{2,i} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{Mul}}\left(\llbracket \bar{\mathsf{z}_3} \rrbracket^{\mathbf{B}}, \llbracket \bar{\boldsymbol{b}}_{2,i} \rrbracket^{\mathbf{B}}\right)\right)$
- $\llbracket \boldsymbol{b}'_{1,i} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{3\text{-}Mul}}\left(\llbracket \bar{\mathsf{z}_3} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{z}_2} \rrbracket^{\mathbf{B}}, \llbracket \bar{\boldsymbol{b}}_{1,i} \rrbracket^{\mathbf{B}}\right)\right)$
- $\llbracket \boldsymbol{b}'_{0,i} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{4\text{-}Mul}}\left(\llbracket \bar{\mathsf{z}_3} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{z}_2} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{z}_1} \rrbracket^{\mathbf{B}}, \llbracket \bar{\boldsymbol{b}}_{0,i} \rrbracket^{\mathbf{B}}\right)\right)$

– Return $\llbracket \boldsymbol{t}' \rrbracket^{\mathbf{B}} = \left(\llbracket \boldsymbol{b}'_3 \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}'_2 \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}'_1 \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}'_0 \rrbracket^{\mathbf{B}}\right)$

Figure 3.1: Operator Ψ.

Protocol $\Pi_{\mathsf{PreOr}}\left(\{\llbracket \mathsf{x}_i \rrbracket\}_{i=\ell-1}^{0}\right)$

– $\llbracket \boldsymbol{b}_i^0 \rrbracket^{\mathbf{B}} = \llbracket \mathsf{x}_i \rrbracket^{\mathbf{B}}$ for $i \in \{0, \ldots, \ell-1\}$, and set $k = \ell$

– for $j = 0$ to $\lfloor \log_4(\ell) \rfloor$ do: (j denotes round number)

- for $i = \lfloor \frac{k}{4} \rfloor$ to 1 do: (i denotes block number)
  - $\llbracket \boldsymbol{t}_i^j \rrbracket^{\mathbf{B}} = \left(\llbracket \boldsymbol{b}_{4i-1}^j \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}_{4i-2}^j \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}_{4i-3}^j \rrbracket^{\mathbf{B}}, \llbracket \boldsymbol{b}_{4i-4}^j \rrbracket^{\mathbf{B}}\right)$
  - $\llbracket \boldsymbol{b}_i^{j+1} \rrbracket^{\mathbf{B}} = \Psi\left(\llbracket \boldsymbol{t}_i^j \rrbracket^{\mathbf{B}}\right)$
- $k = \lfloor \frac{k}{4} \rfloor$

– Return $\llbracket \boldsymbol{b}_3^{\lfloor \log_4(\ell) \rfloor + 1} \rrbracket^{\mathbf{B}}$

Figure 3.2: Prefix OR.

3.4.2 Division

When computing a/b where the divisor b is publicly known, and a is secret-shared, division can be performed easily by computing $1/\mathsf{b}$ on clear followed by secure multiplication with a. On the contrary, division, when b is also secret-shared, is non-trivial. Our division protocol, Π_{Div}, takes as input $\llbracket \mathsf{a} \rrbracket$ and $\llbracket \mathsf{b} \rrbracket$, and outputs $\llbracket \mathsf{d} \rrbracket$ where $\mathsf{d} \approx \mathsf{a}/\mathsf{b}$. It relies on Goldschmidt's approximation [168] for obtaining the result.

To design Π_{Div}, we follow a similar approach as in [43], which involves computing an initial guess for $1/\mathsf{b}$, followed by iteratively computing the approximation of a/b. To ensure a fast convergence of this iterative method, the choice of the initial guess for $\mathsf{w} = 1/\mathsf{b}$ is critical. To

compute w, we proceed along the lines of [43, 44]. We first compute the initial guess w' for the normalized input b', which is then used to obtain the initial guess w for the input b. Elaborately, if $\mathsf{b} > 0$, it is normalized to $\mathsf{b}' \in [0.5, 1)$ and $\mathsf{w}' = 1/\mathsf{b}'$ is approximated as $2.9142 - 2\mathsf{b}'$. Else, if $\mathsf{b} < 0$, it is normalized to $\mathsf{b}' \in (-1, -0.5]$ and $\mathsf{w}' = 1/\mathsf{b}'$ is approximated to $-2.9142 - 2\mathsf{b}'$. We refer readers to [44, 168] for the choice of the constant. Given w', the initial guess for $1/\mathsf{b}$ is computed as $\mathsf{w} = \mathsf{w}' \cdot \mathsf{v}$, where v is the scaling factor used to obtain the normalized $\mathsf{b}' = \mathsf{bv}$. We let Π_{AppRec} denote the protocol that computes the initial guess.

The following is the overview of Π_{AppRec}. Observe that given v, computing w follows directly from the sequence of relations described earlier to compute b' and w'. Thus, the challenge lies in computing v, which is non-trivial to obtain when b is available only in secret-shared format. Observe that if $|\mathsf{b}| \in [2^{m-1}, 2^m - 1]$ ($|\mathsf{b}|$ denotes magnitude of b), then the scaling factor v is given by $2^{\mathsf{k}-m-1}$. Here, k denotes the input length (see §2.1), and $m-1$ denotes the index of the most significant non-zero bit of b if $\mathsf{b} > 0$, and the most significant zero bit of b if $\mathsf{b} < 0$. Consider the case when $\mathsf{b} > 0$. To determine m, we compute prefix OR of the bits of b to generate bits $\{\mathsf{c}_i\}_{i=0}^{\mathsf{k}-2}$ such that $\mathsf{c}_i = 0$ for $i \geq m$, and $\mathsf{c}_i = 1$ for $i < m$. Thus, the bits $\{\mathsf{c}_i\}_{i=0}^{\mathsf{k}-2}$, when composed, yield $\mathsf{c} = 2^m - 1$. Using this, $\mathsf{v} = 2^{\mathsf{k}-m-1}$ can be computed as follows. We XOR the consecutive bits in $\{\mathsf{c}_i\}_{i=0}^{\mathsf{k}-2}$ to generate $\{\mathsf{y}_i\}_{i=0}^{\mathsf{k}-2}$ which ensures that $\mathsf{y}_i = 1$ for $i = m$, and $\mathsf{y}_i = 0$ otherwise. Thus, composing the bits in $\{\mathsf{y}_i\}_{i=0}^{\mathsf{k}-1}$ in the reverse order generates $\mathsf{v} = 2^{\mathsf{k}-m-1}$. The same steps can be used to compute v even when $\mathsf{b} < 0$, provided we work on the flipped bits of b. Thus, we obliviously flip/retain the bits of b depending on its sign, before the prefix OR computation begins. Having computed v, we can compute b'. However, computing w' using b' additionally requires an oblivious selection between 2.1942 and -2.1942 depending on the sign of b. Then, computing w from w' follows easily. The formal protocol for computing w appears in Fig. 3.3. We use $(\mathsf{x})_{\mathsf{f}}$ to denote that x has a precision of f bits. Note that this initial guess w of $1/\mathsf{b}$ is known to have a relative error of $\epsilon_0 = 1 - \mathsf{bw} < 1$ [45].

Given the initial guess w, Π_{Div} (Fig. 3.4) relies on Goldschmidt's method to output d which iteratively approximates a/b. The value of d in the θ^{th} iteration is computed as $\mathsf{d}_\theta = \mathsf{d}_{\theta-1} \cdot (1 - \mathsf{e}_{\theta-1})$. Here, e_θ denotes the relative error in the θ^{th} iteration and can be obtained as $\mathsf{e}_\theta = \mathsf{e}_{\theta-1} \cdot \mathsf{e}_{\theta-1}$. The algorithm begins with initializing $\mathsf{d}_0 = \mathsf{a} \cdot \mathsf{w}$ and $\mathsf{e}_0 = \epsilon_0$. Observe that after θ iterations, d_θ has a relative error of $(\epsilon_0)^{2^\theta}$, indicating that the error reduces exponentially. Since division by 0 is undefined, Π_{Div} additionally outputs a flag bit z (computed as part of Π_{AppRec}), which indicates if the divisor b is 0. For the application considered, we note $\theta = 4$ suffices to obtain the desired level of accuracy. We remark that the round optimized ($\log_4(\ell)$ rounds) protocols for prefix OR aid in attaining improved round complexity.

Protocol $\Pi_{\mathsf{AppRec}}(\llbracket \mathsf{b} \rrbracket)$

- $\alpha = (2.9142)_{\mathsf{k}-1}$
- $\llbracket \mathsf{b} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\llbracket \mathsf{b} \rrbracket)$
- for $i = 0$ to $\mathsf{k}-2$ do: $\llbracket \mathsf{b}'_i \rrbracket^{\mathbf{B}} = \llbracket \mathsf{b}_i \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{b}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}$
- $\{\llbracket \mathsf{c}_i \rrbracket^{\mathbf{B}}\}_{i=\mathsf{k}-2}^{0} = \Pi_{\mathsf{PreOr}}(\{\llbracket \mathsf{b}'_i \rrbracket^{\mathbf{B}}\}_{i=\mathsf{k}-2}^{0})$ and $\llbracket \mathsf{c}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}} = \llbracket 0 \rrbracket^{\mathbf{B}}$
- for $i = \mathsf{k}-2$ to 1 do: $\llbracket \mathsf{c}_i \rrbracket^{\mathbf{B}} = \llbracket \mathsf{c}_i \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{c}_{i+1} \rrbracket^{\mathbf{B}}$
- $\llbracket \mathsf{v} \rrbracket = \Pi_{\mathsf{B2A}}\left(\{\llbracket \mathsf{c}_i \rrbracket^{\mathbf{B}}\}_{i=\mathsf{k}-1}^{0}\right)$ (in reverse order)
- $\llbracket \mathsf{z} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{Mul}}\left(\llbracket \mathsf{b}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}, \llbracket \mathsf{c}_0 \rrbracket^{\mathbf{B}}\right)\right)$
- $\llbracket \mathsf{w}' \rrbracket = \Pi_{\mathsf{Sel}}\left(\alpha, -\alpha, \llbracket \mathsf{b}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}\right) - 2 \cdot \Pi_{\mathsf{Mul}}(\llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{v} \rrbracket, 0)$
- $\llbracket \mathsf{w} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{v} \rrbracket, \llbracket \mathsf{w}' \rrbracket, 2(\mathsf{k}-\mathsf{f}-1))$
- return $\llbracket \mathsf{w} \rrbracket$, $\llbracket \mathsf{z} \rrbracket^{\mathbf{B}}$

Figure 3.3: Computing the initial approximation of $1/\mathsf{b}$.

Protocol $\Pi_{\mathsf{Div}}(\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket)$

- $\llbracket \mathsf{w} \rrbracket, \llbracket \mathsf{z} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{AppRec}}(\llbracket \mathsf{b} \rrbracket)$
- $\alpha = (1)_{2\mathsf{f}}$ and $\llbracket \mathsf{e} \rrbracket = \alpha - \Pi_{\mathsf{Mul}}(\llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{w} \rrbracket, 0)$
- $\llbracket \mathsf{d} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{w} \rrbracket, 0)$
- for $i = 1$ to θ-1 do:
  - $\llbracket \mathsf{d} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{d} \rrbracket, \alpha + \llbracket \mathsf{e} \rrbracket, 2\mathsf{f})$, $\llbracket \mathsf{e} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{e} \rrbracket, \llbracket \mathsf{e} \rrbracket, 2\mathsf{f})$
- $\llbracket \mathsf{d} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{d} \rrbracket, \alpha + \llbracket \mathsf{e} \rrbracket, 2\mathsf{f})$
- return $\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{z} \rrbracket^{\mathbf{B}}$

Figure 3.4: Division.

3.4.3 Shuffle

We next describe the shuffle protocol, Ruffle, that is designed for a single invocation. Note, however, that there arise scenarios where multiple sequential invocations of shuffle are required. For instance, the GraphSC paradigm requires invoking multiple shuffles in a sequential manner with the additional constraint that these shuffles have to be composed sequentially (as elaborated next). Hence, we identify two different scenarios that arise while performing sequential shuffles and design protocols that cater to both separately. We discuss these scenarios first, followed by the protocols.

Independent-Shuffles Let $\pi(\mathsf{T})$ denote the operation of permuting the elements in the ordered set T according to the permutation π. Let $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_m$ be m ordered sets that are required to be shuffled under random secret permutations, say $\pi_1, \pi_2, \ldots, \pi_m$, respectively. Consider the scenario where these shuffles are performed sequentially such that $\pi_{i+1}(\mathsf{T}_{i+1})$ is invoked after $\pi_i(\mathsf{T}_i)$, and T_{i+1} is *independent* of $\pi_i(\mathsf{T}_i)$. We refer to this as the Independent-Shuffles scenario where multiple independent shuffle invocations are required with the constraint that they are invoked sequentially. Our first variant of shuffle protocols, $\mathsf{Ruffle}_{\mathsf{ind}}$, is tailor-made for this scenario. $\mathsf{Ruffle}_{\mathsf{ind}}$ is designed to leverage the independence of m shuffles to perform necessary preprocessing steps in parallel. Thus, our improved online phase supplemented by parallel preprocessing results in improved overall run time, with $\mathsf{Ruffle}_{\mathsf{ind}}$ outperforming [80, 13] for multiple shuffles (i.e., $m \geq 2$). Note that for applications such as the anonymous broadcast (as discussed in §1.1.1.1) that can run perpetually, $\mathsf{Ruffle}_{\mathsf{ind}}$ is apt. Elaborately, in an anonymous broadcast system, client messages are received continuously, and the system is responsible for shuffling every consecutive set of N well-formed messages. Hence, an anonymous broadcast system requires multiple sequential invocations of shuffle, which can be captured by the case of Independent-Shuffles for which $\mathsf{Ruffle}_{\mathsf{ind}}$ is designed.

Composed-Shuffles Unlike the previous scenario of m independent shuffles, in this case, we are interested in determining the composition of m shuffles such that $\mathsf{T}_m = \pi_m(\pi_{m-1}(\ldots \pi_1(\mathsf{T})))$, where T is the input to be shuffled. Such a composition of m shuffles generates a sequence of intermediate shuffled sets, where the i^{th} ordered set is denoted as $\mathsf{T}_i = \pi_i(\ldots \pi_1(\mathsf{T}))$. In this way, the composition of permutations induces a sequential nature to the shuffle invocations with $\pi_{i+1}(\mathsf{T}_i)$ being invoked after $\pi_i(\mathsf{T}_{i-1})$ since $\mathsf{T}_i = \pi_i(\mathsf{T}_{i-1})$. We refer to this as the Composed-Shuffles scenario, where the permutations are required to be composed such that the output of one shuffle invocation is fed as the input to the next. The scenario can indeed be generalized such that π_{i+1} can be invoked on some function f of $\pi_i(\mathsf{T}_i)$, rather than on $\pi_i(\mathsf{T}_i)$ itself. Such a composition of shuffles is heavily used in the GraphSC paradigm for secure computation over graphs.

Recall that $\mathsf{Ruffle}_{\mathsf{ind}}$ is designed to leverage the independence of the m shuffles to facilitate parallel preprocessing. This is in contrast to Composed-Shuffles, where the shuffles are no longer independent. Thus, $\mathsf{Ruffle}_{\mathsf{ind}}$ is not apt for Composed-Shuffles, and we design $\mathsf{Ruffle}_{\mathsf{cmp}}$ to specifically cater to this scenario. As in the case of $\mathsf{Ruffle}_{\mathsf{ind}}$ for Independent-Shuffles, $\mathsf{Ruffle}_{\mathsf{cmp}}$ outperforms [80, 13] with respect to online as well as the overall run time in the Composed-Shuffles scenario. This is achieved by designing $\mathsf{Ruffle}_{\mathsf{cmp}}$ to strategically break the sequential dependence on shuffles in the preprocessing. This enables performing the preprocessing phase in parallel for

the m shuffles. Although $\mathsf{Ruffle}_{\mathsf{cmp}}$ can be used in the scenario of Independent-Shuffles, we note that the design of $\mathsf{Ruffle}_{\mathsf{cmp}}$ for breaking the dependency in the preprocessing comes at the cost of slightly increased preprocessing communication compared to the preprocessing of $\mathsf{Ruffle}_{\mathsf{ind}}$. Hence, the use of $\mathsf{Ruffle}_{\mathsf{ind}}$ is apt for Independent-Shuffles and $\mathsf{Ruffle}_{\mathsf{cmp}}$ for Composed-Shuffles.

3.4.3.1 Ruffle

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

Without loss of generality, let $P_c \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Shuffle}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of the input table T from all parties. Let T_o denote the randomly shuffled input table. $\mathcal{F}_{\mathsf{Shuffle}}$ also receives from $\mathcal{S}$ its $[\![\cdot]\!]^{\mathbf{B}}$-shares of T_o, i.e. it receives $\beta_{\mathsf{T}_o}, [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{ic}, [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{jc}$ where P_i, P_j, P_c denote parties in $\mathcal{P}$.
$\mathcal{F}_{\mathsf{Shuffle}}$ proceeds as follows.

- Reconstruct input T using $[\![\cdot]\!]^{\mathbf{B}}$-shares of the honest parties.
- Sample a random permutation π from the space of all permutations, S_{N} (see §2.5.1) and generate $\mathsf{T}_o = \pi(\mathsf{T})$.
- Set $[\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{ij} = \mathsf{T}_o \oplus \beta_{\mathsf{T}_o} \oplus [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{ic} \oplus [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{jc}$. Let $[\![\mathsf{T}_o]\!]^{\mathbf{B}}_s$ denote the $[\![\cdot]\!]^{\mathbf{B}}$-share of T_o for $P_s \in \mathcal{P}$.
- Send $(\mathsf{Output}, [\![\mathsf{T}_o]\!]^{\mathbf{B}}_s)$ to P_s.

Figure 3.5: Ideal functionality for shuffle.

We begin with defining the ideal functionality for shuffle in Fig. 4.10. Let a table T denote a set of ordered rows where each row consists of an ℓ-bit string. Let N denote the size of T or the number of rows in T (equivalently, T can also be viewed as a vector comprising N ℓ-bit elements). Secure shuffle operation takes as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of table T, i.e., $[\![\cdot]\!]^{\mathbf{B}}$-shares of each of the ℓ-bit string that constitutes a row in T. The output is random $[\![\cdot]\!]^{\mathbf{B}}$-shares of a table T_o, which consists of rows of T in a randomly permuted order. Note that although our shuffle protocol is described to work with Boolean-shared inputs, it can easily be extended to work with arithmetic-shared inputs, as well.

Given that the input table T is $[\![\cdot]\!]^{\mathbf{B}}$-shared, there exists a $\beta_{\mathsf{T}}, \alpha_{\mathsf{T}} \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ such that $\beta_{\mathsf{T}} = \mathsf{T} \oplus \alpha_{\mathsf{T}}$ is held by all parties in $\mathcal{P}$, and α_{T} is $[\cdot]^{\mathbf{B}}$-shared, i.e. $\alpha_{\mathsf{T}} = [\alpha_{\mathsf{T}}]^{\mathbf{B}}_{01} \oplus [\alpha_{\mathsf{T}}]^{\mathbf{B}}_{02} \oplus [\alpha_{\mathsf{T}}]^{\mathbf{B}}_{12}$ where $P_i, P_j \in \mathcal{P}$ hold $[\alpha_{\mathsf{T}}]^{\mathbf{B}}_{ij} \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ (see §2.3 for the sharing semantics of [136]). Let π be the random permutation used to shuffle the rows of T. Observe that, $\mathsf{T}_o = \pi(\mathsf{T}) = \pi(\beta_{\mathsf{T}} \oplus \alpha_{\mathsf{T}}) = \pi(\beta_{\mathsf{T}}) \oplus \pi(\alpha_{\mathsf{T}})$. To respect the $[\![\cdot]\!]^{\mathbf{B}}$-sharing semantics for T_o, we require $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o}$. A naive approach is to thus set $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}})$ and $\alpha_{\mathsf{T}_o} = \pi(\alpha_{\mathsf{T}})$ since this would satisfy $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o} = \pi(\mathsf{T})$. Observe, however, that this approach leaks the secret permutation π to all the parties since

they all hold β_{T} and will now also hold $\pi(\beta_{\mathsf{T}})$ on clear, from which one can recover π. To keep π private, we observe that it suffices to mask $\pi(\beta_{\mathsf{T}})$ with some randomness $\mathsf{R} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, and hence, define this masked value as β_{T_o}, i.e., $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$. Further, to ensure that the relation $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o}$ holds, we redefine $\alpha_{\mathsf{T}_o} = \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$. Thus, given $[\![\mathsf{T}]\!]^{\mathbf{B}}$, our goal is to generate $[\cdot]^{\mathbf{B}}$-shares of α_{T_o}, and ensure that all parties hold β_{T_o}. Observe that $[\alpha_{\mathsf{T}_o}]^{\mathbf{B}} = [\pi(\alpha_{\mathsf{T}})]^{\mathbf{B}} \oplus [\mathsf{R}]^{\mathbf{B}}$. Looking ahead R gets defined during the generation of β_{T_o}. Thus, in what follows, we first describe steps to generate $[\pi(\alpha_{\mathsf{T}})]^{\mathbf{B}}$, followed by steps to generate β_{T_o} and then $[\mathsf{R}]^{\mathbf{B}}$.

Generation of $[\pi(\alpha_{\mathsf{T}})]^{\mathbf{B}}$ Since α_{T} is independent of the input T, it is generated during a preprocessing phase. Hence, $[\cdot]^{\mathbf{B}}$-shares of $\alpha' = \pi(\alpha_{\mathsf{T}})$ where π is a random secret permutation (independent of T), can be generated during preprocessing. For this, we employ the protocol of [13]. The protocol takes as input $[\cdot]^{\mathbf{B}}$-shares of a table, and outputs $[\cdot]^{\mathbf{B}}$-shares of the table shuffled using a random secret permutation π. It also outputs a flag that indicates the correctness of $[\cdot]^{\mathbf{B}}$-shares of shuffled table[1].

Recall from §2.5.2, the protocol of [13] relies on the semi-honest 3PC shuffle protocol from [145] which guarantees privacy against a malicious adversary. [13] then augments this with a robust Set-Equality protocol to verify the correctness of the semi-honest shuffle. The semi-honest shuffle comprises three invocations of Shuffle-Pair protocol. In each instance of Shuffle-Pair, a random permutation is applied to the input (of the Shuffle-Pair), where the permutation is known to a distinct pair of parties and is hidden from the third. The output of the current Shuffle-Pair is fed as input to the next Shuffle-Pair. The composition of all three permutations, thus, makes up the random secret permutation used to shuffle the input table. Since each party is aware of only two permutations, the final permutation remains private. Each invocation of Shuffle-Pair is followed by a Set-Equality protocol which outputs a $\mathsf{flag} \in \{0, 1\}$ indicating whether the table output by the Shuffle-Pair is indeed a random permutation of the input to this Shuffle-Pair. In this way, the output of the shuffle protocol is guaranteed to be correct if all instances of Shuffle-Pair are verified to be correct.

Let $\pi_{12}, \pi_{01}, \pi_{02}$ denote the three permutations used in the three Shuffle-Pair instances, where π_{ij} is held by $P_i, P_j \in \mathcal{P}$. Applying the protocol of [13] on $[\alpha_{\mathsf{T}}]^{\mathbf{B}}$ outputs $[\pi(\alpha_{\mathsf{T}})]^{\mathbf{B}}$ and a flag. Here, we let $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$ where $\circ$ denotes composition operation, and flag indicates correctness of $[\pi(\alpha_{\mathsf{T}})]^{\mathbf{B}}$.

[1]We choose [13] over [80] since the protocol in [13] follows the same $[\cdot]^{\mathbf{B}}$-sharing semantics as required for α, which is not the case in the protocol of [80]. Hence, the use of [13] yields an efficient preprocessing phase. Towards the end of §3.4.3.1, we also showcase how to get GOD for the protocol of [13].

Generation of $\beta_{\mathsf{T}_\circ} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$ As part of the Shuffle-Pair instances performed during the preprocessing, parties generate $\pi_{12}, \pi_{01}, \pi_{02}$. The goal now is to generate $\beta_{\mathsf{T}_\circ} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$ where $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$. Observe that, unlike during preprocessing, the table to be shuffled is now held by all three parties on clear, while the permutation π is still private. Further, each party misses exactly one permutation that is held by the other two parties. We leverage these observations in designing our shuffle protocol to attain a highly efficient online phase. We explain case-by-case how each $P_i \in \mathcal{P}$ obtains $\beta_{\mathsf{T}_\circ}$.

Generating $\beta_{\mathsf{T}_\circ}$ towards P_1 Recall that P_1 misses π_{02}. If P_1 is given $\pi_{02}(\beta_{\mathsf{T}})$, it can locally compute $\pi(\beta_{\mathsf{T}}) = \pi_{12} \circ \pi_{01}(\pi_{02}(\beta_{\mathsf{T}}))$ using its knowledge of $\pi_{12} \circ \pi_{01}$. However, as mentioned earlier, since P_1 holds β_{T} on clear, knowledge of $\pi_{02}(\beta_{\mathsf{T}})$ leaks the permutation π_{02} to it. Hence, we instead provide it with $\pi_{02}(\beta_{\mathsf{T}}) \oplus \mathsf{R}'$, where the randomness R' masks $\pi_{02}(\beta_{\mathsf{T}})$ and prevents leakage of π_{02}. For this, observe that π_{02} is held by both P_0, P_2. We let P_0, P_2 sample a random $\mathsf{R}_{02} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, and compute and send $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ to P_1. Here, $\pi_{02}(\mathsf{R}_{02})$ serves as the random mask R'. Further, note that since at most one among P_0, P_2 can be malicious, making both send the value to P_1 enables the latter to check the consistency of the received messages and detect misbehaviour, if any. Since the message from the second sender only aids in verifying the consistency of the received messages, to save on communicating an entire table, it suffices for one sender to send the value and the other to send the hash of it[2] . On receiving a consistent $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ (which also guarantees its correctness, as otherwise, the received messages would have been inconsistent), P_1 can compute $\beta_{\mathsf{T}_\circ}$ using the received value and the knowledge of permutations π_{12}, π_{01}. Note that $\pi_{02}(\mathsf{R}_{02})$ serves as a mask to hide π from P_1. Looking ahead, similar masks are required in $\beta_{\mathsf{T}_\circ}$ to keep π hidden from P_2 and P_0. This results in additionally introducing the random masks $\pi_{12}(\pi_{01}(\mathsf{R}_{01}))$ and $\pi_{12}(\mathsf{R}_{12})$, respectively. To ensure that all parties have the same $\beta_{\mathsf{T}_\circ}$ and use the same randomness for masking $\pi(\beta_{\mathsf{T}})$, $\beta_{\mathsf{T}_\circ}$ is defined as

$$\begin{aligned} \beta_{\mathsf{T}_\circ} &= \pi_{12}(\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01}) \oplus \mathsf{R}_{12}) \\ &= \pi(\beta_{\mathsf{T}}) \oplus \pi(\mathsf{R}_{02}) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01})) \oplus \pi_{12}(\mathsf{R}_{12}) \end{aligned} \tag{3.1}$$

where $\mathsf{R}_{12}, \mathsf{R}_{01} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, and R_{ij} is jointly sampled by $P_i, P_j \in \mathcal{P}$. At the end of first round, since P_1 holds $\mathsf{R}_{12}, \mathsf{R}_{01}, \pi_{12}, \pi_{01}$, and $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$, it can compute $\beta_{\mathsf{T}_\circ}$ using Equation (3.1).

Generating $\beta_{\mathsf{T}_\circ}$ towards P_2 Observe that P_2 lacks π_{01} that prevents it from computing $\pi_{12}(\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})))$. On the other hand, if provided with the value $\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}))$,

[2]When performing multiple shuffle instances, the cost of sending a hash can be amortized by sending a single hash for messages corresponding to multiple shuffles.

then P_2 can obtain $\pi_{12}(\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})))$ by applying π_{12} on it. However, similar to the case described earlier, this leaks the permutation π_{01} to P_2. To fix this leakage, we first mask $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ with the random value R_{01} and then apply π_{01} on this masked value and communicate it to P_2. This justifies the need for the term $\pi_{12}(\pi_{01}(\mathsf{R}_{01}))$ in Equation (3.1). The value to be communicated, $\delta_{12} = \pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01})$ can be computed by P_0 and sent to P_2, since P_0 possesses the required values. Since we want to maintain the invariant that each message is communicated by two senders to aid in the verification of correctness at the receiver, we require P_1 to also send a hash of this message to P_2. Although P_1 does not possess R_{02} and π_{02} required to compute δ_{12}, observe that it receives $\delta_{02} = \pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ in the first round, and can compute and send the hash of $\delta_{12} = \pi_{01}(\delta_{02} \oplus \mathsf{R}_{01})$ in the next round to P_2. On receiving these values, P_2 can thus verify its correctness and then use Equation (3.1) to compute $\beta_{\mathsf{T}_\circ}$.

Generating $\beta_{\mathsf{T}_\circ}$ towards P_0 Given that $\beta_{\mathsf{T}_\circ}$ is made available to both P_1, P_2, they can send it to P_0 (one sends the value, the other sends the hash). This completes the generation of $\beta_{\mathsf{T}_\circ}$ towards all the parties. Observe the need for using R_{12} as a mask while computing $\beta_{\mathsf{T}_\circ}$. Analogous to the cases for P_1, P_2, absence of R_{12} leaks the permutation π_{12} to P_0. Further, note that although P_2 can compute the correct $\beta_{\mathsf{T}_\circ}$ only after the second round, it receives δ_{12} required for computing $\beta_{\mathsf{T}_\circ}$ in the first round itself. Hence, communication of $\beta_{\mathsf{T}_\circ}$ from P_1, P_2 towards P_0 can happen in the second round.

A pictorial view of the messages exchanged is given in Fig. 3.6.

Generation of $[\alpha_{\mathsf{T}_\circ}]^{\mathbf{B}} = [\pi(\alpha_{\mathsf{T}})]^{\mathbf{B}} \oplus [\mathsf{R}]^{\mathbf{B}}$ Subsequent to the above discussion and as evident from Equation (3.1), R is defined as $\mathsf{R} = \pi(\mathsf{R}_{02}) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01})) \oplus \pi_{12}(\mathsf{R}_{12})$, whose $[\cdot]^{\mathbf{B}}$-shares are required to be generated in the preprocessing phase. Observe that

$$\begin{aligned}
[\alpha_{\mathsf{T}_\circ}]^{\mathbf{B}} &= [\pi(\alpha_{\mathsf{T}})]^{\mathbf{B}} \oplus [\mathsf{R}]^{\mathbf{B}} \\
&= [\pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}})))]^{\mathbf{B}} \oplus [\pi_{12}(\pi_{01}(\pi_{02}(\mathsf{R}_{02})))]^{\mathbf{B}} \\
&\qquad \oplus [\pi_{12}(\pi_{01}(\mathsf{R}_{01}))]^{\mathbf{B}} \oplus [\pi_{12}(\mathsf{R}_{12})]^{\mathbf{B}} \qquad (3.2)
\end{aligned}$$

P_1, P_2 hold $\pi_{12}(\mathsf{R}_{12})$ on clear. Hence, as described in §2.3, $[\pi_{12}(\mathsf{R}_{12})]^{\mathbf{B}}$ can be generated non-interactively. A naive approach of generating $[\cdot]^{\mathbf{B}}$-shares of remainder terms requires three invocations of Shuffle-Pair for generating $[\pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}})))]^{\mathbf{B}}$, three for generating $[\pi_{12}(\pi_{01}(\pi_{02}(\mathsf{R}_{02})))]^{\mathbf{B}}$, and two for $[\pi_{12}(\pi_{01}(\mathsf{R}_{01}))]^{\mathbf{B}}$. However, all these remainder terms in Equation (3.2), need an application of π_{01} followed by an application of π_{12}. Hence, instead of separately computing these terms via multiple Shuffle-Pair instances, we club these terms

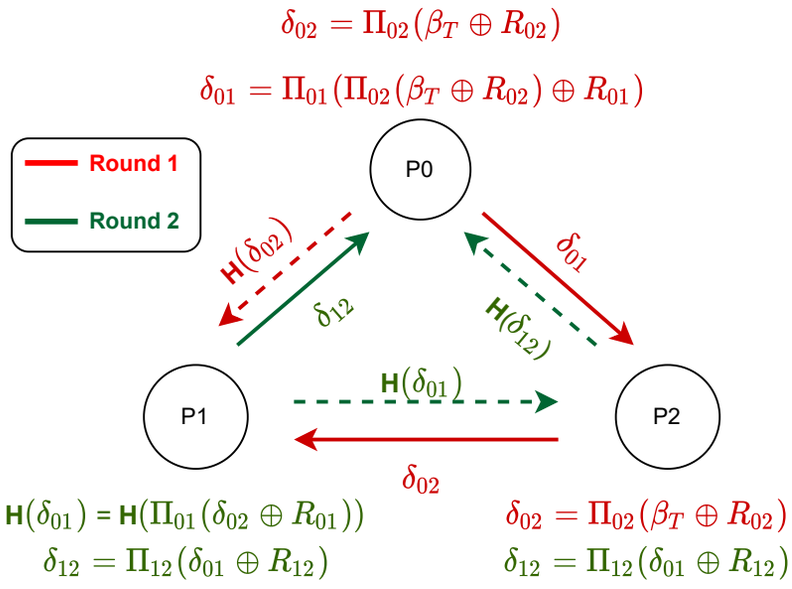


Figure 3.6: Online phase of Ruffle.

together in such a way that we require only three Shuffle-Pair instances to compute $[\alpha_{\mathsf{T}_\circ}]^{\mathbf{B}}$. Elaborately, given that R_{02} is held by P_0, P_2 on clear, parties can non-interactively generate its $[\cdot]^{\mathbf{B}}$-shares. Further, given $[\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}]^{\mathbf{B}} = [\alpha_{\mathsf{T}}]^{\mathbf{B}} \oplus [\mathsf{R}_{02}]^{\mathbf{B}}$, parties invoke Shuffle-Pair with π_{02} as the secret permutation to generate $[\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02})]^{\mathbf{B}}$. Since $[\mathsf{R}_{01}]^{\mathbf{B}}$ can also be generated non-interactively, the remainder terms in (3.2) can be alternatively expressed as,

$$\begin{aligned}&\pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}}))) \oplus \pi_{12}(\pi_{01}(\pi_{02}(\mathsf{R}_{02}))) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01}))\\ &= \pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01})) = \gamma\end{aligned}$$

Hence, given $[\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01}]^{\mathbf{B}} = [\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02})]^{\mathbf{B}} \oplus [\mathsf{R}_{01}]^{\mathbf{B}}$, one can apply two invocations of Shuffle-Pair with π_{01}, π_{12} to generate $[\gamma]^{\mathbf{B}}$, as required for generating $[\alpha_{\mathsf{T}_\circ}]^{\mathbf{B}}$.

Let $[\rho]^{\mathbf{B}} = \mathsf{Shuffle\text{-}Pair}([\mathsf{T}]^{\mathbf{B}}, \pi_{ij})$ denote the application of π_{ij} on $[\mathsf{T}]^{\mathbf{B}}$ to obtain $[\rho]^{\mathbf{B}}$ where $\rho = \pi_{ij}(\mathsf{T})$ and the parties P_i, P_j are the pair who knows π_{ij} on clear. If $[\mathsf{T}_1]^{\mathbf{B}}, [\mathsf{T}_2]^{\mathbf{B}}$ denote the input and output of a Shuffle-Pair instance, let $\mathsf{Set\text{-}Equality}([\mathsf{T}_1]^{\mathbf{B}}, [\mathsf{T}_2]^{\mathbf{B}})$ output $\mathsf{flag} = 0$ if Shuffle-Pair was performed correctly, and $\mathsf{flag} = 1$ otherwise. The steps for generating $\alpha_{\mathsf{T}_\circ}$ is summarised in Fig. 3.7.

1. $[\rho_2]^{\mathbf{B}} = \mathsf{Shuffle\text{-}Pair}([\rho_1]^{\mathbf{B}}, \pi_{02})$ where $\rho_1 = \alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}$ and $\mathsf{flag}_{02} = \mathsf{Set\text{-}Equality}([\rho_1]^{\mathbf{B}}, [\rho_2]^{\mathbf{B}})$

2. $[\rho_4]^{\mathbf{B}} = \mathsf{Shuffle\text{-}Pair}([\rho_3]^{\mathbf{B}}, \pi_{01})$ where $\rho_3 = \rho_2 \oplus \mathsf{R}_{01}$ and $\mathsf{flag}_{01} = \mathsf{Set\text{-}Equality}([\rho_3]^{\mathbf{B}}, [\rho_4]^{\mathbf{B}})$

3. $[\rho_5]^{\mathbf{B}} = \mathsf{Shuffle\text{-}Pair}([\rho_4]^{\mathbf{B}}, \pi_{12})$, $\mathsf{flag}_{12} = \mathsf{Set\text{-}Equality}([\rho_4]^{\mathbf{B}}, [\rho_5]^{\mathbf{B}})$

4. Set $[\alpha_{\mathsf{T}_o}]^{\mathbf{B}} = [\rho_5]^{\mathbf{B}} \oplus [\pi_{12}(\mathsf{R}_{12})]^{\mathbf{B}}$

Figure 3.7: Generation of $[\alpha_{\mathsf{T}_o}]^{\mathbf{B}}$ by parties in $\mathcal{P}$.

Guaranteeing output delivery Note that in the solution described above, an adversary can misbehave, resulting in an `abort` (i.e., failure of shuffle). However, to attain GOD and obtain as output the randomly shuffled input table irrespective of the adversarial behaviour, one can proceed as follows. Inspired by the techniques of [136, 138, 30, 36], we rely on a trusted third party (TTP) based approach. Elaborately, if shuffle fails, we work towards identifying an honest party in $\mathcal{P}$ that is designated as a TTP. Parties robustly reconstruct the input table to TTP, which performs the shuffle operation on the clear table and sends the output (shares of the randomly shuffled input table) to all. We next describe how a TTP can be identified in preprocessing and online phases whenever shuffle fails.

Identifying a TTP *if shuffle fails during preprocessing phase.* The preprocessing phase involves three sequential invocations of the semi-honest Shuffle-Pair protocol where in each invocation, only two parties communicate a message (see §2.5.2 for details). Each invocation of Shuffle-Pair is followed by a robust Set-Equality protocol to verify the correctness of the Shuffle-Pair, which outputs a flag indicating that shuffle failed if some misbehaviour was detected in this Shuffle-Pair instance. We make the following observation that aids in identifying a TTP: If any invocation of Set-Equality outputs a flag indicating that Shuffle-Pair fails, it must be due to misbehaviour by one of the two (communicating) parties in the corresponding Shuffle-Pair instance. This is because Set-Equality protocol is robust against any misbehaviour (owing to the use of a robust 3PC for the same), and hence, shuffle can fail only due to misbehaviour in Shuffle-Pair. Further, since at most one among the three parties is malicious, this guarantees that the (non-communicating) residual party is honest and can be designated as the TTP.

Protocol $\Pi_{\mathsf{Shuffle}}\left([\![\mathsf{T}]\!]^{\mathbf{B}}\right)$

Preprocessing:

– Each pair of parties $P_i, P_j \in \mathcal{P}$ non-interactively sample $\mathsf{R}_{ij} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and random permutations π_{ij}.

– P_1, P_2 compute $\pi_{12}(\mathsf{R}_{12})$, and parties generate its $[\cdot]^{\mathbf{B}}$-shares, non-interactively.

– Parties in $\mathcal{P}$ generate $[\cdot]^{\mathbf{B}}$-shares of $\mathsf{R}_{01}, \mathsf{R}_{02}$, non-interactively.

– Parties in $\mathcal{P}$ follow the steps in Fig. 3.7 to generate $[\alpha_{\mathsf{T}_\circ}]^{\mathbf{B}}$.

– *Identifying* TTP *when shuffle fails:* If flag_{ij} indicates a failure, all parties set TTP to be the non-communicating party in the corresponding Shuffle-Pair protocol. When multiple flag_{ij} indicates failure, break tie deterministically and use one flag_{ij}.

Online:

– Shuffle (Round 1):

- P_0, P_2 compute $\delta_{02} = \pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$. P_2 sends δ_{02} to P_1. P_0 sends $\mathsf{H}(\delta_{02})$ to P_1, where H is a collision-resistant hash function.
- P_0 computes and sends $\delta_{01} = \pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01})$ to P_2.

– Shuffle (Round 2):

- P_1 computes and sends $\mathsf{H}(\delta_{01}) = \mathsf{H}(\pi_{01}(\delta_{02} \oplus \mathsf{R}_{01}))$ to P_2.
- P_1, P_2 compute $\delta_{12} = \pi_{12}(\delta_{01} \oplus \mathsf{R}_{12})$.
- P_1 sends δ_{12} and P_2 sends $\mathsf{H}(\delta_{12})$ to P_0.

– Verification (Round 3)[a]: For each receiver $P_i \in \mathcal{P}$, let P_j, P_k denote the senders. Let P_j send the message and P_k send its hash. P_i checks if the received values are consistent. If not, it broadcasts ("accuse", P_j, P_k, c_j, c_k), where $c_j = \mathsf{H}(x)$, such that x and c_k are the values sent by P_j and P_k, respectively.

– Verification and TTP Identification (Round 4): Consider the first instance when a party P_i broadcasts ("accuse", P_j, P_k, c_j, c_k).

- If $c_j = c_k$, set $\mathsf{TTP} = P_j$.
- Else if c_j is different from the hash of the value sent by P_j to P_i, then P_j broadcasts ("accuse", P_i). Set $\mathsf{TTP} = P_k$. The above steps follow analogously for P_k.
- Else if $c_j \neq c_k$ and neither P_j nor P_k accuses P_i, set $\mathsf{TTP} = P_i$.

– One-time computation through TTP: If TTP is set, all parties robustly reconstruct the input table towards the TTP, who randomly shuffles the input and sends the shuffled table to all parties.

[a]Note that these can be performed as soon as the messages required for detecting inconsistency are available.

Figure 3.8: Secure shuffle.

Identifying a TTP *if shuffle fails during online phase.* Each of the three messages that are exchanged in the online phase have the following communication pattern. There exist two senders who possess the message to be sent to the receiver, where one sender sends the message while the other sends the hash of it. Since this resembles the communication pattern of [136, 61], we use the techniques therein to identify a TTP, if any party receives an inconsistent (message, hash) pair. At a high level, if the received message and hash do not match at the receiver, it broadcasts a complaint accusing the senders. It also broadcasts the received messages. This is followed by the senders broadcasting a complaint against the receiver if the latter's broadcast message was inconsistent with the senders' sent message. Depending on the publicly available complaints, parties can unanimously determine a pair of parties that are in conflict with each other, one of which is guaranteed to be corrupt. Due to at most one malicious corruption among the three parties, the third party that is not a part of this conflict is guaranteed to be honest and can be designated as the TTP.

The formal steps are provided in Fig. 3.8, and correctness follows from [61][3].

3.4.3.2 $\mathsf{Ruffle}_{\mathsf{ind}}$

Ruffle protocol described in §3.4.3, was for a single invocation of shuffle. For the scenario of Independent-Shuffles, where m independent shuffles are required to be performed sequentially, we design $\mathsf{Ruffle}_{\mathsf{ind}}$ as follows. In its preprocessing phase, $\mathsf{Ruffle}_{\mathsf{ind}}$ performs m instances of the preprocessing of Ruffle in parallel, whereas for its online phase, it sequentially executes the online phase of Ruffle m times. As can be seen in Table 3.2, $\mathsf{Ruffle}_{\mathsf{ind}}$ results in having a better complexity than that of [80, 13] for multiple sequential shuffle invocations.

3.4.3.3 $\mathsf{Ruffle}_{\mathsf{cmp}}$

For scenarios that demand the composition of, say m, shuffles (i.e., Composed-Shuffles), observe that the preprocessing phase of $\mathsf{Ruffle}_{\mathsf{ind}}$, which comprises m instances of the preprocessing of Ruffle, can no longer execute in parallel, but will have to be performed sequentially. This is because in Composed-Shuffles, the output of one shuffle operation, say T_1, constitutes the input to a subsequent shuffle operation, which, say outputs $\mathsf{T}_2 = \pi(\mathsf{T}_1)$. Hence, once α_{T_1} is generated as output from the first (preprocessing phase) instance of Ruffle, only then can $\alpha_{\mathsf{T}_2} = \pi(\alpha_{\mathsf{T}_1}) \oplus \mathsf{R}$ be generated (see §3.4.3.1 for the definition of α_{T}). This sequential dependency present in the preprocessing phase of $\mathsf{Ruffle}_{\mathsf{ind}}$, when deployed in the case of Composed-Shuffles, makes its run

[3]These steps can be optimized by using the technique described in [136].

time proportional to the number of sequential shuffles. However, it is desirable to facilitate the generation of necessary preprocessing data in parallel and hence, decouple the dependency between the generation of preprocessing data and the pattern of shuffle invocations. This can aid in significantly reducing preprocessing phase's cost. Hence, in the following, we design an alternative protocol $\mathsf{Ruffle}_{\mathsf{cmp}}$, that breaks this dependence and is tailor-made to handle Composed-Shuffles.

Let T be the input table which has to be shuffled to obtain $\mathsf{T}_o = \pi(\mathsf{T})$. In Ruffle (Fig. 3.8), $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o}$ where $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$, $\alpha_{\mathsf{T}_o} = \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$, and $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$, $\mathsf{R} = \pi(\mathsf{R}_{02}) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01})) \oplus \pi_{12}(\mathsf{R}_{12})$. To break the dependency and ensure that α_{T_o} can be generated independently of α_{T} we proceed as follows. Let $\alpha'_{\mathsf{T}_o}, \beta'_{\mathsf{T}_o}$ be the newly defined values such that $\mathsf{T}_o = \alpha'_{\mathsf{T}_o} \oplus \beta'_{\mathsf{T}_o}$, where α'_{T_o} is the decoupled equivalent of α_{T_o}. We let parties non-interactively sample $[\cdot]^{\mathbf{B}}$-shares of a random $\alpha'_{\mathsf{T}_o} \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ during preprocessing. Having generated α'_{T_o} this way, we need to define β'_{T_o} to ensure that $\mathsf{T}_o = \beta'_{\mathsf{T}_o} \oplus \alpha'_{\mathsf{T}_o}$ holds. Hence, we define $\beta'_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R} \oplus \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R} \oplus \alpha'_{\mathsf{T}_o}$. This is because recall that $\mathsf{T}_o = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R} \oplus \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$ (where R serves as a random mask for $\pi(\beta_{\mathsf{T}})$). We next describe how to generate this β'_{T_o}.

Let $\beta'_{\mathsf{T}_o} = \mathsf{B}_1 \oplus \mathsf{B}_2$, where $\mathsf{B}_1 = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$ and $\mathsf{B}_2 = \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R} \oplus \alpha'_{\mathsf{T}_o}$. In the preprocessing phase, observe that $[\cdot]^{\mathbf{B}}$-shares of $\pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$ can be generated as described in §3.4.3.1. Thus, parties can compute $[\mathsf{B}_2]^{\mathbf{B}} = [\pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}]^{\mathbf{B}} \oplus [\alpha'_{\mathsf{T}_o}]^{\mathbf{B}}$ and reconstruct B_2 towards all parties. In the online phase, to generate β'_{T_o}, observe that parties can generate B_1, as described in §3.4.3.1. Given B_2 generated during preprocessing, parties set $\beta'_{\mathsf{T}_o} = \mathsf{B}_1 \oplus \mathsf{B}_2$. This completes the generation of $[\![\mathsf{T}_o]\!]^{\mathbf{B}}$. In comparison to $\mathsf{Ruffle}_{\mathsf{ind}}$ this protocol, $\mathsf{Ruffle}_{\mathsf{cmp}}$, only requires an additional reconstruction of B_2 (for each shuffle instance) towards all the parties during the preprocessing phase.

In summary, when dealing with Composed-Shuffles, the α and β values need to be redefined, and the computation proceeds as follows. Assuming that we are interested in computing the output shuffled table T_n defined as $\mathsf{T}_n = \pi_n(\pi_{n-1}(\ldots \pi_1(\mathsf{T}_0)))$ and T_0 is $[\![\cdot]\!]^{\mathbf{B}}$-shared, let each intermediate shuffled table $\mathsf{T}_i = \pi_i(\ldots \pi_1(\mathsf{T}_0))$. As per the newly defined values, $\mathsf{T}_i = \alpha'_{\mathsf{T}_i} \oplus \beta'_{\mathsf{T}_i}$, for each $i \in \{1, \ldots, n\}$. Each of the α'_{T_i} are randomly sampled and hence $[\cdot]^{\mathbf{B}}$-shares of each $\pi_i\left(\alpha'_{\mathsf{T}_{i-1}}\right) \oplus \mathsf{R}_i$ can be generated in parallel. Thus, the B_2 component of each β'_{T_i} can be computed as $[\mathsf{B}_2]^{\mathbf{B}} = \left[\pi_i\left(\alpha'_{\mathsf{T}_{i-1}}\right) \oplus \mathsf{R}\right]^{\mathbf{B}} \oplus \left[\alpha'_{\mathsf{T}_i}\right]^{\mathbf{B}}$ in parallel during the preprocessing phase and reconstructed towards all the parties. Each β'_{T_i} is generated sequentially in the online phase by computing $\mathsf{B}_1 = \pi_i\left(\beta'_{\mathsf{T}_{i-1}}\right) \oplus \mathsf{R}_i$ and $\beta'_{\mathsf{T}_i} = \mathsf{B}_1 \oplus \mathsf{B}_2$.

3.4.3.4 Comparison of the shuffle protocols

When a single shuffle is required (i.e., $m = 1$), both $\mathsf{Ruffle}_{\mathsf{ind}}$ and $\mathsf{Ruffle}_{\mathsf{cmp}}$ can be used and both improve over the existing works with respect to the online cost. Further, $\mathsf{Ruffle}_{\mathsf{ind}}$ outperforms $\mathsf{Ruffle}_{\mathsf{cmp}}$ due to the latter's increased communication cost. A comparison of our shuffle protocols with that of [80, 13] is given in Table 3.2. Since all protocols have a common structure for the online phase that comprises steps for semi-honest shuffle followed by its verification, the cost of these is reported in Table 3.2. As shown in the table, $\mathsf{Ruffle}_{\mathsf{ind}}$ becomes prohibitively expensive for Composed-Shuffles case, because it incurs an m factor inflation in the preprocessing round complexity (see highlighted entry). Similarly, $\mathsf{Ruffle}_{\mathsf{cmp}}$ is inapt for Independent-Shuffles, due to inflation of $3N\ell m$ bits in its preprocessing communication complexity (see highlighted entry).

Scenario	Protocol	Online				Preprocessing		Security
		Semi-honest shuffle		Verification				
		Rounds	Comm. (bits)	Rounds	Comm. (bits)	Rounds	Comm. (bits)	
Independent-Shuffles/ Composed-Shuffles	[80]‡	$2m$	$2\mathsf{N}(2\ell+3p)m$	$4m$	$2\mathsf{N}(\ell+2p)m+4pm$	2	$\mathsf{N}(2\ell+9p)m+4pm$	Abort
Independent-Shuffles/ Composed-Shuffles	[13]	$3m$	$6\mathsf{N}(\ell+\kappa)m$	$3(2+\log_2\kappa)m$	$(6\mathsf{N}\kappa+3\kappa)m$	*		Abort ‡‡
Independent-Shuffles	$\mathsf{Ruffle}_{\mathsf{ind}}$	$2m$	$3\mathsf{N}\ell m$	2	3080†	$5+\log_2\kappa$	$(6\mathsf{N}\ell+12\mathsf{N}\kappa+3\kappa)m$**	GOD
Composed-Shuffles	$\mathsf{Ruffle}_{\mathsf{cmp}}$					$6+\log_2\kappa$***	$(9\mathsf{N}\ell+12\mathsf{N}\kappa+3\kappa)m$	

N: number of elements to be shuffled, where each element is an ℓ-bit string; $\kappa(=48)$: statistical security parameter; p: order of field. [80] uses a 128-bit field
‡: Although [80] does not have an explicit preprocessing phase, we observe that the shuffle correlation and other randomness can be preprocessed. Hence, we explicitly distinguish between preprocessing and online to provide a fair comparison.
*: The preprocessing for [13] only involves the generation of randomness, non-interactively. ‡‡: See §2.5.2 for a discussion on security guarantees of [13].
†: The communication for verification comprises broadcasting 2 hashes and 2 bits, the cost of which gets amortized over multiple shuffle instances.
: $\mathsf{Ruffle}_{\mathsf{cmp}}$ for Independent-Shuffles additionally requires communicating $3\mathsf{N}\ell m$ bits. *: $\mathsf{Ruffle}_{\mathsf{ind}}$ for Composed-Shuffles instead requires $(5+\log_2\kappa)m$ rounds.

Table 3.2: Round complexity and communication (amortized) of various shuffle protocols for m invocations.

3.5 Privacy-preserving HKPR

To measure the similarity of nodes prior to performing local clustering, we rely on the graph propagation metric of heat kernel PageRank (HKPR). The HKPR metric is favourable since it is known to converge fast and produce good quality cluster [54, 53, 230]. Hence, we provide a secure local clustering algorithm that uses HKPR as the graph propagation metric (or the similarity measure) and can be accomplished in two steps. Given a seed node as input, we

first compute the HKPR values of all vertices in the graph, assuming the seed node to be the source. The computed HKPR values are then used as input to a clustering algorithm to find a suitable cluster around the seed vertex. In this section, we describe how HKPR values can be generated securely, followed by describing the secure clustering algorithm in §3.6. Although the protocols are described as linear round solutions, they can easily be translated to a sub-linear round solution following the techniques of [179] (see §2.6.1). We begin by describing the cleartext algorithm for HKPR, followed by a secure protocol for its data-oblivious variant. Note that the translation of cleartext to the message-passing algorithm is accounted for implicitly by identifying (in place) the cleartext algorithm components that can benefit from computation via GraphSC.

3.5.1 The cleartext algorithm

For a seed node $s \in \mathsf{V}$, the HKPR value $\boldsymbol{\rho}[\mathsf{v}]$ of a node v, captures the probability of a heat kernel random walk from s terminating at v. Thus, the graph propagation equation to compute HKPR of all nodes with respect to a specific seed node can be given as a $|\mathsf{V}|$-dimensional vector as :

$$\boldsymbol{\rho} = \sum_{i=0}^{\infty} \mathsf{w}_i \cdot \left(\mathbf{A}\mathbf{D}^{-1}\right)^i \cdot \mathbf{x}, \tag{3.3}$$

where $\mathsf{w}_i = \frac{e^{-t}t^i}{i!}$ is the weight with t being the Poisson distribution parameter, $\mathbf{A}$ is the adjacency matrix of the input graph, $\mathbf{D}$ is degree matrix of the input graph with i^{th} diagonal entry storing degree of i^{th} vertex in the graph (deg_i), and $\mathbf{x}$ is the signal vector (of dimension $|\mathsf{V}|$), which is a one-hot encoding of s, that is, entry at position s is set to 1 (i.e., $\mathbf{x}[s] = 1$) and the rest are set to 0.

We rely on the basic propagation algorithm in [222] that approximates $\boldsymbol{\rho}$ for each vertex by iteratively computing the sum in Equation (3.3) only up to L terms. In fact, the algorithm in [222] is capable of computing various other graph propagation metrics such as PageRank, Personalized PageRank, L-hop transition probability, etc. Each of these metrics can be computed using the generalized graph propagation equation (Equation (3.4)) and by appropriately parameterizing it. Further details regarding this are provided in §3.5.4. Steps for computing propagation vector $\boldsymbol{\rho}$ in Equation (3.4) via graph propagation algorithm are described in Algorithm 2. The intuition is described next.

$$\boldsymbol{\rho} = \sum_{i=0}^{\infty} \mathsf{w}_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \mathbf{x} \tag{3.4}$$

Algorithm 2: Graph propagation

Input: Undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, signal vector $\mathbf{x}$ of dimension $|\mathsf{V}|$, number of iterations L, parameters of graph propagation equation a, b, weights $\{\mathsf{w}_j\}_{j=0}^{\mathsf{L}}$, partial weights $\{\mathsf{Y}_j\}_{j=0}^{\mathsf{L}}$
Output: Estimated propagation vector $\boldsymbol{\rho}$ of dimension $|\mathsf{V}|$

$\mathbf{r}^{(0)} = \mathbf{x}$, $\boldsymbol{\rho} = \mathbf{0}$ (the all zero vector);
for $i = 0$ *to* $\mathsf{L} - 1$ **do**
  for *each* $\mathsf{u} \in \mathsf{V}$ *with non-zero* $\mathbf{r}^{(i)}[\mathsf{u}]$ **do**
    for *for each* $\mathsf{v} \in \mathsf{Nb}_{\mathsf{u}}$ **do**
      $\mathbf{r}^{(i+1)}[\mathsf{v}] = \mathbf{r}^{(i+1)}[\mathsf{v}] + \left(\frac{\mathsf{Y}_{i+1}}{\mathsf{Y}_i}\right) \frac{\mathbf{r}^{(i)}[\mathsf{u}]}{(\mathsf{deg}_{\mathsf{v}})^a (\mathsf{deg}_{\mathsf{u}})^b}$
    end
    $\mathbf{q}^{(i)}[\mathsf{u}] = \mathbf{q}^{(i)}[\mathsf{u}] + \left(\frac{\mathsf{w}_i}{\mathsf{Y}_i}\right) \mathbf{r}^{(i)}[\mathsf{u}]$
  end
  $\boldsymbol{\rho} = \boldsymbol{\rho} + \mathbf{q}^{(i)}$
end

The algorithm requires that the weights add up to 1, and hence the weight entries in the generalized graph propagation equation (Equation (3.4)) are normalised in such a way that $\mathsf{w}_i = \frac{\mathsf{w}_i}{\Sigma_{i=0}^{\infty} \mathsf{w}_i}$. The i^{th} iteration of the algorithm computes the i^{th} term in the infinite sum of $\boldsymbol{\rho}$. To compute this efficiently, the authors in [222] make the following observation that any two consecutive terms i and $i+1$ in the infinite sum in Equation (3.4) have a lot of computation in common. To identify a recursive structure and avoid unnecessary re-computation, two vectors known as residue vector $\mathbf{r}$ and reserve vector $\mathbf{q}$ are defined as follows:

$$\mathbf{r}^{(i)} = \mathsf{Y}_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \mathbf{x}, \qquad \mathbf{q}^{(i)} = \mathsf{w}_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \mathbf{x}$$

Here, the partial weight sum, Y_i, is defined as $\mathsf{Y}_i = \Sigma_{k=i}^{\infty} \mathsf{w}_k$. Thus, it is clear that the $(i+1)^{\text{th}}$ residue vector can be derived from the i^{th} residue vector as given in Equation (3.5). Similarly, the i^{th} reserve vector is nothing but the i^{th} term in the infinite sum of the graph propagation equation, and it can be expressed in terms of the i^{th} term of the residue vector as given in Equation (3.5).

$$\mathbf{r}^{(i+1)} = \frac{\mathsf{Y}_{i+1}}{\mathsf{Y}_i} \cdot \mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b} \cdot \mathbf{r}^{(i)}, \qquad \mathbf{q}^{(i)} = \frac{\mathsf{w}_i}{\mathsf{Y}_i} \cdot \mathbf{r}^{(i)} \tag{3.5}$$

By using $\mathbf{r}^{(0)}$ and the relation between $\mathbf{r}^{(i)}$, $\mathbf{r}^{(i+1)}$, all residue vectors $\mathbf{r}^{(i)}$ for $i \in \{0, \ldots, \mathsf{L}\}$ can be computed. Each of the reserve vectors can also be obtained from the corresponding residue vectors, which can then be used to compute the graph propagation vector as $\boldsymbol{\rho} = \Sigma_{i=0}^{\infty} \mathbf{q}^{(i)}$. Given the above background, the steps of Algorithm 2 can be summarised as follows.

Initially, residue vector $\mathbf{r}^{(0)} = \mathsf{Y}_0 \cdot (\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b})^0 \cdot \mathbf{x}$ is set as the signal vector $\mathbf{x}$ with the partial sum $Y_0 = \Sigma_{k=0}^{\infty}\mathsf{w}_k = 1$. The algorithm repeats for L iterations where in each iteration i, the i^{th} term of the graph propagation sum is computed. Lines 4-6 in the algorithm compute the residue vector $\mathbf{r}^{(i+1)}$. Note that the term $(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b})$ in $\mathbf{r}^{(i+1)}$ is a square matrix with the (u, v) entry having the corresponding value of $\mathbf{A}$ weighted by a factor of $\frac{1}{(\mathsf{deg_u})^a(\mathsf{deg_v})^b}$. Hence, entries in $\mathbf{A}$ that are 0 will not contribute in the computation of $\mathbf{r}^{(i+1)}$. Thus, the algorithm makes use of the fact that the index v of the residue vector, $\mathbf{r}^{(i+1)}[\mathsf{v}]$ is only updated by the residue entries of all it's neighbours u such that $\mathsf{v} \in \mathsf{Nb_u}$. Here $\mathsf{Nb_u}$ denotes the neighbours of node u. In this way, the authors in [222] design a vertex-centric algorithm that operates on vectors rather than matrices. In line 8, the reserve vector is derived from the residue vector, and in line 10, the reserve vector is used to update the graph propagation result vector $\boldsymbol{\rho}$. Note that, for the case of HKPR, the above computation is parameterized with $a = 0, b = 1$.

3.5.2 The data-oblivious variant

We begin by discussing the challenges that arise in naively using MPC to translate Algorithm 2 into a secure variant, followed by discussing the resolutions for the same. Consider the steps in Algorithm 2. For each node u, step 3 executes for only those nodes in the input graph that have a non-zero value in the corresponding component of the residue vector $\mathbf{r}[\mathsf{u}]$. Hence, step 3 reveals whether a given node u has a non-zero value $\mathbf{r}[\mathsf{u}]$. Further, steps 4-6 selectively update the residue vector components $\mathbf{r}[\mathsf{v}]$ corresponding to each neighbour v of the current node u. Hence, the number of times the residue vector $\mathbf{r}$ is updated reveals the degree of the node u. Thus, the algorithm clearly does not qualify to be data-oblivious. Translating Algorithm 2 via MPC to obtain its secure variant by using secure protocols for operations used in Algorithm 2, will thus result in leaking the above-mentioned information. Hence, designing a secure variant of Algorithm 2 first requires designing a data-oblivious equivalent of the same. Since Algorithm 2 adheres to the message-passing paradigm, we obtain its data-oblivious equivalent by relying on the GraphSC paradigm. This requires defining the Scatter, Gather primitives specific to the considered algorithm, which we describe next.

To adhere to the GraphSC paradigm, the input graph is expressed using a DAG list representation G (see §2.6). The i^{th} tuple in the list is denoted by $\mathsf{G}[i]$. The data values associated with each tuple $\mathsf{G}[i]$ include—(i) $\mathsf{G}[i].\mathsf{dt}$ to store messages that are sent across the edges, (ii) $\mathsf{G}[i].\mathsf{deg}$ to store the degree of a node, (iii) $\mathsf{G}[i].\mathsf{r}$ to store the residue vector component of a node, and (iv) $\mathsf{G}[i].\rho$ to store the propagation vector component of each node (i.e., HKPR values). Recall that $\mathsf{G}[i].\mathsf{isV}$ denotes whether a tuple corresponds to a vertex or not. Since the graph

propagation Algorithm 2 proceeds in an iterative manner, assuming the index of the current iteration to be j, the GraphSC primitives of Scatter and Gather are defined in Fig. 3.9. Running one iteration of Scatter followed by Gather accomplishes the same computation as in one iteration of Algorithm 2, albeit in a data-oblivious manner, as follows. Computing the graph propagation value $\boldsymbol{\rho}[\mathsf{v}]$ for a node v requires the residue vector component $\mathbf{r}[\mathsf{u}]$ corresponding to each of its neighbour u as well as $(\mathsf{deg}_{\mathsf{u}})^b$. Hence, the required information is made available on the edges via Scatter. Since the weights w_j and partial sums Y_j are public parameters, each node propagates $\frac{\mathsf{Y}_{j+1}\mathsf{G}[i].\mathsf{r}}{\mathsf{Y}_j(\mathsf{G}[i].\mathsf{deg})^b}$ across its outgoing edges in Scatter. During Gather, these values over incoming edges are aggregated in agg by summing them up. The residue vector component at vertex v is updated as $\frac{\mathsf{agg}}{\mathsf{Y}_j(\mathsf{G}[\mathsf{v}].\mathsf{deg})^a}$. However, note that this generates the residue vector $\mathbf{r}^{(j+1)}$, required for the next iteration. Hence, prior to performing this update, the value stored in the residue vector $\mathbf{r}^{(j)}$ is used to update the graph propagation vector at vertex v as $\mathsf{G}[\mathsf{v}].\rho = \mathsf{G}[\mathsf{v}].\rho + \frac{\mathsf{w}_j\mathsf{G}[\mathsf{v}].\mathsf{r}^{(j)}}{\mathsf{Y}_{j+1}}$. Our approach to designing Gather in this way does not require explicitly storing the reserve vector $\mathbf{q}$ as well as both, $\mathbf{r}^{(j)}$ and $\mathbf{r}^{(j+1)}$.

Scatter(G)
for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\mathsf{val} = \left(\frac{\mathsf{Y}_{j+1}}{\mathsf{Y}_j}\right) \frac{\mathsf{G}[i].\mathsf{r}}{(\mathsf{G}[i].\mathsf{deg})^b}$
  else
    $\mathsf{G}[i].\mathsf{dt} = \mathsf{val}$

Gather(G)
for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\mathsf{G}[i].\rho = \mathsf{G}[i].\rho + \frac{\mathsf{w}_j}{\mathsf{Y}_j}\mathsf{G}[i].\mathsf{r}$
    $\mathsf{G}[i].\mathsf{r} = \frac{\mathsf{agg}}{(\mathsf{G}[i].\mathsf{deg})^a}$
    $\mathsf{agg} = 0$
  else
    $\mathsf{agg} = \mathsf{agg} + \mathsf{G}[i].\mathsf{dt}$

Figure 3.9: Scatter and Gather for j^{th} iteration of Algorithm 2.

3.5.3 The secure variant

We now describe the MPC protocol Π_{SGP} designed to securely evaluate the graph propagation algorithm in Fig. 3.10. The protocol begins by assuming the required inputs G, $\mathbf{x}$, are already held in a secret-shared fashion among the parties. It also takes as input the public parameters of $\mathsf{Y}_j, \mathsf{w}_j$ for $j \in \{0, \ldots, \mathsf{L}-1\}$. Since the components of the residue vector $\mathbf{r}$ are initialized using the entries in the signal vector $\mathbf{x}$, unlike in the cleartext variant, $\mathbf{x}$ is required to be of the same size as the DAG list G. That is, $\mathbf{x}$ must be secretly shared as a vector of dimension $|\mathsf{V}| + |\mathsf{E}|$, with value 1 at the location of the source vertex and 0 at all other entries. As in the definition of Scatter in Fig. 3.9, val should be computed only at the tuples corresponding to nodes. Hence, Π_{SGP} computes val' at each tuple. However, val is obliviously updated to store

the correct value using Π_{Sel} (see Table 2.1). Similarly, for Gather, aggregation is obliviously performed only at the tuples corresponding to edges using Π_{Sel}. Further, note that multiplying values that are secret-shared requires invoking the Π_{Mul} protocol. Finally, note that an explicit call to division for computing $[\![\frac{1}{\mathsf{G}[i].\mathsf{deg}}]\!]$ is avoided by ensuring this value is made available in shares when obtaining shares of G (and it is set to 0 when $\mathsf{G}[i]$ represents an edge). Since public parameters $a, b \in \{0, 1\}$, no additional multiplications are required.

Protocol $\Pi_{\mathsf{SGP}}\left(\mathcal{P}, [\![\mathsf{G}]\!], [\![\mathsf{x}]\!], \{\mathsf{Y}_j, \mathsf{w}_j\}_{j=0}^{\mathsf{L}}, a, b\right)$

- for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do: $[\![\mathsf{G}[i].\mathsf{r}]\!] = [\![\mathsf{x}[i]]\!]$, $[\![\mathsf{G}[i].\rho]\!] = [\![0]\!]$
- for $j = 0$ to $\mathsf{L} - 1$ do
  - $[\![\mathsf{G}]\!] = \Pi_{\mathsf{Shuffle}}([\![\mathsf{G}]\!])$; apply public permutation to source sort G. Set $[\![\mathsf{val}]\!] = [\![0]\!]$.

  Scatter(G)
  - for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do
    - $[\![\mathsf{val}']\!] = \left(\frac{\mathsf{Y}_{j+1}}{\mathsf{Y}_j}\right) \cdot \Pi_{\mathsf{Mul}}\left([\![\mathsf{G}[i].\mathsf{r}]\!], [\![\frac{1}{(\mathsf{G}[i].\mathsf{deg})^b}]\!], \mathsf{f}\right)$
    - $[\![\mathsf{val}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{val}]\!], [\![\mathsf{val}']\!], [\![\mathsf{G}[i].\mathsf{isV}]\!]^{\mathbf{B}})$
    - $[\![\mathsf{G}[i].\mathsf{dt}]\!] = [\![\mathsf{val}]\!]$
  - $[\![\mathsf{G}]\!] = \Pi_{\mathsf{Shuffle}}([\![\mathsf{G}]\!])$; apply public permutation to destination sort G

  Gather(G)
  - for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do
    - $[\![\mathsf{G}[i].\rho]\!] = [\![\mathsf{G}[i].\rho]\!] + \left(\frac{\mathsf{w}_j}{\mathsf{Y}_j}\right) [\![\mathsf{G}[i].\mathsf{r}]\!]$
    - $[\![\mathsf{G}[i].\mathsf{r}]\!] = \Pi_{\mathsf{Mul}}\left([\![\mathsf{agg}]\!], [\![\frac{1}{(\mathsf{G}[i].\mathsf{deg})^a}]\!], \mathsf{f}\right)$
    - $[\![\mathsf{agg}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{agg}]\!] + [\![\mathsf{G}[i].\mathsf{dt}]\!], [\![0]\!], [\![\mathsf{G.isV}]\!]^{\mathbf{B}})$

Figure 3.10: Secure HKPR computation.

3.5.4 Other graph propagation metrics

The graph propagation Algorithm 2 is used to compute the graph propagation result vector $\boldsymbol{\rho}$ using the graph propagation equation $\boldsymbol{\rho} = \Sigma_{i=0}^{\infty} \mathsf{w}_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \mathbf{x}$. Recall that this graph propagation equation can also be used to compute many other graph propagation metrics such as PageRank, personalised PageRank, Katz centrality score etc. These propagation metrics, as described in the work of [222], are listed in Table 3.3. Given the parameters of the graph propagation equation, such as a, b, weights w and the initial seed node captured in the signal vector $\mathbf{x}$, the graph propagation algorithm can be used to find the corresponding information regarding the nodes of the graph. Hence, using the secure protocol given in Fig. 3.10, several

graph propagation metrics can be computed without leaking any private information.

Graph propagation metric	Parameters			Graph propagation equation
	a	b	w_i	
L-hop transition probability	0	1	$\mathsf{w}_i = 0$ $(i \neq \mathrm{L})$, $\mathsf{w}_L = 1$	$\boldsymbol{\rho} = (\mathbf{AD}^{-1})^L \cdot \mathbf{x}$
PageRank	0	1	$\alpha(1-\alpha)^i$	$\boldsymbol{\rho} = \Sigma_{i=0}^{\infty} \alpha(1-\alpha)^i \cdot (\mathbf{AD}^{-1})^i \cdot \mathbf{x}$
Personalised PageRank	0	1	$\alpha(1-\alpha)^i$	$\boldsymbol{\rho} = \Sigma_{i=0}^{\infty} \alpha(1-\alpha)^i \cdot (\mathbf{AD}^{-1})^i \cdot \mathbf{x}$
Single Target PageRank	1	0	$\alpha(1-\alpha)^i$	$\boldsymbol{\rho} = \Sigma_{i=0}^{\infty} \alpha(1-\alpha)^i \cdot (\mathbf{D}^{-1}\mathbf{A})^i \cdot \mathbf{x}$
Heat Kernel PageRank	0	1	$\frac{e^{-t}t^i}{i!}$	$\boldsymbol{\rho} = \Sigma_{i=0}^{\infty} \frac{e^{-t}t^i}{i!} \cdot (\mathbf{AD}^{-1})^i \cdot \mathbf{x}$
Katz	0	0	β^i	$\boldsymbol{\rho} = \Sigma_{i=0}^{\infty} \beta^i \cdot (\mathbf{A})^i \cdot \mathbf{x}$

Note that $\mathbf{x}$ denotes the signal vector, which is the one-hot encoding of the seed node. However, in the case of PageRank, $\mathbf{x}$ is set as the uniform probability distribution vector with each entry being $\frac{1}{|\mathsf{V}|}$. Similarly, in the case of Personalized PageRank, it is set as the teleportation probability distribution vector corresponding to the seed node.

Table 3.3: Graph propagation metrics.

3.6 Privacy-preserving clustering

Here, we describe the protocol for realizing local clustering in a privacy-preserving manner using the computed HKPR values.

3.6.1 The cleartext algorithm

We rely on the HKPR-based clustering algorithm described in [54] as it provides state-of-the-art results for local clustering. The steps for the same are provided in Algorithm 3, which describes an approximate algorithm to find a local cluster around a given seed node. To find a suitable local cluster, the algorithm uses heat kernel PageRank values of all vertices in the graph, computed with respect to the seed node. To assess the quality of the cluster, the algorithm uses *Cheeger ratio* as the metric. Cheeger ratio Φ_s with respect to a set of nodes S is defined as in Equation (3.6) where, ∂ is the number of edges that cross from the set S to the set of remaining vertices in the graph $\mathsf{V}\backslash\mathsf{S}$, and volume $\mathsf{vol_S}$ of a set S is defined as the sum of degrees of all vertices present in the set S.

$$\Phi_s = \frac{\partial}{min\{\mathsf{vol_S}, \mathsf{vol_{V\backslash S}}\}} \tag{3.6}$$

Thus, a smaller Cheeger ratio signifies a better cluster, since it minimizes the number of edges that cross the cluster and maximizes the degree of nodes within the cluster. The clustering algorithm takes as input the target Cheeger ratio ϕ and the target cluster volume ς, with the

constraint that ς must be less than or equal to $\mathsf{vol_G}/4$. The algorithm identifies $\mathsf{S_i}$ as a suitable cluster if it satisfies the following condition:

$$\varsigma/2 \leq \mathsf{vol_{S_i}} \leq 2\varsigma \quad \text{and} \quad \Phi_{s_i} \leq \sqrt{8\phi} \tag{3.7}$$

Cluster $\mathsf{S_i}$ with the lowest Cheeger ratio among all suitable clusters, if found, is output. As clusters are identified based on HKPR, the algorithm also requires the HKPR values $\boldsymbol{\rho}$ supplied as input. The algorithm begins by sorting the vertices in the graph in the descending order of $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$, where $\mathsf{deg_v}$ is the degree of vertex v. This ensures that a higher priority is given to those vertices having high HKPR values and low degrees when forming the cluster. In lines 3-12, the algorithm performs a linear scan over the sorted vertices such that in each iteration i, the cluster $\mathsf{S_i}$ under consideration is the set of vertices $\bigcup_{j\leq i} \mathsf{v}_j$. If the volume of this cluster is more than 2ς, then the protocol can break and consider no further clusters. Since each subsequent iteration includes more vertices in the considered cluster, the corresponding volume will continue to increase. Thus, the volume constraint will continue to fail, and hence the larger clusters can be discounted. We refer an interested reader to [54] for further details on Algorithm 3 and its correctness.

Algorithm 3: HKPR-based graph clustering

Input: Undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, $\boldsymbol{\rho}$: Graph propagation vector of dimension $|\mathsf{V}|$, ς : Target cluster volume, ϕ : Target Cheeger ratio, u : Seed vertex

Output: S = Set of nodes that satisfies constraints in equation 3.7 and have minimum Cheeger ratio

```
1  sort vertices of G with respect to ρ[v]/deg_v;
2  φ_min = ∞, S = Empty set;
3  for i = 1 to |V| do
4  |   S_i = ⋃_{j≤i} v_j ;
5  |   Φ_{s_i} = Cheeger ratio of S_i as per Equation (3.6)
6  |   if vol_{S_i} > 2ς then
7  |   |   Break;
8  |   else if ς/2 ≤ vol_{S_i} and Φ_{s_i} ≤ √(8φ) then
9  |   |   if Φ_{s_i} < φ_min then
10 |   |   |   φ_min = Φ_{s_i}; S = S_i;
11 |   |   end
12 |   end
13 end
14 if S is not an Empty set then Output S else Output "No Cluster found";
```

3.6.2 The data-oblivious variant

Algorithm 3 does not qualify as a data-oblivious algorithm since many of the steps are input-dependent. For example, step 1 requires sorting the vertices in the graph with respect to the ratio $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$. The sorted order clearly is dependent on the structure of the graph G. Further, computing the Cheeger ratio in step 5 depends on the current cluster and the overall topology of G, as evident from equation 3.6. Thus, similar to the graph propagation algorithm, we first design a data-oblivious algorithm, followed by designing its secure variant. Given that the vertices are sorted with respect to the ratio $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$, note that each iteration of the clustering algorithm computes the Cheeger ratio specific to the considered set $\mathsf{S_i}$. This requires the computation of $\partial, \mathsf{vol_{S_i}}, \mathsf{vol_{V \setminus S_i}}$, which can benefit from translation to the GraphSC paradigm. Taking Algorithm 3 as the input algorithm to GraphSC, computing $\mathsf{vol_{S_i}}, \mathsf{vol_{V \setminus S_i}}$ can be performed in one linear scan given the degree of the nodes. With respect to the computation of ∂, a naive approach via Scatter and Gather would require computing ∂ for each $\mathsf{S_i}$, and thereby require a linear scan each time. Since performing Scatter and Gather has $O(|\mathsf{V}| + |\mathsf{E}|)$ complexity, computing ∂ for all vertices has a complexity of $O(|\mathsf{V}|(|\mathsf{V}| + |\mathsf{E}|))$. This is highly inefficient. We overcome the inefficiency and avoid the need for multiple linear scans by carefully modifying the algorithm such that only a single scan over the DAG list G suffices for computing ∂ for all vertices. The complexity of our resulting data-oblivious algorithm is thus drastically reduced to $O(|\mathsf{V}| + |\mathsf{E}|)$ from $O(|\mathsf{V}|(|\mathsf{V}| + |\mathsf{E}|))$ of the naive approach. Recall that this linear complexity in $|\mathsf{V}| + |\mathsf{E}|$ can further be reduced to sub-linear following the technique of [179]. To achieve this, the data values associated with $\mathsf{G}[i]$ (see §3.5.2), are augmented with a new component $\mathsf{G}[i].\mathsf{GreaterCount}$. This component is used to store the number of neighbours v of a given node that have a greater $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$ value than the current node. As will be described next, GreaterCount is used to determine the number of edges that cross a cluster of vertices, and thereby compute ∂ in a single scan of the DAG list. Thus, Scatter and Gather are defined to populate the GreaterCount component.

We let FindCluster (Algorithm 4) denote the modified version of Algorithm 3 that relies on GreaterCount to compute local clusters. We now discuss how FindCluster uses GreaterCount to compute ∂, and then define the Scatter and Gather primitives to compute GreaterCount. FindCluster proceeds to sort the DAG list in descending order of $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$, assuming that GreaterCount is computed. This is followed by performing a scan of the DAG list to identify the suitable cluster with the minimum Cheeger ratio. Each time a vertex v_i is encountered, it is added to the existing cluster, say S_{i-1}, to form a new cluster, say S_i. To compute the Cheeger ratio of the new cluster, we require the number of edges ∂ that cross the new cluster $\mathsf{S_i}$. We observe that the computation of the same can be simplified by accounting for (i) ∂ of the previous

cluster S_{i-1}, (ii) new cross edges (i.e., cross edges between v_i and $\mathsf{V} \setminus \mathsf{S}_{i-1}$), and (iii) previous cross edges that now lie within the new cluster (i.e., cross edges between S_{i-1} and v_i). Since GreaterCount was defined to count the number of neighbours u of the vertex v_i that had a higher value of $\frac{\boldsymbol{\rho}[\mathsf{u}]}{\mathsf{deg_u}}$, it implicitly tracks number of neighbours of v_i that were part of previous cluster S_{i-1}. Thus, GreaterCount keeps track of the number of type (iii) edges. Given that all the edges of v_i are either of type (ii) or type (iii), the number of edges of type (ii) can be computed as $\mathsf{G}[\mathsf{v}].\mathsf{deg} - \mathsf{G}[\mathsf{v}].\mathsf{GreaterCount}$. Further, $\mathsf{vol}_{\mathsf{S}_i}$ and $\mathsf{vol}_{\mathsf{V}\setminus\mathsf{S}_i}$ can be updated based on the degree $\mathsf{G}[\mathsf{v}_i].\mathsf{deg}$ of the vertex v_i being added to the cluster. Hence, the Cheeger ratio of the new cluster S_i can be calculated using the updated ∂ and $\mathsf{vol}_{\mathsf{S}_i}, \mathsf{vol}_{\mathsf{V}\setminus\mathsf{S}_i}$. Further, note that since $\mathsf{G}[i].\rho$ component for an edge will be 0, when G is sorted in the descending order according to $\frac{\rho}{\mathsf{deg}}$, all the vertices will be placed before the edges. Hence, the flag variable is used to keep track of the number of vertices in the sorted list that constitute the largest cluster that satisfies the condition in Equation (3.7).

Algorithm 4: FindCluster(G)

Input: Undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$
Output: S = Set of nodes that belong to the local cluster

Sort G in source order and perform Scatter as given in Fig. 3.11 ;
Sort G in destination order and perform Gather as given in Fig. 3.11 ;
Sort G by $\boldsymbol{\rho}[\mathsf{v}]/\mathsf{deg_v}$ and set $\partial = 0$, $\mathsf{vol_S} = 0$, $\mathsf{vol_{V\setminus S}} = \sum_{\mathsf{v}\in\mathsf{V}} \mathsf{G}[\mathsf{v}].\mathsf{deg}$ $\phi_{\mathsf{min}} = \infty$, $\mathsf{flag} = 0$;
for $i = 1$ *to* $|\mathsf{V}| + |\mathsf{E}|$ **do**
    if $\mathsf{G}[i].\mathsf{isV}$ **then**
        $\partial = \partial + \mathsf{G}[i].\mathsf{deg} - 2 \cdot \mathsf{G}[i].\mathsf{GreaterCount}$; $\mathsf{vol_S} = \mathsf{vol_S} + \mathsf{G}[i].\mathsf{deg}$; $\mathsf{vol_{V\setminus S}} = \mathsf{vol_{V\setminus S}}$ - $\mathsf{G}[i].\mathsf{deg}$;
        $\Phi_s = \frac{\partial}{min(\mathsf{vol_S}, \mathsf{vol_{V\setminus S}})}$;
        if $(\varsigma/2 \leq \mathsf{vol_S} \leq 2\varsigma) \& \ (\Phi_s \ \leq \sqrt{8\phi}) \& (\Phi_s < \phi_{\mathsf{min}})$ **then**
            $\phi_{\mathsf{min}} = \Phi_s$; $\mathsf{flag} = i$;
        end
    end
end
if $\mathsf{flag} > 0$ **then** Output the first flag elements of G **else** Output "No Cluster found";

We now define the Scatter and Gather primitives (Fig. 3.11) to compute GreaterCount. During Scatter, the vertices scatter $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$ across their outgoing edges. To compute GreaterCount, Gather is defined to first perform a reverse scan of the DAG list, followed by a forward scan. During the reverse scan, the data value at each edge (u, v) is updated to store the difference $\frac{\boldsymbol{\rho}[\mathsf{u}]}{\mathsf{deg_u}} - \frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$ corresponding to its endpoints. In the forward scan, each vertex counts the number of incoming edges having data value (i.e., the above difference) greater than zero and stores the same in GreaterCount. In this way, performing a Scatter followed by Gather allows every node

v to count the number of neighboring vertices u with a greater $\frac{\boldsymbol{\rho}[\mathsf{u}]}{\deg_{\mathsf{u}}}$ than $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\deg_{\mathsf{v}}}$.

Scatter(G)

for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do

  if $\mathsf{G}[i].\mathsf{isV}$ then

    $\mathsf{val} = \frac{\mathsf{G}[i].\rho}{\mathsf{G}[i].\mathsf{deg}}$

  else

    $\mathsf{G}[\mathrm{i}].\mathsf{dt} = \mathsf{val}$

Gather(G)

$\mathsf{agg} = 0$

for $i = |\mathsf{V}| + |\mathsf{E}|$ to 1 do

  if $\mathsf{G}[\mathrm{i}].\mathsf{isV}$ then $\mathsf{val} = \frac{\mathsf{G}[i].\rho}{\mathsf{G}[i].\mathsf{deg}}$

  else $\mathsf{G}[i].\mathsf{dt} = \mathsf{G}[i].\mathsf{dt} - \mathsf{val}$

for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do

  if $\mathsf{G}[\mathrm{i}].\mathsf{isV}$ then

    $\mathsf{G}[\mathrm{i}].\mathsf{GreaterCount} = \mathsf{agg}$

    $\mathsf{agg} = 0$

  else if $\mathsf{G}[\mathrm{i}].\mathsf{dt} > 0$ then $\mathsf{agg} = \mathsf{agg} + 1$

Figure 3.11: Scatter and Gather to compute GreaterCount.

3.6.3 The secure variant

The secure protocol for computing the local cluster is given in Fig. 3.12.

Protocol $\Pi_{\mathsf{SClustering}}(\mathcal{P}, [\![\mathsf{G}]\!], [\![\boldsymbol{\rho}]\!], \phi)$

- $\Pi_{\mathsf{greaterCount}}(\mathcal{P}, [\![\mathsf{G}]\!], [\![\boldsymbol{\rho}]\!])$ (Fig. 3.13)
- $[\![\mathsf{G}]\!] = \Pi_{\mathsf{Sort}}(\mathsf{G}, [\![\mathsf{G}.\boldsymbol{\rho}]\!])$
- Initialize $\partial = [\![0]\!], [\![\mathsf{vol_S}]\!] = [\![0]\!], [\![\mathsf{vol}_{\mathsf{V}\setminus\mathsf{S}}]\!] = \sum_{\mathsf{v}\in\mathsf{V}} [\![\mathsf{G}[\mathsf{v}].\mathsf{deg}]\!]$, $[\![\phi_{\mathsf{min}}]\!] = [\![2^{32}]\!]$ and $[\![\mathsf{flag}]\!] = [\![0]\!]$
- for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do
  - $[\![\partial]\!] = [\![\partial]\!] + [\![\mathsf{G}[i].\mathsf{deg}]\!]$ - 2·$[\![\mathsf{G}[i].\mathsf{GreaterCount}]\!]$
  - $[\![\mathsf{vol_S}]\!] = [\![\mathsf{vol_S}]\!] + [\![\mathsf{G}[i].\mathsf{deg}]\!]$, $[\![\mathsf{vol}_{\mathsf{G}\setminus\mathsf{S}}]\!] = [\![\mathsf{vol}_{\mathsf{G}\setminus\mathsf{S}}]\!]$ - $[\![\mathsf{G}[i].\mathsf{deg}]\!]$
  - $[\![\mathrm{min}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}([\![\mathsf{vol}_{\mathsf{G}\setminus\mathsf{S}}]\!], [\![\mathsf{vol_S}]\!])$, $[\![\mathsf{vol}_{\mathrm{min}}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{vol_S}]\!], [\![\mathsf{vol}_{\mathsf{G}\setminus\mathsf{S}}]\!], [\![\mathrm{min}]\!]^{\mathbf{B}})$
  - $[\![\Phi_s]\!] = \Pi_{\mathsf{Div}}([\![\partial]\!], [\![\mathsf{vol}_{\mathrm{min}}]\!])$
  - $[\![\theta_1]\!]^{\mathbf{B}} = 1 \oplus \Pi_{\mathsf{Comp}}(2[\![\varsigma]\!], [\![\mathsf{vol_S}]\!])$, $[\![\theta_2]\!]^{\mathbf{B}} = 1 \oplus \Pi_{\mathsf{Comp}}([\![\mathsf{vol_S}]\!], [\![\varsigma]\!]/2)$
    $[\![\theta_3]\!]^{\mathbf{B}} = 1 \oplus \Pi_{\mathsf{Comp}}\left([\![\sqrt{8\phi}]\!], [\![\Phi_s]\!]\right)$
  - $[\![\mathrm{min}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}([\![\Phi_s]\!], [\![\phi_{\mathsf{min}}]\!])$
  - $[\![\theta]\!]^{\mathbf{B}} = \Pi_{\mathsf{4\text{-}Mul}}([\![\theta_1]\!]^{\mathbf{B}}, [\![\theta_2]\!]^{\mathbf{B}}, [\![\theta_3]\!]^{\mathbf{B}}, [\![\mathrm{min}]\!]^{\mathbf{B}})$
  - $[\![\phi_{\mathsf{min}}]\!] = \Pi_{\mathsf{Sel}}([\![\phi_{\mathsf{min}}]\!], [\![\Phi_s]\!], [\![\theta]\!]^{\mathbf{B}})$, $[\![\mathsf{flag}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{flag}]\!], [\![i]\!], [\![\theta]\!]^{\mathbf{B}})$
- Output $[\![\mathsf{flag}]\!]$ and $[\![\mathsf{G}]\!]$.

Figure 3.12: Secure clustering.

$\Pi_{\mathsf{SClustering}}$ takes as input secret shares of G, ρ, and the public target Cheeger ratio ϕ. In ad-

dition to the primitives required for securely computing the HKPR metric, the current protocol mainly requires Π_{Comp} for securely evaluating the conditional statements that require comparison and Π_{Div} for computing the Cheeger ratio securely. The secure protocol for computing the local cluster internally relies on the secure protocol to compute GreaterCount via the Scatter, and Gather. The secure protocol for the latter appears in Fig. 3.13.

Protocol $\Pi_{\mathsf{greaterCount}}(\mathcal{P}, [\![\mathsf{G}]\!], [\![\boldsymbol{\rho}]\!])$

– Compute $[\![\mathsf{G}]\!] = \Pi_{\mathsf{Shuffle}}([\![\mathsf{G}]\!])$ and apply public permutation to source order DAG list

Scatter(G)

– $[\![\mathsf{val}]\!] = [\![0]\!]$

– for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do

- $[\![\mathsf{G}[i].\rho]\!] = \Pi_{\mathsf{Mul}}([\![\mathsf{G}[i].\rho]\!], [\![\frac{1}{\mathsf{G}[i].\mathsf{deg}}]\!], \mathsf{f})$
- $[\![\mathsf{v}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{v}]\!], [\![\mathsf{G}[i].\rho]\!], [\![\mathsf{G}[i].\mathsf{isV}]\!]^{\mathbf{B}})$
- $[\![\mathsf{G}[i].\mathsf{dt}]\!] = [\![\mathsf{val}]\!]$

– Compute $[\![\mathsf{G}]\!] = \Pi_{\mathsf{Shuffle}}([\![\mathsf{G}]\!])$ and apply public permutation to destination order DAG list

Gather(G)

– $[\![\mathsf{agg}]\!] = [\![0]\!]$

– for $i = |\mathsf{V}| + |\mathsf{E}|$ to 1 do

- $[\![\mathsf{val}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{v}]\!], [\![\mathsf{G}[i].\rho]\!], [\![\mathsf{G.isV}]\!]^{\mathbf{B}})$
- $[\![\mathsf{G}[i].\mathsf{dt}]\!] = [\![\mathsf{G}[i].\mathsf{dt}]\!] - [\![\mathsf{val}]\!]$

– for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do

- $[\![\mathsf{G}[i].\mathsf{GreaterCount}]\!] = [\![\mathsf{agg}]\!]$
- $[\![\mathsf{agg}]\!] = \Pi_{\mathsf{Sel}}\left([\![\mathsf{agg}]\!], 0, [\![\mathsf{G}[i].\mathsf{isV}]\!]^{\mathbf{B}}\right)$
- $[\![c]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(0, [\![\mathsf{G}[i].\mathsf{dt}]\!])$
- $[\![\mathsf{agg}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{agg}]\!], [\![\mathsf{agg}]\!] + 1, [\![c]\!]^{\mathbf{B}})$

Figure 3.13: Computing GreaterCount.

3.7 Benchmarks

In this section, we demonstrate the practicality of our protocols by benchmarking their performance. We benchmark the secure HKPR-based clustering protocol on synthetic graphs, and YouTube social network [172]. We additionally compare the accuracy of our secure protocols to their cleartext counterparts.

3.7.1 Accuracy results

Computing the HKPR metric, as well as the Cheeger ratio used in clustering demands operating on decimal values. Hence, the secure computation of these proceeds via fixed-point arithmetic (FPA). Since numbers in FPA representation have limited precision, operating over FPA introduces errors in comparison to floating-point arithmetic since the latter allows for higher precision. This holds true even when performing the computations on cleartext. Further, the method of probabilistic truncation and approximation of division used for secure computation introduces additional errors over the cleartext FPA algorithm. To showcase that secure computation via FPA does not result in significant accuracy loss in comparison to computations performed on cleartext floating-point equivalent, we compare and report the accuracy of both. Further, to showcase the error introduced when moving from floating-point to fixed-point representation, we report the difference in the accuracy of the two when considering the cleartext algorithm. All the implementations are in Python over a 64-bit ring and use NumPy[102] and NetworkX[97] libraries. Henceforth, the terms *float* and *fixed* denote the floating-point and fixed-point computation of the cleartext algorithm, respectively. Similarly, the term *secure* refers to the secure fixed-point computation. We report the accuracy of the graph propagation algorithm first, followed by that of clustering.

Datasets: We benchmark the accuracy of algorithms on three different types of synthetic graphs. To showcase the accuracy loss when computing over large real-world networks, we also consider YouTube social network [172]. Table 3.4 summarises details of these graphs.

3.7.1.1 Graph propagation metrics

Recall that our secure graph propagation protocol Π_{SGP} allows computing various graph propagation metrics. Since accuracy varies with the propagation metric under consideration, we account for the following metrics– L-hop transition probability, PageRank, single target PageRank and HKPR. We use MaxError and L1 error to capture accuracy loss. Given two graph propagation vectors $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$, the MaxError is computed as $\max_{\mathsf{v}\in\mathsf{V}}\left(\left|\frac{\boldsymbol{\rho}_1[\mathsf{v}]}{\mathsf{deg}_\mathsf{v}} - \frac{\boldsymbol{\rho}_2[\mathsf{v}]}{\mathsf{deg}_\mathsf{v}}\right|\right)$. The MaxError measures the maximum absolute error in $\boldsymbol{\rho}_2$ with respect to $\boldsymbol{\rho}_1$. Similarly, the L1 error is given by $\text{L1} = \frac{1}{|\mathsf{V}|}\sum_{\mathsf{v}\in\mathsf{V}}\left(\left|\frac{\boldsymbol{\rho}_1[\mathsf{v}]}{\mathsf{deg}_\mathsf{v}} - \frac{\boldsymbol{\rho}_2[\mathsf{v}]}{\mathsf{deg}_\mathsf{v}}\right|\right)$. The L1 error measures the average absolute error in $\boldsymbol{\rho}_2$ with respect to $\boldsymbol{\rho}_1$ over all vertices of the graph. We compute the MaxError and L1 error in the fixed-point cleartext as well as *secure* algorithms, each with respect to the floating-point cleartext algorithm. The errors are reported with respect to the YouTube graph in Table 3.5. As evident from the results of Table 3.5, both the MaxError and L1 error are in the order of $\times 10^{-5}$ or smaller with respect to the floating-point algorithm. This implies

Model	|V|	d	p
Small world	100	5	0.1
	500	5	0.1
	800	5	0.1
	1000	5	0.1
Powerlaw cluster	100	5	0.1
	500	5	0.1
	800	5	0.1
Preferential Attachment	100	5	-
	500	5	-
	800	5	-
Youtube	1134890	-	-

|V| denotes the number of vertices, d denotes the number of neighbours each vertex is assigned, and p denotes the probability of switching an edge for the case of a small world graph, or the probability of forming a triangle for the case of the power-law cluster.

Table 3.4: Graph datasets used for accuracy testing.

that the difference in accuracy is visible only after the 5^{th} decimal digit, and thus, the loss is very small. Benchmarks on synthetic graphs yield errors in similar orders and hence are not reported.

Similarity measures	MaxError		L1 Error	
	Fixed	Secure	Fixed	Secure
L-Hop transition	1.3674×10^{-5}	6.0872×10^{-5}	1.6719×10^{-6}	1.6719×10^{-6}
PageRank(PR)*	1.8223×10^{-08}	1.8622×10^{-8}	5.0661×10^{-7}	3.7420×10^{-7}
Single target PR	3.2904×10^{-5}	5.2424×10^{-5}	1.7061×10^{-7}	3.0636×10^{-7}
HKPR	1.3865×10^{-5}	2.2393×10^{-5}	0.7911×10^{-8}	1.0205×10^{-8}

* Precision is set to 28 bits when operating on fixed-point to accommodate small values of signal vector

Table 3.5: MaxError and L1 error comparison of cleartext FPA and secure FPA algorithms with cleartext floating-point algorithm for various graph propagation metrics.

3.7.1.2 Clustering

We perform local clustering on the various graphs described in Table 3.4 and report the accuracy results. While reporting the accuracy of the secure local clustering algorithm, we also account for the accuracy loss incurred in securely computing the HKPR propagation vector ($\boldsymbol{\rho}$), which is fed as input to our secure clustering algorithm. Although Personalized PageRank has also been used to perform local clustering, it is known from the literature [230, 134, 54] that HKPR-based clustering outputs better quality clusters. Hence, we report accuracy results for clustering based on HKPR only. We take the Cheeger ratio (Φ_s) and intersection difference (dist) as parameters to analyze the quality of the clusters output by the clustering algorithm. The Cheeger ratio for a cluster S, as explained in section §3.6.1, is given by $\Phi_s = \frac{\partial}{min(\mathsf{vol_S}, \mathsf{vol_{G \setminus S}})}$. Note that a lower Cheeger ratio implies a cluster of higher quality. We report the Cheeger ratio of the cluster output by the cleartext algorithm (which operates with floating-point as well as fixed-point representation) and our secure algorithm (which operates over fixed-point representation) in Table 3.6. The intersection difference, dist, measures the similarity between two sets. Given two clusters S and S' sorted with respect to $\boldsymbol{\rho}[\mathsf{v}]/\mathsf{deg_v}$, the intersection difference of S' with respect to S is given by $\mathsf{dist}(S, S') = \frac{1}{n}\sum_{i=1}^{n} \frac{|(S_i \oplus S'_i)|}{2i}$. Here, S_i, S'_i are sets containing first i elements of S and S', respectively, and $S_i \oplus S'_i = (S_i \setminus S'_i) \bigcup (S'_i \setminus S_i)$. The values of set intersection difference lie between $[0, 1]$ where it is 0 for absolutely identical sets and 1 for disjoint sets. We compute the dist of the cluster generated by our secure protocol (operating over fixed-point representation) with respect to the cluster generated by the floating-point cleartext algorithm. We also compute the dist of cluster generated by the fixed-point cleartext algorithm with respect to that generated by the floating-point cleartext algorithm to showcase the effect of moving from floating-point computation to fixed-point computation. These values are reported in Table 3.6. As evident from Table 3.6, the Cheeger ratio of the output clusters in the three variants of the clustering algorithm is small and almost similar. This indicates that the quality of the cluster output by the secure protocol is similar to that output by the cleartext algorithm. Further, the set intersection distance is also very close to 0. This, too, implies the cluster output by the secure protocol is nearly identical to the cluster generated by the cleartext algorithm.

3.7.2 Secure computation

To show the practicality of our secure protocols, we benchmark and report their performance. We describe the benchmark environment and parameters first, followed by our observations.

Model	$\|V\|$	ς	Minimum Cheeger ratio			Intersection Difference	
			Float	Fixed	Secure	Fixed	Secure
Small world	100	100	0.24223	0.24221	0.24221	0.01149	0.01942
	500	500	0.1737	0.17507	0.17466	0.07723	0.03248
	800	500	0.14965	0.14964	0.14964	0.02118	0.01312
	1000	500	0.17176	0.17224	0.17218	0.01866	0.07169
Powerlaw cluster	100	20	0.63963	0.63963	0.63963	0.03439	0.01587
	500	100	0.45454	0.45452	0.45452	0.02120	0.06819
	800	100	0.4923	0.49227	0.49227	0.01830	0.04332
Preferential Attachment	100	100	0.45915	0.45913	0.45913	0.01252	0.01234
	500	500	0.45213	0.45211	0.45211	0.01761	0.02174
	800	500	0.54759	0.54747	0.54757	0.01809	0.02302
YouTube	1134890	500	0.61556	0.61433	0.61433	0.01020	0.01281

Table 3.6: Cluster quality with respect to Cheeger ratio and intersection difference. $|V|$ denotes the number of vertices and ς denotes the target cluster volume. We set the target Cheeger ratio, ϕ, to 0.1 for all the graphs. The choice of parameters ς and ϕ follow from [54].

Benchmark environment and parameters We report results in LAN (16 Gbps bandwidth) using n1-standard-64 instances of Google Cloud with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors, 64 vCPUs, and 240 GB of RAM Memory. Our protocols are implemented in Python over a 64-bit ring. We instantiate the communication layer between the parties using the PyTorch library. We use the pyaes library for AES and hashlib for generating SHA256 hash. Our code accounts for multithreading. We note that our benchmarking code is not fully optimized for industry-grade use. We also note that a C++-based implementation can give better performance compared to Python and witness improvements that are at least an order better. We consider the run time of protocols as the parameter for performance analysis and account for online as well as preprocessing costs when doing so.

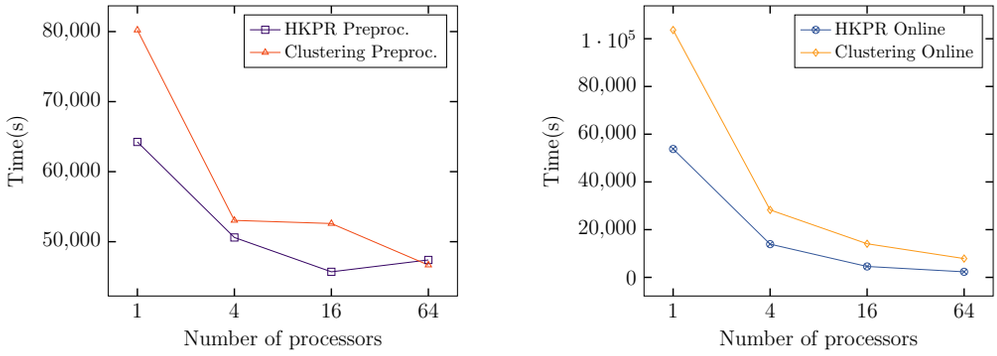
3.7.2.1 Clustering and HKPR metric computation

Since the complexity of graph propagation and clustering algorithms depend on the size of the edge list (G), we analyze these algorithms by varying $|V|+|E|$ between 100 and 10^6. We estimate the performance in a multiprocessor setting with 64 processors and report the maximum time taken by a processor. GraphSC provides a way to perform Scatter, Gather operations in parallel, which we use in the multiprocessor setting to ensure that our algorithms do not require linear complexity in $|V| + |E|$. To perform the sort operation in parallel, we use bitonic sort, which can be performed in parallel at the circuit level [195]. However, it is not trivial to perform the

shuffle in a parallel setting. Hence, the reported times for multiprocessor setting account for a non-optimized version of the secure shuffle.

For secure graph propagation, we estimate the run time for one iteration of the algorithm. Note that since all the graph propagation metrics can be computed via Π_{SGP}, the cost of evaluation of all these is the same. As expected and is evident from Table 3.7, the run time of the algorithm increases with increasing $|\mathsf{V}| + |\mathsf{E}|$ in the multiprocessor setting. We perform similar benchmarks for the secure clustering algorithm. Unlike the algorithm in [54], our algorithm takes the graph propagation vector ($\boldsymbol{\rho}$) as input, and hence, the run times reported do not account for the HKPR computation time. We report the results in Table 3.8 and observe the same trend as seen for secure graph propagation. Further, we note that the protocol for clustering can be optimized if the number of nodes in the graph is known and the same is accounted for in our benchmarks.

Fig. 3.14 gives a visual representation of variation in the run time of graph propagation and clustering algorithm when the number of processors is varied. In comparison to the single processor setting, the parallel variant of the algorithm in the multiprocessor setting gives up to 1.38× improvement in the preprocessing time and 23.4× improvement in the online time for HKPR computation. For clustering, the improvements are up to 1.75× in the preprocessing and 14.4× in the online phase.



(a) Preprocessing time.

(b) Online time.

Figure 3.14: Run times of HKPR and clustering for a graph of size 10^6 for varying the number of processors.

3.7.2.2 Shuffle

Shuffle being an important primitive, we empirically evaluate its performance under various parameters and compare them against their state-of-the-art counterparts. Following prior works, here we consider run time and communication of protocols as the parameters for comparison.

$\lvert V\rvert + \lvert E\rvert$	Preprocessing(s)	Online(s)
10^2	4.09	0.199
10^3	44.07	2.19
10^4	464.32	22.99
10^5	4310.00	261.00
10^6	47088.15	2397.34

Table 3.7: HKPR: Parallel computation for varying $|V| + |E|$ using 64 processors.

$\lvert V\rvert + \lvert E\rvert$	Preprocessing(s)	Online(s)
10^2	4.32	0.77
10^3	44.89	7.01
10^4	412.43	76.56
10^5	4681.01	763.83
10^6	46370.14	7929.82

Table 3.8: Clustering: Parallel computation for varying $|V| + |E|$ using 64 processors.

We account for online as well as total (preprocessing + online) costs when doing so. To capture the combined effect of both these parameters, we additionally report online throughput (TP).

We begin by comparing Ruffle to the shuffle protocol of [80] and [13] for the case of a single invocation of shuffle. Table 3.9 reports the online phase comparisons to capture the fast response time and the communication involved. Observe that Ruffle clearly outperforms both [80, 13]. Concretely, we observe improvements up to 15× in run time and 2.5× in communication over [80]. When compared to [13], Ruffle has an improvement of up to 11.2× in run time and 2.5× in communication. The improvements in the run time and communication are reflected in a high throughput, which captures the number of such single invocations that can be performed in parallel. The improvements in throughput range up to 5.5× and 2.2× with respect to [80] and [13], respectively. When considering the overall cost, we note that Ruffle fares better than [80] but is slightly higher, yet comparable, to that of [13]. We report this in Table 3.10 for completeness. We remark that $\mathsf{Ruffle}_{\mathsf{ind}}$ has the same complexity as Ruffle for a single shuffle, while $\mathsf{Ruffle}_{\mathsf{cmp}}$ is not apt for single shuffle invocation due to its higher preprocessing cost.

To capture the improvements of $\mathsf{Ruffle}_{\mathsf{ind}}$ and $\mathsf{Ruffle}_{\mathsf{cmp}}$, we benchmark their performance for multiple sequential shuffle invocations, i.e., scenarios of Independent-Shuffles and Composed-Shuffles. Recall that $\mathsf{Ruffle}_{\mathsf{ind}}$ is apt for Independent-Shuffles while $\mathsf{Ruffle}_{\mathsf{cmp}}$ for Composed-Shuffles. Since [13] outperforms [80] (as evident from Table 3.9 and Table 3.10), we restrict to comparing

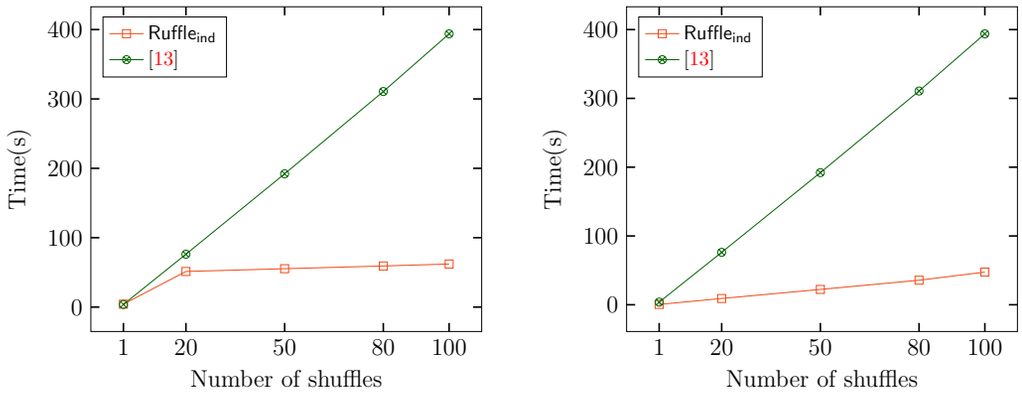
|T|	Protocol	Time (s)	Comm. (MB)	TP (per min)
10^3	Ruffle	0.005	0.092	12000.000
	[13]	0.056	0.231	5415.162
	[80]	0.075	0.228	2181.421
10^4	Ruffle	0.046	0.915	1200.000
	[13]	0.434	2.318	593.589
	[80]	0.718	2.289	218.177
10^5	Ruffle	0.457	9.155	120.000
	[13]	3.959	22.918	59.935
	[80]	7.692	22.888	21.818
10^6	Ruffle	7.033	91.553	12.000
	[13]	49.577	228.912	5.999
	[80]	95.089	228.881	2.181

Table 3.9: Online complexity of shuffle for varying table sizes for a single shuffle invocation.

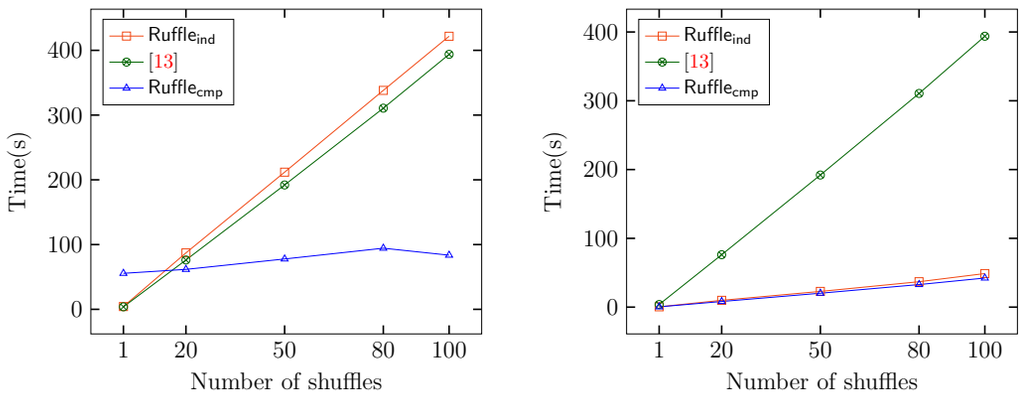
|T|	Protocol	Time (s)	Comm. (MB)
10^3	Ruffle	0.062	0.323
	[13]	0.056	0.258
	[80]	0.079	0.427
10^4	Ruffle	0.504	3.232
	[13]	0.434	2.318
	[80]	0.794	4.272
10^5	Ruffle	4.211	32.074
	[13]	3.959	22.919
	[80]	8.012	42.724
10^6	Ruffle	55.559	320.465
	[13]	49.577	228.912
	[80]	98.576	427.246

Table 3.10: Total complexity of shuffle for varying table sizes for single shuffle invocation.

$\mathsf{Ruffle}_{\mathsf{ind}}$ and $\mathsf{Ruffle}_{\mathsf{cmp}}$ in their respective settings against [13]. Further, to capture improvements $\mathsf{Ruffle}_{\mathsf{cmp}}$ protocol brings over $\mathsf{Ruffle}_{\mathsf{ind}}$, we also report the cost for performing $\mathsf{Ruffle}_{\mathsf{ind}}$ in the scenario of Composed-Shuffles. The comparison for varying number of shuffle invocations is



(a) Total time for Independent-Shuffles. (b) Online time for Independent-Shuffles.



(c) Total time for Composed-Shuffles. (d) Online time for Composed-Shuffles.

Figure 3.15: Comparison of Ruffle$_{ind}$, Ruffle$_{cmp}$, [13] in terms of online and total time for scenario of Independent-Shuffles and Composed-Shuffles for varying number of shuffle invocations and table size of 10^5.

reported in Fig. 3.15 (and Table 3.11). We make the following observations:

- The cost of [13] remains the same for Independent-Shuffles and Composed-Shuffles since it is indifferent to both.
- We infer the following with respect to the online complexity. Irrespective of the scenario and the number of shuffle invocations, recall from Table 3.2 that Ruffle$_{ind}$ and Ruffle$_{cmp}$ are comparable since their online phase is the same, except for the extra computation required in Ruffle$_{cmp}$. Hence, as expected, Ruffle$_{ind}$ (and thereby Ruffle$_{cmp}$) outperforms [13] by up to 10×.
- We infer the following with respect to the overall run time. For a single shuffle invocation, both Ruffle$_{ind}$ (i.e. Ruffle for $m = 1$) and Ruffle$_{cmp}$ have a slightly higher run time than [13]. However, starting from as low as two invocations, Ruffle$_{ind}$ begins to outperform [13] for Independent-Shuffles. This is justified as follows—since Ruffle$_{ind}$'s online phase is faster than that of [13], performing the preprocessing for m shuffles in parallel results in improving the overall

Number of shuffles	Protocol	Online			Total		
		Time(s)	Comm.(MB)	TP (per min)	Time(s)	Comm.(MB)	Monetary cost (USD)
1	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Independent-Shuffles)	0.38	9.16	120.00	4.21	32.06	0.020
	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Composed-Shuffles)	0.38	9.16	120.00	4.21	32.06	0.020
	$\mathsf{Ruffle}_{\mathsf{cmp}}$ (Composed-Shuffles)	0.46	9.16	120.00	4.35	41.21	0.022
	[13]	3.81	22.91	59.93	3.81	22.91	0.016
2	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Independent-Shuffles)	0.92	18.31	60.00	5.16	64.13	0.030
	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Composed-Shuffles)	0.92	18.31	60.00	10.21	64.13	0.044
	$\mathsf{Ruffle}_{\mathsf{cmp}}$ (Composed-Shuffles)	0.98	18.31	60.00	5.46	82.44	0.035
	[13]	7.58	45.81	29.95	7.62	45.81	0.032
25	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Independent-Shuffles)	10.30	228.88	4.80	53.72	801.56	0.349
	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Composed-Shuffles)	10.30	228.88	4.80	105.25	801.56	0.491
	$\mathsf{Ruffle}_{\mathsf{cmp}}$ (Composed-Shuffles)	11.18	228.88	4.80	61.68	1030.44	0.429
	[13]	95.24	572.68	2.39	95.74	572.68	0.408
50	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Independent-Shuffles)	20.50	457.76	2.40	55.31	1603.33	0.556
	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Composed-Shuffles)	20.50	457.76	2.40	211.82	1603.33	0.986
	$\mathsf{Ruffle}_{\mathsf{cmp}}$ (Composed-Shuffles)	20.53	457.76	2.40	77.76	2060.74	0.733
	[13]	192.06	1145.55	1.20	194.29	1145.55	0.817
100	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Independent-Shuffles)	40.18	960.01	1.20	62.09	3206.66	0.978
	$\mathsf{Ruffle}_{\mathsf{ind}}$ (Composed-Shuffles)	40.18	960.01	1.20	421.76	3206.66	1.968
	$\mathsf{Ruffle}_{\mathsf{cmp}}$ (Composed-Shuffles)	42.29	960.01	1.20	83.60	3206.66	1.037
	[13]	393.82	2402.40	0.59	395.91	2291.11	1.660

Table 3.11: Comparison of $\mathsf{Ruffle}_{\mathsf{ind}}$, $\mathsf{Ruffle}_{\mathsf{cmp}}$, [13] with respect to the scenario of Independent-Shuffles and Composed-Shuffles for varying number of shuffle invocations and table size of 10^5. Note that the cost of [13] remains the same for both scenarios.

complexity. This improvement is not seen in [13] since the m shuffles that can be performed in parallel during our preprocessing are required to be performed sequentially in case of [13], which adds to the overhead. We see improvements of up to 6.4× in this case. On the other hand, for Composed-Shuffles, [13] continues to outperform $\mathsf{Ruffle}_{\mathsf{ind}}$ for the following reason. The composition of shuffles induces a sequential nature in the preprocessing phase of $\mathsf{Ruffle}_{\mathsf{ind}}$ (which indeed is the complete protocol of [13]). The computations performed additionally in the online phase of $\mathsf{Ruffle}_{\mathsf{ind}}$ render its overall complexity slightly higher than that of [13]. To break this chain of sequential shuffles in the preprocessing phase, $\mathsf{Ruffle}_{\mathsf{cmp}}$ was designed to outperform $\mathsf{Ruffle}_{\mathsf{ind}}$ (and thereby [13]), where we see improvements of up to 4.7× with respect to [13].

- To capture the effect of both total run time and total communication, we additionally report the monetary cost in Table 3.11, which is the price paid for performing the secure shuffle computation. This is calculated using the pricing of Google Cloud Platform [205], where for 1GB and 1 hour of usage, the costs are USD 0.12 and USD 3.3025, respectively. With respect to

the monetary cost, we note that for a small number of shuffle invocations (≤ 25), our protocols have a slightly higher monetary cost in comparison to [13]. However, as the number of shuffle invocations increases (> 25), the savings in run time seen in our shuffle protocols compensate for the increased communication. Thus, both $\mathsf{Ruffle}_{\mathsf{ind}}$, $\mathsf{Ruffle}_{\mathsf{cmp}}$ outperform [13] in terms of monetary cost.

3.7.2.3 Anonymous broadcast

As described earlier, since anonymous broadcast directly builds on $\mathsf{Ruffle}_{\mathsf{ind}}$, we next briefly discuss how this application benefits in terms of efficiency by relying on $\mathsf{Ruffle}_{\mathsf{ind}}$. Recall that anonymous broadcast, as the name suggests, enables a set of N clients to anonymously broadcast their messages while guaranteeing that none learns about the association between a message and the identity of its sender. Instead of requiring the clients to send their messages to a centralized server, which can output the randomly shuffled messages back to the clients, we rely on a distributed solution to guarantee client privacy. At a high level, to achieve anonymous broadcast, the clients secret-share their messages to a set of three servers (the three parties in $\mathcal{P}$, henceforth interchangeably called servers), who invoke a secure shuffle protocol on the same and reconstruct the shuffled output. The protocol can be described in the following steps.
1. *Input sharing and consistency check:* Each client wanting to broadcast a message receives randomness, using which it generates $[\![\cdot]\!]$-shares of its message. On receiving shares of a client's message, servers verify if these are malformed. If so, they discard the message. Input sharing can thus be performed via the steps described in §2.3.
2. *Shuffle:* Assuming N messages pass the verification, servers securely shuffle the N-sized table using $\mathsf{Ruffle}_{\mathsf{ind}}$ described in §3.4.3.2.
3. *Output reconstruction:* On receiving the output shares after executing $\mathsf{Ruffle}_{\mathsf{ind}}$, servers reconstruct the shuffled table using the steps described in §2.3. The shuffled table is then broadcast to clients.

Comparison of our anonymous broadcast system with [80]'s We first compare the proposed anonymous broadcast system with the most efficient shuffle-based 3-server system in Clarion [80] in terms of the guarantees provided by the two systems, and then in terms of performance. Note that Clarion provides security with abort while our protocol allows attaining the strongest security notion of GOD. Further, Clarion provides no way of distinguishing between a malicious act by a server and a client, and the system rejects the request from a client whenever the verification for input consistency fails. Specifically, since Clarion only provides security with abort, it allows a malicious server to make the input consistency check (with re-

spect to an honest client's input) fail by aborting the computation. Thus, an honest client may be dropped due to misbehaviour by a malicious server, and hence Clarion does not guarantee censorship resistance. Note here that a system is said to provide censorship resistance if a malicious server cannot discard an honest client's message from the system. On the other hand, in our system, the only way a malicious server can cheat to discard an honest client's message m is by broadcasting an incorrect β_{m} during the input consistency check. Since, at most, one server among the three is malicious, there will always be agreement among the honest servers with respect to the correct β_{m}. Thus, our protocol does not allow discarding an honest client's messages, thereby attaining censorship resistance.

We now empirically compare the two anonymous broadcast systems. Since the complexity of anonymous broadcast varies based on the number of clients (N) as well as their message size, we compare with respect to these parameters. For a fair comparison, we implement all the protocols in Python, including that of [80]. Thus, costs reported for prior works are higher than that reported in the original works due to some operations, such as random number generation, which can be performed much more efficiently in the Go-based implementation of [80]. Hence, the concrete improvements over [80] reported next capture the relative improvements with respect to the underlying protocols. That is, we do not account for the system-level optimizations that may have been included as a part of the implementations in the original work of [80]. However, we note that the reported communication costs are invariant with the implementation.

When varying the number of clients, we analyze the server-side complexity and report the performance in Table 3.12, which accounts for—checking the consistency of clients' messages (32 bytes in size) where the check is performed in parallel for N clients, shuffling the N-sized table, and reconstruction of the shuffled result. Observe from Table 3.12 that our anonymous broadcast system outperforms [80] in every aspect. This can be attributed not only to the use of our efficient shuffle protocol but also due to the simplicity of the input sharing and consistency check, and output reconstruction. On the other hand, [80] relies on several MAC verifications and encryption operations, which render the system of [80] less efficient. The improvements we observe with respect to online and total time is up to 29× and 13×, respectively, whereas that of online and total communication is up to 2×.

The effect of varying the client message size on run time and communication with respect to the server is reported in Table 3.13. Our system outperforms [80] in terms of both. Concretely, with respect to online and total time, we see improvements up to 39× and 9×, respectively. With respect to online and total communication, we see improvements up to 1.2× and 1.3×.

Table 3.12 and Table 3.13 do not account for the time/communication required to share a client's input. Hence, to showcase the overhead of input sharing, on both the client and the

N	Anonymous broadcast	Online		Total	
		Time (s)	Comm. (MB)	Time (s)	Comm. (MB)
10^3	Ours	0.01	0.36	0.09	0.62
	[80]	0.20	0.76	1.11	1.23
10^4	Ours	0.06	3.66	0.69	5.97
	[80]	1.88	7.63	10.19	12.34
10^5	Ours	0.61	36.62	6.73	59.53
	[80]	20.59	76.29	105.88	123.59
10^6	Ours	8.64	366.21	107.52	595.12
	[80]	248.99	762.94	1082.53	1235.96

Table 3.12: Comparison of online run time and communication of servers for varying number of clients and message size of 32 bytes.

Message Size	Anonymous broadcast	Online		Total	
		Time (s)	Comm. (MB)	Time (s)	Comm. (MB)
32B	Ours	0.64	36.62	6.75	59.53
	[80]	20.59	76.29	104.55	123.59
160B	Ours	1.40	183.12	11.47	279.25
	[80]	49.97	247.96	135.96	395.51
1KB	Ours	6.97	1145.14	66.10	1721.21
	[80]	269.26	1373.29	553.28	2224.16

Table 3.13: Comparison of online run time and communication of servers for varying message size and clients of $\mathsf{N} = 10^5$.

server, we report the costs in Table 3.14. Since this overhead is dependent on the client message size, Table 3.14 also account for the same. Recall that our system additionally requires the client to wait to receive the preprocessing data from the server. Despite this, the time for which a client has to remain online in our system is 18× lesser in comparison to [80]. The higher cost of [80] can be attributed to the need for PRG (pseudorandom generator) invocations, encryption of message followed by MAC tag computation at the client. This is unlike our system, which relies on simple operations such as XOR. On the other hand, since [80] requires the clients to communicate to only two servers instead of the three servers as required in our case, they have lesser communication. The reduced time a client has to remain online comes at the cost of server-to-client communication, which is absent in [80]. This, we note, is a small price paid. Thus, our realization of anonymous broadcast not only provides improved efficiency but also offers censorship resistance and allows attaining the improved security of GOD.

Message Size	Anonymous broadcast	Client time (ms)	Client-server Communication (KB)	Server-client Communication (KB)
32B	Ours	0.13	0.09	0.47
	[80]	2.34	0.16	-
160B	Ours	0.12	0.47	1.22
	[80]	6.05	0.41	-
1KB	Ours	1.74	3.00	6.97
	[80]	30.55	2.06	-

Table 3.14: Comparison of client-side and server-side complexity for input sharing by one client.

3.8 Security proofs

The simulation-based security proofs for the designed primitives and the local clustering protocol are presented in this section. At a high level, observe that the designed protocols rely on invoking protocols given in SWIFT [136] whose security was established therein in the standard real-world/ideal-world simulation paradigm. Hence, the security of the designed protocols follows directly from the security of the underlying protocols of SWIFT. We let the following denote the ideal functionalities for the sub-protocols provided by SWIFT.

1. $\mathcal{F}_{\mathsf{Mul}}$: Takes as input $[\![\cdot]\!]$-shares (or equivalently $[\![\cdot]\!]^{\mathbf{B}}$-shares) of x, y and outputs $[\![\cdot]\!]$-shares (or equivalently $[\![\cdot]\!]^{\mathbf{B}}$-shares) of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$.

2. $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$: Takes as input $[\![\cdot]\!]$-shares of x, y and outputs $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ by truncated by f bits using probabilistic truncation.

3. $\mathcal{F}_{\mathsf{3\text{-}Mul}}$: Takes as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathsf{a}, \mathsf{b}, \mathsf{c}$ and outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathsf{z} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c}$.

4. $\mathcal{F}_{\mathsf{4\text{-}Mul}}$: Takes as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$ and outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathsf{z} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c} \cdot \mathsf{d}$.

5. $\mathcal{F}_{\mathsf{Not}}$: Takes as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of a value x and outputs the $[\![\cdot]\!]^{\mathbf{B}}$-shares of 1's complement of x, denoted as $\bar{\mathsf{x}}$.

6. $\mathcal{F}_{\mathsf{Sel}}$: Takes as input $[\![\cdot]\!]$-shares of $\mathsf{x}_0, \mathsf{x}_1$ and $[\![\cdot]\!]^{\mathbf{B}}$-shares of a bit b, and outputs $[\![\mathsf{x}_\mathsf{b}]\!]$.

7. $\mathcal{F}_{\mathsf{A2B}}$: Takes as input $[\![\cdot]\!]$-shares of a value x, and outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares for its equivalent Boolean representation.

8. $\mathcal{F}_{\mathsf{B2A}}$: Takes as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of the Boolean representation of a value x, and outputs $[\![\cdot]\!]$-shares for its equivalent arithmetic representation.

9. $\mathcal{F}_{\mathsf{Comp}}$: Takes as input $[\![\cdot]\!]$-shares of x, y and outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares of b such that $\mathsf{b} = 1$ if $\mathsf{x} < \mathsf{y}$, else $\mathsf{b} = 0$.

We use the simulation strategy as described in SWIFT [136], where we simulate the end-to-end computation of a function f for which the designed primitives serve as building blocks. The simulation begins with the simulator $\mathcal{S}$ emulating the shared-key setup $\mathcal{F}_{\mathsf{Setup}}$ functionality (Fig. 2.3) and giving the respective keys to the adversary $\mathcal{A}$. This is followed by the input sharing phase in which $\mathcal{S}$ extracts the input of $\mathcal{A}$, using the known keys, and sets the inputs of the honest parties to be 0 (see simulator for input sharing in [136]). This allows $\mathcal{S}$ to have access to the shares of the honest parties. Since $\mathcal{S}$ knows all the inputs, it can honestly carry out the computation and compute all the intermediate values as required for simulating the view of $\mathcal{A}$. $\mathcal{S}$ proceeds to simulate the various sub-protocols required to compute f in topological order using the aforementioned values. Observe that since $\mathcal{S}$ knows $\mathcal{A}$'s inputs, it can detect any malicious behaviour carried out by $\mathcal{A}$. Finally, depending on $\mathcal{A}$'s behaviour, $\mathcal{S}$ invokes the ideal functionality for the function f with $\mathcal{A}$'s input, obtains the function output and forwards the same to $\mathcal{A}$ during the output reconstruction phase. For simplicity of presentation, we stick to a modular approach of providing simulation steps for each of the (newly designed) sub-protocols, as done in [136]. Note that carrying out these simulation steps in respective topological order (starting from $\mathcal{F}_{\mathsf{Setup}}$, the input sharing phase, all the intermediate sub-protocols, and output reconstruction) results in simulating the computation of the desired function f.

3.8.1 Security of the designed primitives

Prefix OR:

The ideal functionality for prefix OR appears in Fig. 3.16.

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

$\mathcal{F}_{\mathsf{PreOr}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

- Receive as input the $[\![\cdot]\!]^{\mathbf{B}}$-shares of bits $\mathsf{x}_{\ell-1}, \ldots, \mathsf{x}_0$ from all parties.
- Reconstruct $\mathsf{x}_{\ell-1}, \ldots, \mathsf{x}_0$ using the shares of honest parties.
- Compute $\mathsf{y}_i = \vee_{j=i}^{\ell-1} \mathsf{x}_j$ for $i \in \{0, \ldots, \ell-1\}$.
- Generate $[\![\cdot]\!]^{\mathbf{B}}$-shares of y_i for $i \in \{0, \ldots, \ell-1\}$ and send $(\mathsf{Output}, [\![\mathsf{y}_i]\!]_s^{\mathbf{B}})$ to $P_s \in \mathcal{P}$.

Figure 3.16: Ideal functionality for prefix OR.

Lemma 3.1 (Security) *Protocol* Π_{PreOr} *(Fig. 3.2) securely realizes* $\mathcal{F}_{\mathsf{PreOr}}$ *(Fig. 3.16) in the computational 3PC setting against a malicious adversary* $\mathcal{S}$ *in the* $(\mathcal{F}_{\mathsf{Mul}}, \mathcal{F}_{\mathsf{3\text{-}Mul}}, \mathcal{F}_{\mathsf{4\text{-}Mul}}, \mathcal{F}_{\mathsf{Not}})$*-hybrid model.*

Proof: The simulator $\mathcal{S}_{\mathsf{PreOR}}$ appears in Fig. 3.17.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{PreOR}}$ honestly executes the protocol steps and proceeds as follows.

– Define the shares of the sub-blocks $[\![\boldsymbol{b}_i^0]\!]^{\mathbf{B}} = [\![\mathsf{x}_i]\!]^{\mathbf{B}}$ for $i \in \{0, \ldots, \ell-1\}$ on behalf of the honest parties using their shares of x_i, and set $k = \ell$.

– for $j = 0$ to $\lfloor \log_4(\ell) \rfloor$ do: (j denotes round number)

∘ for $i = \lfloor \frac{k}{4} \rfloor$ to 1 do: (i denotes block number)

- Define shares of the blocks as $[\![\boldsymbol{t}_i^j]\!]^{\mathbf{B}} = \left([\![\boldsymbol{b}_{4i-1}^j]\!]^{\mathbf{B}}, [\![\boldsymbol{b}_{4i-2}^j]\!]^{\mathbf{B}}, [\![\boldsymbol{b}_{4i-3}^j]\!]^{\mathbf{B}}, [\![\boldsymbol{b}_{4i-4}^j]\!]^{\mathbf{B}}\right)$ on behalf of the honest parties.

- For $i \in \{0, 1, \ldots, \mathsf{d}-1\}$, set

∘ $[\![\boldsymbol{b}'_{3,i}]\!]^{\mathbf{B}} = [\![\boldsymbol{b}_{3,i}]\!]^{\mathbf{B}}$

∘ Emulate $\mathcal{F}_{\mathsf{Mul}}$ on inputs $[\![\bar{\mathsf{z}_3}]\!]^{\mathbf{B}}, [\![\bar{\boldsymbol{b}}_{2,i}]\!]^{\mathbf{B}}$ where z_3 denotes the last bit of $\boldsymbol{b}_3$ and return $[\![\mathsf{v}]\!]^{\mathbf{B}}$ to $\mathcal{A}$ where $\mathsf{v} = \bar{\mathsf{z}_3} \cdot \bar{\boldsymbol{b}}_{2,i}$. Emulate $\mathcal{F}_{\mathsf{Not}}$ on input $[\![\mathsf{v}]\!]^{\mathbf{B}}$ and return $[\![\boldsymbol{b}'_{2,i}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.

∘ Emulate $\mathcal{F}_{\mathsf{3\text{-}Mul}}$ on inputs $[\![\bar{\mathsf{z}_3}]\!]^{\mathbf{B}}, [\![\bar{\mathsf{z}_2}]\!]^{\mathbf{B}}, [\![\bar{\boldsymbol{b}}_{1,i}]\!]^{\mathbf{B}}$ where $\mathsf{z}_3, \mathsf{z}_2$ denote the last bits of $\boldsymbol{b}_3, \boldsymbol{b}_2$, respectively, and return $[\![\mathsf{v}]\!]^{\mathbf{B}}$ to $\mathcal{A}$ where $\mathsf{v} = \bar{\mathsf{z}_3} \cdot \bar{\mathsf{z}_2}, \cdot \bar{\boldsymbol{b}}_{1,i}$. Emulate $\mathcal{F}_{\mathsf{Not}}$ on input $[\![\mathsf{v}]\!]^{\mathbf{B}}$ and return $[\![\boldsymbol{b}'_{1,i}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.

∘ Emulate $\mathcal{F}_{\mathsf{4\text{-}Mul}}$ on inputs $[\![\bar{\mathsf{z}_3}]\!]^{\mathbf{B}}, [\![\bar{\mathsf{z}_2}]\!]^{\mathbf{B}}, [\![\bar{\mathsf{z}_1}]\!]^{\mathbf{B}}, [\![\bar{\boldsymbol{b}}_{0,i}]\!]^{\mathbf{B}}$ where $\mathsf{z}_3, \mathsf{z}_2, \mathsf{z}_1$ denote the last bits of $\boldsymbol{b}_3, \boldsymbol{b}_2, \boldsymbol{b}_1$, respectively, and return $[\![\mathsf{v}]\!]^{\mathbf{B}}$ to $\mathcal{A}$ where $\mathsf{v} = \bar{\mathsf{z}_3} \cdot \bar{\mathsf{z}_2} \cdot \bar{\mathsf{z}_1} \cdot \bar{\boldsymbol{b}}_{0,i}$. Emulate $\mathcal{F}_{\mathsf{Not}}$ on input $[\![\mathsf{v}]\!]^{\mathbf{B}}$ and return $[\![\boldsymbol{b}'_{0,i}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.

- Set $[\![\boldsymbol{b}_i^{j+1}]\!]^{\mathbf{B}} = \left([\![\boldsymbol{b}'_3]\!]^{\mathbf{B}}, [\![\boldsymbol{b}'_2]\!]^{\mathbf{B}}, [\![\boldsymbol{b}'_1]\!]^{\mathbf{B}}, [\![\boldsymbol{b}'_0]\!]^{\mathbf{B}}\right)$.

∘ $k = \lfloor \frac{k}{4} \rfloor$

– Send $[\![\boldsymbol{b}_3^{\lfloor \log_4(\ell) \rfloor + 1}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.

Figure 3.17: Simulator for Π_{PreOr}.

The simulator begins by defining the blocks and sub-blocks. It then emulates $\mathcal{F}_{\mathsf{Mul}}, \mathcal{F}_{\mathsf{3\text{-}Mul}}$, $\mathcal{F}_{\mathsf{4\text{-}Mul}}, \mathcal{F}_{\mathsf{Not}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from

the simulator. This is indistinguishable from its view in the real world, where it sees random values.

□

Division:

The ideal functionality for computing approximate reciprocal appears in Fig. 3.18.

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

$\mathcal{F}_{\mathsf{AppRec}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

- Receive as input the $[\![\cdot]\!]$-shares of b from all parties.
- Reconstruct b using the shares of honest parties.
- Set $\mathsf{z} = 1$ if $\mathsf{b} = 0$, else set $\mathsf{z} = 0$. Compute $\mathsf{w} = 1/\mathsf{b}$ in fixed-point arithmetic representation as follows using probabilistic truncation when performing multiplication.
  - If $\mathsf{b} \geq 0$, normalize it to $\mathsf{b}' \in [0.5, 1)$, else if $\mathsf{b} < 0$, normalize it to $\mathsf{b}' \in (-1, -0.5]$ by computing $\mathsf{b}' = \mathsf{bv}$, where v is the scaling factor used to normalize b.
  - If $\mathsf{b} \geq 0$, approximate $\mathsf{w}' = 1/\mathsf{b}'$ as $2.9142 - 2\mathsf{b}'$, else $\mathsf{w}' = 1/\mathsf{b}'$ is approximated to $-2.9142 - 2\mathsf{b}'$.
  - Set $\mathsf{w} = \mathsf{w}' \cdot \mathsf{v}$, where v is the scaling factor used to obtain the normalized $\mathsf{b}' = \mathsf{bv}$.
- Generate $[\![\cdot]\!]$-shares of w, z and send $(\mathsf{Output}, [\![\mathsf{w}]\!]_s, [\![\mathsf{z}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 3.18: Ideal functionality for approximate reciprocal.

The ideal functionality for computing division appears in Fig. 3.19.

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

$\mathcal{F}_{\mathsf{Div}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

- Receive as input the $[\![\cdot]\!]$-shares of a, b from all parties.
- Reconstruct a, b using the shares of honest parties.
- Set $\mathsf{z} = 1$ if $\mathsf{b} = 0$, else set $\mathsf{z} = 0$. Compute $\mathsf{d} = \mathsf{a}/\mathsf{b}$ using Goldschmidt's approximate division method in fixed-point arithmetic representation, where $1/\mathsf{b}$ is computed using the approximate reciprocal approach described in Fig. 3.18.
- Generate $[\![\cdot]\!]$-shares of d, z and send $(\mathsf{Output}, [\![\mathsf{d}]\!]_s, [\![\mathsf{z}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 3.19: Ideal functionality for division.

Lemma 3.2 (Security) *Protocol* Π_{AppRec} *(Fig. 3.3) securely realizes* $\mathcal{F}_{\mathsf{AppRec}}$ *(Fig. 3.18) in the computational 3PC setting against a malicious adversary* $\mathcal{S}$ *in the* $(\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{Mul}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Not}})$*-hybrid model.*

Proof: The simulator for Π_{AppRec} appears in Fig. 3.20.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{AppRec}}$ honestly executes the protocol steps and proceeds as follows.

- Emulate $\mathcal{F}_{\mathsf{A2B}}$ on input $[\![\mathsf{b}]\!]$ and return $[\![\mathsf{b}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- On behalf of the honest parties, honestly compute $[\![\mathsf{b}'_i]\!]^{\mathbf{B}} = [\![\mathsf{b}_i]\!]^{\mathbf{B}} \oplus [\![\mathsf{b}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$ for $i = 0$ to $\mathsf{k}-2$.
- Emulate $\mathcal{F}_{\mathsf{PreOr}}$ on inputs $[\![\mathsf{b}'_{\mathsf{k}-2}]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{b}'_0]\!]^{\mathbf{B}}$ and output $[\![\mathsf{c}_{\mathsf{k}-2}]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{c}_0]\!]^{\mathbf{B}}$ to $\mathcal{A}$ and set $[\![\mathsf{c}_{\mathsf{k}-1}]\!]^{\mathbf{B}} = [\![0]\!]^{\mathbf{B}}$ on behalf of the honest parties.
- On behalf of the honest parties, honestly compute $[\![\mathsf{c}_i]\!]^{\mathbf{B}} = [\![\mathsf{c}_i]\!]^{\mathbf{B}} \oplus [\![\mathsf{c}_{i+1}]\!]^{\mathbf{B}}$ for $i = \mathsf{k}-2$ to 1.
- Emulate $\mathcal{F}_{\mathsf{B2A}}$ on inputs $[\![\mathsf{c}_0]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{c}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{v}]\!]$.
- Emulate $\mathcal{F}_{\mathsf{Mul}}$ on inputs $[\![\mathsf{b}_{\mathsf{k}-1}]\!]^{\mathbf{B}}, [\![\mathsf{c}_0]\!]^{\mathbf{B}}$ and output $[\![\mathsf{x}]\!]^{\mathbf{B}}$ to $\mathcal{A}$. Emulate $\mathcal{F}_{\mathsf{Not}}$ on $[\![\mathsf{x}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{z}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Mul}}$ on inputs $[\![\mathsf{b}]\!], [\![\mathsf{v}]\!]$ and output x to $\mathcal{A}$. Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $\alpha, -\alpha, [\![\mathsf{b}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$ where $\alpha = (2.9142)_{\mathsf{k}-1}$, and output $[\![\mathsf{y}]\!]$ to $\mathcal{A}$. Honestly compute $[\![\mathsf{w}']\!] = [\![\mathsf{y}]\!] - 2 \cdot [\![\mathsf{x}]\!]$.
- Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{v}]\!], [\![\mathsf{w}']\!], 2(\mathsf{k}-\mathsf{f}-1)$ to multiply $[\![\mathsf{v}]\!], [\![\mathsf{w}']\!]$ and truncate the output by $2(\mathsf{k}-\mathsf{f}-1)$ bits to generate $[\![\mathsf{w}]\!]$. Output $[\![\mathsf{w}]\!]$ to $\mathcal{A}$.

Figure 3.20: Simulator for Π_{AppRec}.

The protocol does not involve any interaction apart from what is required while invoking the protocols for $\Pi_{\mathsf{A2B}}, \Pi_{\mathsf{B2A}}, \Pi_{\mathsf{PreOr}}, \Pi_{\mathsf{Mul}}, \Pi_{\mathsf{Sel}}, \Pi_{\mathsf{NOT}}$. Hence, the simulator emulates $\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{Mul}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Not}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from the simulator. This is indistinguishable from its view in the real world, where it sees random values. □

Lemma 3.3 (Security) *Protocol* Π_{Div} *(Fig. 3.4) securely realizes* $\mathcal{F}_{\mathsf{Div}}$ *(Fig. 3.19) in the computational 3PC setting against a malicious adversary* $\mathcal{S}$ *in the* $(\mathcal{F}_{\mathsf{AppRec}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}})$*-hybrid model.*

Proof: The simulator for Π_{Div} appears in Fig. 3.21.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{Div}}$ honestly executes the protocol steps and proceeds as follows.

- Emulate $\mathcal{F}_{\mathsf{AppRec}}$ on input $[\![\mathsf{b}]\!]$ and output $[\![\mathsf{w}]\!], [\![\mathsf{z}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{b}]\!], [\![\mathsf{w}]\!], 0$ and output $[\![\mathsf{v}]\!]$ to $\mathcal{A}$ where $\mathsf{v} = \mathsf{b} \cdot \mathsf{w}$. Compute $[\![\mathsf{e}]\!] = \alpha - [\![\mathsf{v}]\!]$ on behalf of the honest parties, where $\alpha = (1)_{\mathsf{2f}}$.
- Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{a}]\!], [\![\mathsf{w}]\!], 0$ and output $[\![\mathsf{d}]\!]$ to $\mathcal{A}$.
- Do the following for $i = 1$ to θ-1.
  - Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{d}]\!], \alpha + [\![\mathsf{e}]\!], 2\mathsf{f}$, and output $[\![\mathsf{d}]\!]$ to $\mathcal{A}$.
  - Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $([\![\mathsf{e}]\!], [\![\mathsf{e}]\!], 2\mathsf{f})$ and output $[\![\mathsf{e}]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{d}]\!], \alpha + [\![\mathsf{e}]\!], 2\mathsf{f}$, and output $[\![\mathsf{d}]\!]$ to $\mathcal{A}$.

Figure 3.21: Simulator for Π_{Div}.

The simulator begins by emulating $\mathcal{F}_{\mathsf{AppRec}}$. Following this, it emulates $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ as per its invocation in the real-world protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from the simulator. This is indistinguishable from its view in the real world, where it sees random values. □

Shuffle:

Lemma 3.4 (security) *The shuffle protocol,* Π_{Shuffle} *(Fig. 3.8) securely realizes the functionality* $\mathcal{F}_{\mathsf{Shuffle}}$ *(Fig. 4.10) in the computational 3PC setting against a malicious adversary* $\mathcal{S}$ *in the* $\mathcal{F}_{\mathsf{Setup}}$*-hybrid model.*

Proof: At a high level, $\mathcal{S}$ begins by first emulating $\mathcal{F}_{\mathsf{Setup}}$ during which common keys are established with $\mathcal{A}$ that are used to sample the common randomness required throughout the protocol. Thus, $\mathcal{S}$ is aware of all the randomness used by $\mathcal{A}$, using which it can extract the input of $\mathcal{A}$ (specifically, $\mathcal{S}$ knows $[\alpha_{\mathsf{T}}]^{\mathbf{B}}, \beta_{\mathsf{T}}$ held by $\mathcal{A}$) as well as verify the correctness of messages sent by $\mathcal{A}$. Following this, it simulates the steps of the shuffle protocol. The simulation steps for a corrupt P_0 are provided next, where the corresponding simulator is denoted as $\mathcal{S}^{P_0}$. Analogously the corruption of P_1, P_2 can also be simulated. $\mathcal{S}^{P_0}$ proceeds as follows.

Preprocessing:

1. Using the keys commonly held with $\mathcal{A}$ (generated as part of $\mathcal{F}_{\mathsf{Setup}}$), sample the common randomness.

2. Simulate the steps of Shuffle-Pair pair using π_{02}. Receive the corresponding message from $\mathcal{A}$. If the received message is incorrect ($\mathcal{S}^{P_0}$ can verify the correctness of the received message since it possesses all the randomness used by $\mathcal{A}$ to send this message as $\mathcal{S}^{P_0}$ emulates $\mathcal{F}_{\mathsf{Setup}}$), set $\mathsf{flag}_{02} = 1$. Honestly simulate the steps of Set-Equality protocol.

3. Honestly simulate the steps of Shuffle-Pair using π_{12} followed by simulating the steps of Set-Equality.

4. Analogous to the case of Shuffle-Pair with π_{02}, simulate the steps of Shuffle-Pair pair using π_{01}. If an incorrect message is received from $\mathcal{A}$, set $\mathsf{flag}_{01} = 1$.

5. If $\mathsf{flag}_{02} = 1$, set $\mathsf{TTP} = P_1$. Else, if $\mathsf{flag}_{01} = 1$, $\mathsf{TTP} = P_2$.

Online:

1. Let $[\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{01}, [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{02}$ denote the partial shares of T_o generated towards $\mathcal{A}$ during preprocessing. Let $\beta_{\mathsf{T}_o} \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ be sampled randomly. Invoke the ideal functionality $\mathcal{F}_{\mathsf{Shuffle}}$ with $\mathcal{A}$'s $[\![\cdot]\!]$-shares of the table–$[\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{01}, [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{02}, \beta_{\mathsf{T}_o}$.

2. Receive $\mathsf{H}(\delta_{02})$ from $\mathcal{A}$ on behalf of P_1. Set $\mathsf{flag}_1 = 1$ if the received message is incorrect ($\mathcal{S}^{P_0}$ can verify the correctness of the received message since it possesses all the randomness used by $\mathcal{A}$ to send this message as $\mathcal{S}^{P_0}$ emulates $\mathcal{F}_{\mathsf{Setup}}$).

3. Receive δ_{01} from $\mathcal{A}$ on behalf of P_2.

4. If $\mathsf{flag}_1 = 1$, broadcast ("accuse", P_2, P_0, c_2, c_0), where $c_0 = \mathsf{H}(\delta_{02})$ as received from $\mathcal{A}$, and $c_2 = \mathsf{H}(\pi_{02}\,(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}))$ is honestly computed by $\mathcal{S}^{P_0}$.
   - If $\mathcal{A}$ broadcasts ("accuse", P_1), set $\mathsf{TTP} = P_2$.
   - Else set $\mathsf{TTP} = P_1$.

5. Else, if $\mathsf{flag}_1 = 0$, proceed as follows. Set $\delta_{12} = \beta_{\mathsf{T}_o}$ (as defined in step 1) and compute $\mathsf{H}(\delta_{12})$. Send $\delta_{12}, \mathsf{H}(\delta_{12})$ to $\mathcal{A}$ on behalf of honest P_1, P_2, respectively. Compute $c_1 = \mathsf{H}(\pi_{01}\,(\pi_{02}\,(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01}))$ and compare it with $c_0 = \mathsf{H}(\delta_{01})$ where δ_{01} was received from $\mathcal{A}$. If $c_0 \neq c_1$, set $\mathsf{flag}_2 = 1$.

6. If $\mathsf{flag}_2 = 1$, broadcast ("accuse", P_0, P_1, c_0, c_1).
   - If $\mathcal{A}$ broadcasts ("accuse", P_2), set $\mathsf{TTP} = P_1$.
   - Else set $\mathsf{TTP} = P_2$.

7. If $\mathsf{flag}_2 = 0$, and if $\mathcal{A}$ broadcasts ("accuse", P_1, P_2, c_1, c_2), then

– If $c_1 = c_2$, set $\mathsf{TTP} = P_1$.

– Else, if only c_1 is not the same as $\mathsf{H}(\delta_{12})$, where δ_{12} was sent by $\mathcal{S}^{P_0}$, broadcast ("accuse", P_0) on behalf of P_1, and set $\mathsf{TTP} = P_2$. Analogously for c_2.

8. If at any point during the simulation a TTP is identified, then send the output shares $[\![\mathsf{T}_o]\!]_0 = \left(\beta_{\mathsf{T}_o}, [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{01}, [\alpha_{\mathsf{T}_o}]^{\mathbf{B}}_{02}\right)$ to $\mathcal{A}$ on behalf of TTP.

Observe that in the real world, during the preprocessing phase, $\mathcal{A}$ receives messages that are computed as part of Shuffle-Pair and Set-Equality, where the messages are masked using some randomness to hide the missing permutation as well as information regarding the missing share at $\mathcal{A}$. During the online phase, $\mathcal{A}$ receives $\delta_{12}, \mathsf{H}(\delta_{12})$ where δ_{12} is a randomized using the random mask R_{12}. In this way, observe that the messages received by $\mathcal{A}$ in the real world are random and uniform. In the ideal world, too, observe that $\mathcal{A}$ receives messages that are sampled randomly from the uniform distribution. Moreover, $\mathcal{S}^{P_0}$ can verify the correctness of the messages sent by $\mathcal{A}$, as described earlier. This allows $\mathcal{S}^{P_0}$ to also simulate all the accuse messages as done in the real world. In this way, real-world and ideal-world executions are indistinguishable. □

3.8.2 Security of clustering protocols

Observe that our secure protocols for HKPR computation and clustering invoke the underlying 3PC protocols of SWIFT and the newly designed protocols provided in §3.4. Since the designed 3PC protocols invoke the protocols of SWIFT (as described in §2.3), whose security has been established therein, informally, the security of Π_{SGP} and $\Pi_{\mathsf{SClustering}}$ follows from the security of the underlying primitives. While the correctness and obliviousness of the designed protocols were given in place, an overview is provided next.

Correctness and data obliviosness In §3.5.2, the graph propagation algorithm is given using the Scatter and Gather primitives of GraphSC. Observe that in each of these primitives, every entry in the entire DAG list representation (including both nodes and edges) is accessed in each iteration. Further, the steps within these primitives can be made data-oblivious. Combined with the fact that Scatter and Gather are data-oblivious due to source and destination sort as provided by GraphSC, this ensures that the graph propagation algorithm in 3.5.2 is data-oblivious.

Regarding the algorithm in §3.6.2, the Scatter and Gather primitives are oblivious using the same arguments given as above. The only other function in §3.5.2 is FindCluster. FindCluster

makes use of the number of neighbours of a vertex v, $\mathsf{u} \in \mathsf{Nb_v}$, that have a greater value of $\frac{\boldsymbol{\rho}[\mathsf{u}]}{\mathsf{deg_u}}$ than that of vertex v's $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\mathsf{deg_v}}$. This value is called $\mathsf{G}[\mathsf{v}].\mathsf{GreaterCount}$, and is computed during the Scatter-Gather step. Then, the function makes a single sweep over the list representation of the graph. If a vertex satisfying the required constraints is found, it is added to a set S. The correctness of FindCluster relies on correctly computing the Cheeger ratio Φ_s of set S. For this, the volumes of sets S, $\mathsf{V} \setminus \mathsf{S}$, denoted as $\mathsf{vol_S}$ and $\mathsf{vol_{V \setminus S}}$, respectively, can be computed easily by adding or subtracting the degree of the vertex v being added to the set S. Next, ∂, which is the number of edges crossing the cluster S needs to be computed. A common global variable is used for computing ∂ by accumulating edges in it during a sweep over the DAG list that is sorted according to $\frac{\boldsymbol{\rho}[i]}{\mathsf{deg}_i}$ for all $\mathsf{G}[i]$ belonging to the DAG list.

The edges incident on a vertex v that is yet to be added to the set S can be categorized as either (i) having one endpoint within S and the other endpoint outside S, or (ii) having both endpoints outside S. The number of edges at v of type (i) are $\mathsf{G}[\mathsf{v}].\mathsf{GreaterCount}$, while those of type (ii) are $\mathsf{G}[\mathsf{v}].\mathsf{deg}$ - $\mathsf{G}[\mathsf{v}].\mathsf{GreaterCount}$. Thus, the number of edges that cross the new cluster $\mathsf{S} \cup \mathsf{v}$ is updated by subtracting the number of edges of type (i) and adding the number of edges of (ii). Updating ∂ in this way ensures that the Cheeger ratio is computed correctly. This ensures the correctness of FindCluster. Observe that during the computation of FindCluster, all the edges and vertices of the graph are swept through once. Further, since every update happens depending on only whether the current entry in the DAG list is a vertex or an edge, and not based on the structure of the graph, the sweep is oblivious. Finally, since the Scatter, Gather, and FindCluster is oblivious, this ensures that the entire algorithm is oblivious.

Security Ideal functionality for computing graph propagation metric appears in Fig. 3.22.

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

$\mathcal{F}_{\mathsf{SGP}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

- Receive as input the $[\![\cdot]\!]$-shares of G, x and $\{\mathsf{Y}_j, \mathsf{w}_j\}_{j=0}^{\mathsf{L}}, a, b$ from all parties.
- Reconstruct G, x using the shares of honest parties, and run algorithm 2.
- Generate $[\![\cdot]\!]$-shares of the updated G and send $(\mathsf{Output}, [\![\mathsf{G}]\!]_s^{\mathbf{B}})$ to $P_s \in \mathcal{P}$.

Figure 3.22: Ideal functionality for computing graph propagation metric.

Lemma 3.5 (Security) *Protocol Π_{SGP} (Fig. 3.10) securely realizes $\mathcal{F}_{\mathsf{SGP}}$ (Fig. 3.22) in computational 3PC setting against a malicious adversary $\mathcal{S}$ in $(\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Mul-Tr}}, \mathcal{F}_{\mathsf{Sel}})$-hybrid model.*

Proof: The simulator for Π_{SGP} appears in Fig. 3.23.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{SGP}}$ honestly executes the protocol steps and proceeds as follows.

- Set $[\![\mathsf{G}[i].\mathsf{r}]\!] = [\![\mathsf{x}[i]]\!]$ and $[\![\mathsf{G}[i].\rho]\!] = [\![0]\!]$ for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$, on behalf o the honest parties.
- Do the following for $j = 0$ to $\mathsf{L} - 1$.
  - Emulate $\mathcal{F}_{\mathsf{Shuffle}}$ on input $[\![\mathsf{G}]\!]$ and output the shuffled result to $\mathcal{A}$. Apply public permutation to source sort G and set $[\![\mathsf{val}]\!] = [\![0]\!]$.

  // Scatter(G)
  - for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do
    - Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{G}[i].\mathsf{r}]\!], [\![\frac{1}{(\mathsf{G}[i].\mathsf{deg})^b}]\!], \mathsf{f}$ and output $[\![\mathsf{v}]\!]$ to $\mathcal{A}$ where $\mathsf{v} = \mathsf{G}[i].\mathsf{r} \cdot \frac{1}{(\mathsf{G}[i].\mathsf{deg})^b}$ truncated by f bits. Compute $[\![\mathsf{val}']\!] = \left(\frac{\mathsf{Y}_{j+1}}{\mathsf{Y}_j}\right) \cdot [\![\mathsf{v}]\!]$ on behalf of the honest parties.
    - Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $[\![\mathsf{val}]\!], [\![\mathsf{val}']\!], [\![\mathsf{G}[i].\mathsf{isV}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{val}]\!]$ to $\mathcal{A}$. Set $[\![\mathsf{G}[i].\mathsf{dt}]\!] = [\![\mathsf{val}]\!]$ on behalf of the honest parties.
  - Emulate $\mathcal{F}_{\mathsf{Shuffle}}$ on input $[\![\mathsf{G}]\!]$ and output the shuffled result to $\mathcal{A}$. Apply public permutation to destination sort G.

  // Gather(G)
  - for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do
    - Compute $[\![\mathsf{G}[i].\rho]\!] = [\![\mathsf{G}[i].\rho]\!] + \left(\frac{\mathsf{w}_j}{\mathsf{Y}_j}\right) [\![\mathsf{G}[i].\mathsf{r}]\!]$ on behalf of the honest parties.
    - Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{agg}]\!], [\![\frac{1}{(\mathsf{G}[i].\mathsf{deg})^a}]\!], \mathsf{f}$ and output $[\![\mathsf{G}[i].\mathsf{r}]\!]$ to $\mathcal{A}$.
    - Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $[\![\mathsf{agg}]\!] + [\![\mathsf{G}[i].\mathsf{dt}]\!], [\![0]\!], [\![\mathsf{G.isV}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{agg}]\!]$ to $\mathcal{A}$.

Figure 3.23: Simulator for Π_{SGP}.

The simulator emulates $\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from the simulator. This is indistinguishable from its view in the real world, where it sees random values. □

The ideal functionality for securely computing greater count appears in Fig. 3.24.

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

$\mathcal{F}_{\mathsf{GreaterCount}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]$-shares of $\mathsf{G}, \boldsymbol{\rho}$ from all parties.

– Reconstruct $\mathsf{G}, \boldsymbol{\rho}$ using the shares of honest parties.

– For each node $\mathsf{v} \in \mathsf{G}$, compute the number of neighbors u of v that have a greater $\frac{\boldsymbol{\rho}[\mathsf{u}]}{\deg_{\mathsf{u}}}$ than $\frac{\boldsymbol{\rho}[\mathsf{v}]}{\deg_{\mathsf{v}}}$, and store it in $\mathsf{G}[\mathsf{v}].\mathsf{GreaterCount}$. Here, the output of multiplication is truncated using the probabilistic truncation method.

– Generate $[\![\cdot]\!]$-shares of the updated G and send $(\mathsf{Output}, [\![\mathsf{G}]\!]_s^{\mathbf{B}})$ to $P_s \in \mathcal{P}$.

Figure 3.24: Ideal functionality for computing greater count.

Lemma 3.6 (Security) $\Pi_{\mathsf{greaterCount}}$ *securely realizes* $\mathcal{F}_{\mathsf{GreaterCount}}$ *in computational 3PC setting against a malicious adversary* $\mathcal{S}$ *in* $(\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Comp}})$*-hybrid model.*

Proof: The simulator for $\Pi_{\mathsf{greaterCount}}$ appears in Fig. 3.25.

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

$\mathcal{S}_{\mathsf{greaterCount}}$ honestly executes the protocol steps and proceeds as follows.

– Emulate $\mathcal{F}_{\mathsf{Shuffle}}$ on input $[\![\mathsf{G}]\!]$ and output the shuffled result to $\mathcal{A}$. Apply public permutation to source sort G and set $[\![\mathsf{v}]\!] = [\![0]\!]$ on behalf of the honest parties.

– $\mathsf{Scatter}(\mathsf{G})$: Do the following for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$.

- Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{G}[i].\rho]\!], [\![\frac{1}{\mathsf{G}[i].\deg}]\!], \mathsf{f}$ and output $[\![\mathsf{G}[i].\rho]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $[\![\mathsf{v}]\!], [\![\mathsf{G}[i].\rho]\!], [\![\mathsf{G}[i].\mathsf{isV}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{v}]\!]$ to $\mathcal{A}$.
- Set $[\![\mathsf{G}[i].\mathsf{dt}]\!] = [\![\mathsf{val}]\!]$ on behalf of the honest parties.

– Emulate $\mathcal{F}_{\mathsf{Shuffle}}$ on input $[\![\mathsf{G}]\!]$ and output the shuffled result to $\mathcal{A}$. Apply public permutation to destination sort G and set $[\![\mathsf{agg}]\!] = [\![0]\!]$ on behalf of the honest parties.

– $\mathsf{Gather}(\mathsf{G})$: Do the following for $i = |\mathsf{V}| + |\mathsf{E}|$ to 1.

- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $([\![\mathsf{v}]\!], [\![\mathsf{G}[i].\rho]\!], [\![\mathsf{G.isV}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{val}]\!]$ to $\mathcal{A}$.
- Compute $[\![\mathsf{G}[i].\mathsf{dt}]\!] = [\![\mathsf{G}[i].\mathsf{dt}]\!] - [\![\mathsf{val}]\!]$ on behalf of the honest parties.

– Do the following for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$.

- Set $[\![\mathsf{G}[i].\mathsf{GreaterCount}]\!] = [\![\mathsf{agg}]\!]$ on behalf of the honest parties.
- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $[\![\mathsf{agg}]\!], 0, [\![\mathsf{G}[i].\mathsf{isV}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{agg}]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Comp}}$ on inputs $0, [\![\mathsf{G}[i].\mathsf{dt}]\!]$ and output $[\![c]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $[\![\mathsf{agg}]\!], [\![\mathsf{agg}]\!] + 1, [\![c]\!]^{\mathbf{B}}$ and output $[\![\mathsf{agg}]\!]$ to $\mathcal{A}$.

Figure 3.25: Simulator for $\Pi_{\mathsf{greaterCount}}$.

The simulator emulates $\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Comp}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from the simulator. This is indistinguishable from its view in the real world, where it sees random values. □

The ideal functionality for clustering appears in Fig. 3.26.

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

$\mathcal{F}_{\mathsf{Cluster}}$ interacts with the parties in $\mathcal{P}$ and the ideal world malicious adversary $\mathcal{S}$, and proceeds as follows.

– Receive as input the $[\![\cdot]\!]$-shares of $\mathsf{G}, \boldsymbol{\rho}$, and ϕ from all parties. Here, tuples in G have associated with them ρ, which is the HKPR metric computed by taking the target node into account.

– Reconstruct $\mathsf{G}, \boldsymbol{\rho}$ using the shares of honest parties.

– Run algorithm 4 to determine the set S of nodes that belong to a local cluster, and the flag which indicates the number of nodes in S. Reorder G such that nodes in S appear in the beginning of G.

– Generate $[\![\cdot]\!]$-shares of the updated G and send $(\mathsf{Output}, [\![\mathsf{flag}]\!]_s, [\![\mathsf{G}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 3.26: Ideal functionality for clustering.

Lemma 3.7 (Security) *Protocol $\Pi_{\mathsf{SClustering}}$ (Fig. 3.12) securely realizes $\mathcal{F}_{\mathsf{Cluster}}$ (Fig. 3.26) in the computational 3PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{GreaterCount}}, \mathcal{F}_{\mathsf{Sort}}, \mathcal{F}_{\mathsf{Div}}, \mathcal{F}_{\mathsf{4\text{-}Mul}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Comp}})$-hybrid model.*

Proof:

The simulator for $\Pi_{\mathsf{SClustering}}$ appears in Fig. 3.27.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{SCluster}}$ honestly executes the protocol steps and proceeds as follows.

– Emulate $\mathcal{F}_{\mathsf{GreaterCount}}$ on inputs $[\![\mathsf{G}]\!], [\![\boldsymbol{\rho}]\!]$ and output the updated $[\![\mathsf{G}]\!]$ to $\mathcal{A}$ that includes shares of the greater count values.

– Emulate $\mathcal{F}_{\mathsf{Sort}}$ to sort $[\![\mathsf{G}]\!]$ with $\mathsf{G}.\boldsymbol{\rho}$ as the key and output the sorted $[\![\mathsf{G}]\!]$ to $\mathcal{A}$.

– Initialize $\partial = [\![0]\!], [\![\mathsf{vol_S}]\!] = [\![0]\!], [\![\mathsf{vol_{V\setminus S}}]\!] = \sum_{\mathsf{v}\in\mathsf{V}} [\![\mathsf{G[v].deg}]\!]$, $[\![\phi_{\mathsf{min}}]\!] = [\![2^{32}]\!]$ and $[\![\mathsf{flag}]\!] = [\![0]\!]$ on behalf of the honest parties.

– Do the following for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$.

- Compute $[\![\partial]\!] = [\![\partial]\!] + [\![\mathsf{G}[i].\mathsf{deg}]\!]$ - $2\cdot[\![\mathsf{G}[i].\mathsf{GreaterCount}]\!]$ and $[\![\mathsf{vol_S}]\!] = [\![\mathsf{vol_S}]\!] + [\![\mathsf{G}[i].\mathsf{deg}]\!]$ and $[\![\mathsf{vol_{G\setminus S}}]\!] = [\![\mathsf{vol_{G\setminus S}}]\!]$ - $[\![\mathsf{G}[i].\mathsf{deg}]\!]$ on behalf of the honest parties.
- Emulate $\mathcal{F}_{\mathsf{Comp}}$ on inputs $[\![\mathsf{vol_{G\setminus S}}]\!], [\![\mathsf{vol_S}]\!]$ and output $[\![\mathrm{min}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $[\![\mathsf{vol_S}]\!], [\![\mathsf{vol_{G\setminus S}}]\!], [\![\mathrm{min}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{vol}_{\mathrm{min}}]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Div}}$ on inputs $[\![\partial]\!], [\![\mathsf{vol}_{\mathrm{min}}]\!]$ and output $[\![\Phi_s]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Comp}}$ on $(2[\![\varsigma]\!], [\![\mathsf{vol_S}]\!])$ and $([\![\mathsf{vol_S}]\!], [\![\varsigma]\!]/2)$ and $([\![\sqrt{8\phi}]\!], [\![\Phi_s]\!])$ and $([\![\Phi_s]\!], [\![\phi_{\mathsf{min}}]\!])$ to output $[\![\bar{\theta_1}]\!]^{\mathbf{B}}$, $[\![\bar{\theta_2}]\!]^{\mathbf{B}}$, $[\![\bar{\theta_3}]\!]^{\mathbf{B}}$, $[\![\mathrm{min}]\!]^{\mathbf{B}}$, respectively, to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{3\text{-}Mul}}$ on inputs $[\![\theta_1]\!]^{\mathbf{B}}, [\![\theta_2]\!]^{\mathbf{B}}, [\![\theta_3]\!]^{\mathbf{B}}, [\![\mathrm{min}]\!]^{\mathbf{B}}$ and output $[\![\theta]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $([\![\phi_{\mathsf{min}}]\!], [\![\Phi_s]\!], [\![\theta]\!]^{\mathbf{B}})$ and $([\![\mathsf{flag}]\!], [\![i]\!], [\![\theta]\!]^{\mathbf{B}})$ to output $[\![\phi_{\mathsf{min}}]\!]$, $[\![\mathsf{flag}]\!]$, respectively, to $\mathcal{A}$.

Figure 3.27: Simulator for $\Pi_{\mathsf{SClustering}}$.

The simulator emulates $\mathcal{F}_{\mathsf{GreaterCount}}, \mathcal{F}_{\mathsf{Sort}}, \mathcal{F}_{\mathsf{Div}}, \mathcal{F}_{\mathsf{4\text{-}Mul}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Comp}}$ in the order in which they appear in the protocol. The simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from the simulator. This is indistinguishable from its view in the real world, where it sees random values. □

Chapter 4

Secure Graph Neural Networks

This chapter discusses the secure framework, Entrada, that allows to efficiently realize graph convolutional networks (GCNs).

4.1 Overview

We design Entrada, a secure framework for efficiently evaluating GCNs. To the best of our knowledge, this is done for the first time. We build Entrada over the 4-party computation (4PC) framework of Tetrad [138] since Tetrad is a robust framework and is known to outperform other frameworks in small-party (honest-majority) setting. Note, however, that Entrada is a more versatile framework than Tetrad. Entrada provides the following key features.

– Apart from secure inference, Entrada additionally offers secure training of GCN.

– While traditional (cleartext) GCNs [133] consider operating on a graph which comprises a single type of edge, recent works [158] also design GCNs for graphs which account for multiple types of edges (heterogeneous GCNs). Thus, we design Entrada to enable operating on both these types of graphs.

To highlight the performance of Entrada, we carry out extensive experiments. Entrada provides an improved GCN accuracy of 79.3% in comparison to 74.1% of Tetrad, while cleartext computations provide an accuracy of 79.9%. We note that the reduced accuracy of Tetrad is due to its reliance on approximate variants for non-linear functions (e.g., Softmax), as opposed to Entrada, which uses the accurate versions for the same. With respect to efficiency, Entrada witnesses gains of up to 4× in online run time, and three orders of magnitude in preprocessing time over Tetrad for GCN training. We also showcase the practicality of our solution by benchmarking fraud detection algorithms of [223], [158].

These improvements witnessed by Entrada are a result of two-fold contributions. On the first hand, we enhance Tetrad by adding new primitives and improving existing primitives, which makes it a more accurate, comprehensive and efficient framework for privacy-preserving machine learning (PPML). On the other hand, specific to GCN, we leverage GraphSC framework [179, 13] and bring in new contributions therein, including a secure shuffle protocol. We elaborate on these below.

Enhancing Tetrad

Entrada enhances Tetrad by providing support for efficient realizations of prefix OR, double bit-injection, exponentiation, division, and inverse square root. While most of these are well-studied [43, 126, 135, 138], our choice of Tetrad and optimizations thereof aid in obtaining efficient realizations for the same.

Tailoring GraphSC for GCN

To further enhance efficiency, we leverage the GraphSC paradigm [179, 13]. Our contribution entails:

– *cleartext → message-passing algorithm:* Identifying the relevant cleartext computations in GCNs that can benefit from GraphSC and can be rendered as message-passing algorithms.

– *message-passing → secure protocol:* Redefining the graph algorithm in terms of Scatter-Gather primitives as well as defining the GCN specific computations that are required to be performed within these Scatter-Gather primitives. The Scatter-Gather primitives are then securely realized via Entrada.

– *Secure shuffle for* Entrada*:* Designing a secure *shuffle* protocol in the 4-party setting, as required for GraphSC, which has an amortized communication of 3N ring elements (where N denotes the size of the vector to be shuffled) and 1 round of interaction in the online phase. Note that the secure shuffle protocol forms an integral part of various applications such as anonymous broadcast [80], secure sorting [13], etc. Hence, the inclusion of the shuffle protocol makes Entrada a versatile framework and opens up avenues for exploring its use in other shuffle-based application scenarios.

Concretely, while 'Enhancing Tetrad' brings in an improvement of 1.7× in the online run time of GCN training over Tetrad (and 486× in preprocessing run time), additionally 'Tailoring GraphSC' further enhances the efficiency up to 4× (5782× for preprocessing).

Input sharing for graph-structured data

To enable multiple clients to efficiently secret-share their input to the servers, we design a secure protocol for input sharing. The input comprises the graph represented as an adjacency matrix and data (features) associated with the vertices of the graph. Recall from §1.1.1 that a client's input only comprises a partial view of the entire graph (i.e., a subset of vertices together with the corresponding data, and their associated edges). Hence, ensuring that the input generated at the servers indeed corresponds to an adjacency matrix, and the associated data adheres to the structure as required for a GCN, is challenging. Designed independent of the graph algorithm, our input-sharing protocol to generate secret shares of the adjacency matrix and the associated data may find use in other graph-based applications too.

Note that GraphSC operates on a list representation of the graph. Hence, we additionally describe the method to translate the adjacency matrix-based representation of the graph, generated as part of the input-sharing phase in secret shares, into its list representation. Our shuffle protocol also finds use in performing this translation. Note that the adjacency matrix has $|\mathsf{V}|^2$ entries that account for every possible edge, while the list representation only stores information regarding edges that are actually present. Hence, generating the $(|\mathsf{V}| + |\mathsf{E}|)$-sized list representation of the graph from the $|\mathsf{V}|^2$-sized matrix representation while hiding the graph topology is challenging.

4.2 Related work

We discuss the related work for GCNs in cleartext first, followed by MPC works that consider designing secure solutions for evaluating neural networks, in general, since secure GCNs via MPC have not been well explored. Following this, we discuss the different works that have studied the primitives that are used in GCNs, such as division, exponentiation, inverse square root, bit injection, conversions between arithmetic and Boolean sharing, dot products, etc. Here, note that the primitives of division, exponentiation and inverse square root have been newly included in the framework of Tetrad [138].

GCNs: The increasing popularity of convolutional neural networks led to the study of performing convolutions on graph-structured data. This resulted in the design of graph convolutional networks (GCNs). There are several works which study GCNs in the cleartext [133, 70, 100, 219, 229, 227]. These can broadly be classified as spatial-based GCNs [133, 70], spectral-based GCNs [35, 105], attention-based GCNs [219] and recurrent-based GCNs [150, 232], to name a few. While this categorization is not mutually exclusive, some GCNs combine

different types of layers. Among these, spatial-based GCNs have proven to be advantageous for graph-structured data because they are computationally more efficient and can incorporate local neighbourhood information. Hence, they are more effective for applications such as fraud detection and allow capturing dependencies between nodes of the graph, thereby aiding in identifying anomalous patterns that are indicative of fraudulent behaviour. Among these, the work of [133] was the first to propose GCNs for semi-supervised learning on graph-structured data and serves as the benchmark in the domain of GCNs. Hence, we focus on designing privacy-preserving solutions for evaluating the GCN described in the work of [133].

Most works in the privacy-preserving literature look at securely computing neural networks (NN) via MPC [173, 136, 193, 138, 50, 174, 194, 49, 220, 143, 199]. Despite the interest in securely evaluating neural networks, none consider operating on graph-structured data. Hence, GCNs have not been well explored. Prior work that explores GCNs only considers performing secure inference [208]. Moreover, it considers providing inference only for the relatively older GCN model of [70]. One may consider extending the works that provide secure solutions for neural networks to securely realize GCNs. However, these works either lack the necessary primitives required for evaluating GCNs or provide weaker security guarantees, or fare poorly in terms of accuracy and efficiency of evaluating GCNs. For instance, several works trade off accuracy for efficiency by relying on MPC-friendly alternatives for non-linear functions such as sigmoid and softmax[138, 50, 173]. Instead, we strive to design *accurate* protocols for GCN evaluation while *not* compromising on *efficiency*.

MPC primitives: As discussed in the previous chapter, secure division has been studied in various works such as [45, 43, 126, 6, 220, 50, 138, 174]. When working over the 4PC setting, we adapt the protocol of [43] that is based on Goldschmidt's division since it renders more efficient protocols. Performing secure division via Goldschmidt's algorithm additionally relies on prefixOR computation which has also been explored in several works [44, 43, 132]. Further, secure truncation is a special case of secure division where the divisor is publicly known. There are two approaches to performing truncation that have been explored in the literature—probabilistic and deterministic. In probabilistic truncation [174, 173, 61, 136], the output of the truncation protocol may be different from the true output by 1 bit with some probability. [174] proposed a non-interactive probabilistic truncation protocol in the 2-party setting which was extended to the 3-party setting (and cal also be extended to n parties) in the work of [173], albeit while requiring interaction among the parties. On the other hand, deterministic truncation does not have the issue of 1-bit error. The 2-party works of [110, 199] provide solutions for deterministic truncation, albeit while incurring a high communication overhead in comparison to probabilistic truncation. We note that we rely on the probabilistic

truncation approach, as done in [138], since the 1-bit error does not incur significant accuracy loss in the ML model while being more efficient than deterministic approaches.

Next, secure exponentiation has been studied in various works [6, 63, 126, 123]. The work of [123] designs a customised exponentiation protocol in the 2-party setting with the end application of secure Poisson regression. The protocol leverages the knowledge of a lower bound on the exponent and is designed to work only for positive exponents. As opposed to this, [6, 126] design generic exponentiation protocols that work for negative exponents as well by relying on the bit decomposition of the exponent. In this regard, the protocol of [6] implicitly relies on a division protocol to account for negative exponents. A reliance on division is avoided in the work of [126] which results in an improved efficiency. We further improve the efficiency of this protocol by leveraging the 4-party setting.

Several works have studied the problem of securely computing inverse square root [4, 118, 130, 152, 6, 126, 132, 163] where they approach it by either expressing the output as a Taylor series, or via some other kind of numerical approximation (e.g. via Goldschmidt or Newton-Raphson approximation). [6] proposed computing square root using Goldschmidt and Raphson-Newton iterations. [163] proposed a more direct computation that avoids running two successive iterations, which was further optimized in the work of [126]. We adapt and optimize the protocol of [126] when considering the 4-party setting.

A bit injection allows one to multiply a Boolean-shared bit with an arithmetic-shared value. This primitive has been studied in several works [173, 50, 138]. However, in some scenarios, there is a need to perform multiplication between two Boolean-shared bits and an arithmetic-shared value. Hence, we design double bit injection protocol to cater to such scenarios.

With respect to the other primitives, we note that bit decomposition (arithmetic to Boolean conversion) protocol has been studied in various works [43, 45, 193, 50, 138, 136, 173]. While [173] provided a Boolean circuit-based approach to obtain the bit decomposition of a secret shared value, the circuit used for decomposition can be depth-optimized by relying on multi-input multiplication protocol of [194]. Bit to arithmetic conversion has also been looked at in several works [173, 50, 138]. Finally, dot product and matrix multiplications are fundamental building blocks in PPML where matrix multiplication can be reduced to dot products. While the communication complexity of dot product in the 3PC of [173] was dependent on the vector size, the online communication was made independent of the vector size in the 3PC of [193] while the preprocessing phase continued to have a communication complexity dependent on the vector size. The 3PC of [136] showed how to make the communication cost of dot product independent of vector size in online as well as the preprocessing phase. This independence of communication cost from the vector size has also been achieved in 4PC [50, 138, 136].

4.3 Preliminaries

4.3.1 System model

We consider secure outsourced computation in the 4-party setting of Tetrad [138], where four hired servers enact the role of parties $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$ (or equivalently $\mathcal{P} = \{P_i, P_j, P_k, P_l\}$). We let $\mathcal{A}$ denote a static malicious probabilistic polynomial time adversary which corrupts at most one party in $\mathcal{P}$. Each client (possibly malicious) secret-shares its input among servers, which evaluate the MPC protocol to obtain the secret-shared output. The output is then reconstructed towards the client. Similar to Tetrad, our constructions are secure in the real-world/ideal-world simulation paradigm. Further, Entrada, similar to Tetrad, enables achieving two different levels of security: (i) fairness—depending on the adversary's misbehaviour, either all parties obtain the output of the computation, or none do, and (ii) robustness—regardless of adversary's misbehaviour, all parties are guaranteed to obtain the output of the computation. In the remainder, we focus on describing the robust protocols. However, their fair variants can be attained easily, following Tetrad. We refer the reader to §2.4 for an overview of Tetrad, including the functionality $\mathcal{F}_{\mathsf{Setup}}$, sharing semantics, and the description of protocols from Tetrad that this chapter relies on.

4.3.2 GraphSC paradigm

The designed protocols leverage the GraphSC paradigm [179, 13] for efficiency reasons. Hence, we refer readers to §2.6 to familiarize themselves with the necessary background.

4.3.3 Graph convolutional networks (GCN)

The celebrated result of Kipf and Welling [133] showcases how convolutions can be generalized to graphs. Let $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ denote a graph, where V is the set of n nodes, and E is the set of edges. Let $\mathbf{A}$ denote the adjacency matrix of G of dimension $n \times n$, and $\mathbf{X}$ denote the matrix of node features of dimension $n \times f$, where f denotes the number of features. The authors in [133] propose a simple GCN model which allows learning succinct representation of nodes in the graph. In the case of node classification via GCN, the task is to assign labels to the unlabelled nodes in G by learning from those nodes that are labelled. To achieve this, the GCN model is trained using the labelled nodes provided as input, which makes up the training phase. This entails generating labels for the (already labelled) nodes via the computations in the *forward pass*, and updating the model parameters by accounting for the difference in the

generated labels and the true labels of the nodes via the computations in the *backward pass*. The trained model can then be used to perform inference where node classification is performed for the unlabelled nodes. Each of these is described in detail next.

4.3.3.1 Forward pass

In the forward pass, node representations $\mathbf{H}^{(i)}$, for the i^{th} layer are computed using the following equation[1].

$$\mathbf{H}^{(i)} = g^{(i)}\left(\hat{\mathbf{A}} \cdot \mathbf{H}^{(i-1)}\mathbf{W}^{(i-1)}\right) \tag{4.1}$$

Here $g^{(i)}$ denotes the activation function for layer i, and $\mathbf{H}^{(0)} = \mathbf{X}$, $\mathbf{W}^{(i)}$ is the weight matrix specific to layer i and $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$. Further, $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with self-loops, and $\tilde{\mathbf{D}}$ is the degree matrix for $\tilde{\mathbf{A}}$ (i.e., $\tilde{\mathbf{D}}$ stores the number of incident edges in $\tilde{\mathbf{A}}$ as the diagonal entry for each node). Matrices $\hat{\mathbf{A}}, \tilde{\mathbf{A}}, \tilde{\mathbf{D}}$ are all of dimension $n \times n$. Specifically, the work of [133] considers a two-layer instantiation of Equation (4.1), where the GCN model $\mathbf{Z}$ is given as a function of $\mathbf{X}, \mathbf{A}$ and is parameterised by the weight matrices $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ as given in Equation (4.2).

$$\mathbf{Z} = \overbrace{\mathsf{Softmax}(\hat{\mathbf{A}} \cdot \underbrace{\mathsf{ReLU}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})}_{\mathbf{H}^{(1)}}\mathbf{W}^{(1)})}^{\mathbf{H}^{(2)}} \tag{4.2}$$

Here, $\mathbf{W}^{(0)}$ has a dimension of $f \times h$, where h denotes the number of feature maps, whereas $\mathbf{W}^{(1)}$ is of dimension $h \times c$, where c is the number of labels to which the nodes of the graph will be mapped.

4.3.3.2 Backward pass

Since the weight matrices are trainable parameters, these are updated in the backward pass by accounting for the error in the computed output representation $\mathbf{Z}$ and the true representation $\mathbf{Y}$, with respect to the labelled nodes in G. This is captured using the cross-entropy error, denoted as $\mathcal{L}$, as given in Equation (4.3). Here, both $\mathbf{Z}, \mathbf{Y}$ are of dimension $n \times c$, and L denotes the index set of labelled nodes in G and hence is a subset of $\{1, \ldots, n\}$.

[1]GCNs have a multilayered architecture. Informally $\mathbf{H}^{(i)}$ in the intermediate layers captures the feature maps (properties) of the nodes, and its dimensions may vary across the layers. For the final layer, it captures the likelihood of a node being mapped to a particular label and hence has the dimension of $n \times c$ where c denotes the number of class labels.

$$\mathcal{L} = -\sum_{i \in L} \sum_{j=1}^{c} \mathbf{Y}_{ij} \ln \mathbf{Z}_{ij} \tag{4.3}$$

Given the loss function $\mathcal{L}$, derivative of $\mathcal{L}$ with respect to weight matrix $\mathbf{W}^{(0)}$, and derivative of $\mathcal{L}$ with respect to $\mathbf{W}^{(1)}$ is computed. This is then used to update weight matrices via stochastic gradient descent optimizer [133]. This completes the backward pass.

4.3.3.3 Traning and inference

Performing the computations in the forward pass, followed by the backward pass, constitutes one epoch of the training phase. Computations over several such epochs minimize $\mathcal{L}$, thereby yielding the trained weight matrices. Having generated the trained model, performing GCN inference for node classification is now possible by computing the forward pass, as described in Equation (4.2), using the obtained weight matrices[2].

4.3.3.4 Heteregeneous GCN

While the GCN of [133] deals with a single type of edge, the work of [158] designs heterogeneous GCNs. The main difference lies in the fact that the underlying graph now consists of different types of edges, $\mathcal{D}$. Specifically, $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ can now be seen as a collection of $|\mathcal{D}|$ subgraphs $\{\mathsf{G}_d = (\mathsf{V}, \mathsf{E}_d)\}$, where each subgraph comprises all the vertices of G, but edges of only type $d \in \mathcal{D}$. The heterogeneous graph representation $\{\mathsf{G}_d\}$ leads to $|\mathcal{D}|$ adjacency matrices $\{\mathbf{A}_d\}$, each of dimension $n \times n$. Thus, the heterogeneous GCN computation [158] is given as $\mathbf{H}^{(0)} = \mathbf{0}$ and for $i = 1, \ldots, T$:

$$\mathbf{H}^{(i)} = g^{(i)} \left(\mathbf{X} \cdot \mathbf{W}^{(i-1)} + \frac{1}{|\mathcal{D}|} \sum_{d=1}^{|\mathcal{D}|} \mathbf{A}_d \cdot \mathbf{H}^{(i-1)} \cdot \mathbf{W}_d^{(i-1)} \right) \tag{4.4}$$

Here, $\mathbf{0}$ denotes the matrix of all 0s of dimension $n \times c$ where c is the number of labels, T denotes the number of layers, and $g^{(i)}$ denotes the non-linear activation function for layer i. $\mathbf{W}^{(i)}, \mathbf{W}_d^{(i)}$ are the i^{th} layer weight matrices of dimension $c \times c$, where the latter additionally depends on the edge type d. To draw an analogy to the GCN of Kipf and Welling [133], note that the final output $\mathbf{Z}$ computed in [133] is equivalent to $\mathbf{H}^{(T)}$ which constitutes the final output for heterogeneous GCNs. Further, as can be seen from Equation (4.4), heterogeneous GCNs additionally require summing the values over each of the subgraphs to account for the

[2]Since Softmax is used to normalize the final result between $[0, 1]$, note that inference can be performed without it.

heterogeneity in the edges. The cross-entropy error for the backward pass is computed similarly to the GCN of [133], as given in Equation (4.3). In our work, we take $T = 2$ since this suffices for obtaining the desired accuracy. We let the activation function $g^{(1)}, g^{(2)}$ be ReLU for the first layer and $\mathsf{Softmax}$ for the second layer, respectively.

4.3.4 Notations

The notations pertaining to GCNs are summarized in Table 4.1. Further, notations with respect to Tetrad and the graphSC paradigm appear in §2.4 and §2.6, respectively.

Notation	Description
$\mathsf{G} = (\mathsf{V}, \mathsf{E})$	Graph G with set of n vertices V and set of edges E
$\mathbf{A}$	Adjacency matrix of G of dimension $n \times n$
$\tilde{\mathbf{A}}$	Adjacency matrix of G with self loops, i.e., $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$
$\tilde{\mathbf{D}}$	Degree matrix for $\tilde{\mathbf{A}}$
$\hat{\mathbf{A}}$	Normalized adjacency matrix $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$
$\mathbf{X}$	Matrix of node features of dimension $n \times f$ where f denotes the number of features
$\mathbf{H}^{(i)}$	Node representation matrix of i^{th} layer during GCN evaluation
$\mathbf{W}^{(i)}$	Weight matrix of i^{th} layer during GCN evaluation
$\mathbf{Z}$	GCN model $\mathbf{Z} = \mathsf{Softmax}(\hat{\mathbf{A}} \cdot \mathsf{ReLU}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})\mathbf{W}^{(1)})$ (computed output)
L	Index set of labelled nodes in G
$\mathbf{Y}$	Actual output with respect to the labelled nodes in G
$\mathcal{L}$	Cross-entropy error
$\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(i)}}$	Derivative of loss with respect to the weights in the i^{th} layer
c	Number of output classes
$\mathcal{D}$	Types of edges in a graph G
$\mathsf{G}_d = (\mathsf{V}, \mathsf{E}_d)$	Subgraph of G with edges of type $d \in \mathcal{D}$
$\mathbf{A}_d$	Adjacency matrix of G_d
$\mathbf{v}[i]$	i^{th} element of vector $\mathbf{v}$
$\odot$	Element-wise multiplication operator
dReLU	Derivative of ReLU

Table 4.1: Notations pertaining to GCNs.

4.4 Secure GCN

Recall that while securely evaluating GCNs, the inputs comprise the matrices $\tilde{\mathbf{D}}$, $\tilde{\mathbf{A}}$, $\hat{\mathbf{A}}$, $\mathbf{X}$ and $\mathbf{Y}$, in secret shares. We begin by describing the input-sharing phase, which comprises the steps

to obtain these matrices (in secret shares) from the clients. Following this, we discuss the steps to securely evaluate GCN. Since output reconstruction follows from Tetrad, we do not highlight the same.

4.4.1 Input sharing

This phase involves generating $[\![\cdot]\!]$-shares of $\mathbf{A}, \mathbf{X}, \mathbf{Y}$. Given these matrices, the other inputs required for GCN evaluation, i.e., $\tilde{\mathbf{D}}$, $\tilde{\mathbf{A}}$, $\hat{\mathbf{A}}$, can be generated without the involvement of the client. Recall that in a distributed setting, the client may possess only a partial view of the input, i.e., information regarding some k nodes in the graph which correspond to k rows of $\mathbf{A}, \mathbf{X}$ and $\mathbf{Y}$. Thus, each client secret-shares entries corresponding to the rows they possess towards the servers. Having received the entries of all rows from the clients, the servers generate the complete matrices by stacking up the rows (assuming the mapping between clients and rows of the matrices is known to servers). We give a high-level overview of the challenges in achieving this and their resolutions next.

Generating $[\![\cdot]\!]$-shares of $\mathbf{X}$ To ensure that a possibly malicious client, C, has not cheated while secret-sharing the rows of $\mathbf{X}$, it suffices to ensure that C consistently secret-shares each element in the respective rows of $\mathbf{X}$ that it possesses. This can be performed similarly to as done in [136], albeit more efficiently without relying on commitments. Elaborately, say $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ is an input element to be shared by C. Servers non-interactively generate $[\cdot]$-shares of the mask $\alpha_{\mathsf{x}} \in \mathbb{Z}_{2^\ell}$ using $\mathcal{F}_{\mathsf{Setup}}$ (Fig. 2.5). To obtain $\beta_{\mathsf{x}} = \mathsf{x} + \alpha_{\mathsf{x}}$ from the client, $\alpha_{\mathsf{x}} = \alpha_{\mathsf{x}1} + \alpha_{\mathsf{x}2} + \alpha_{\mathsf{x}3}$ is sent to C as follows. Since each $\alpha_{\mathsf{x}i}$ for $i \in \{1, 2, 3\}$ is held by three servers, two of them send $\alpha_{\mathsf{x}i}$ to C, while the third sends $\mathsf{H}(\alpha_{\mathsf{x}i})$. Note that all servers are required to communicate $\alpha_{\mathsf{x}i}$ to C to ensure that a corrupt server's attempt of cheating by sending an incorrect value is subverted. Moreover, since multiple elements are required to be shared by C, the use of hash allows computing and sending a single hash value on the concatenation of all these elements, thereby reducing the communication complexity. Further, note that among the three versions of the received $\alpha_{\mathsf{x}i}$, since at most one can be incorrect (owing to the presence of at most one corrupt server), taking the value which appears in majority enables C to obtain the correct value for each $\alpha_{\mathsf{x}i}$. C then computes $\alpha_{\mathsf{x}} = \alpha_{\mathsf{x}1} + \alpha_{\mathsf{x}2} + \alpha_{\mathsf{x}3}$, $\beta_{\mathsf{x}} = \mathsf{x} + \alpha_{\mathsf{x}}$, and sends β_{x} to servers P_1, P_2, P_3. Finally, to ensure that C has sent the consistent β_{x} to P_1, P_2, P_3, they exchange the value received from C among themselves. Since at most one among P_1, P_2, P_3 can be corrupt, there will exist a majority in the exchanged values, which is taken as the final value for β_{x}.

Generating $[\![\cdot]\!]$-shares of A In addition to performing the consistency checks as described above to ensure consistent sharing, servers are also required to ensure that the client's inputs correspond to valid rows of an adjacency matrix $\mathbf{A}$, i.e. $\mathbf{A}^\intercal = \mathbf{A}$. For this, they non-interactively generate $[\![\cdot]\!]$-shares of a random symmetric matrix $\mathbf{R}$, compute $[\![\mathbf{S}]\!] = [\![\mathbf{A}]\!] + [\![\mathbf{R}]\!]$, and open the resultant matrix $\mathbf{S}$. Servers can then locally verify if $\mathbf{A}$ is symmetric by checking if $\mathbf{S}_{ij} = \mathbf{S}_{ji}$ for $i, j \in \{1, 2, \ldots, n\}$. If so, they proceed to verify whether the elements of $\mathbf{A}$ as generated by the clients are either a 0 or a 1. For this, servers use the fact that $\mathsf{z}^2 - \mathsf{z} = 0$ only if $\mathsf{z} \in \{0, 1\}$. In order to verify this equation with respect to all the elements shared by a client, servers compute a random linear combination with respect to each element z_i shared by the client and verify if the combination yields a 0. However, this check still allows a client to cheat with probability 1/2 when working over the ring algebraic structure [2, 30]. Thus, to reduce the cheating probability, the check is repeated κ times, which bounds the cheating probability by $1/2^\kappa$. If any of the checks fail, depending on the application scenario, one of the following can be done: (i) entries in i^{th} row and j^{th} column can be set to default, (ii) entries pertaining i^{th} and j^{th} nodes can be deleted from the graph (the same should be reflected in the other inputs, $\mathbf{X}, \mathbf{Y}$, as well), (iii) the computation is halted.

Generating $[\![\cdot]\!]$-shares of Y Recall that each row of $\mathbf{Y}$ has at most one position set as 1 (and all others as 0s). Thus, after verifying the consistency of the received shares, to verify if the i^{th} row $\{\mathbf{Y}_{i1}, \mathbf{Y}_{i2}, \ldots, \mathbf{Y}_{ic}\}$ satisfies this condition, we use the idea of [161]. The approach is to check if $\left(\sum_{j=1}^{c} \mathbf{Y}_{ij} \cdot \mathsf{r}_j\right)^2 - \left(\sum_{j=1}^{c} \mathbf{Y}_{ij} \cdot \mathsf{r}_j^2\right) = 0$. Here, $\mathsf{r}_j \in \mathbb{Z}_{2^\ell}$ are random public values, and the check passes with high probability over a field if at most one $\mathbf{Y}_{ij}$ is a 1. However, similar to the case of $\mathbf{A}$, this check also has a failure probability of 1/2 over rings. Hence, we repeat this check κ times to bound the failure probability by $1/2^\kappa$.

Generating $[\![\cdot]\!]$-shares of $\tilde{\mathbf{D}}$, $\tilde{\mathbf{A}}$ and $\hat{\mathbf{A}}$ Having generated $[\![\mathbf{A}]\!]$, generation of $[\![\tilde{\mathbf{A}}]\!] = [\![\mathbf{A}]\!] + [\![\mathbf{I}]\!]$ can happen non-interactively. The $[\![\cdot]\!]$-shares of diagonal entries of $\tilde{\mathbf{D}}$ can also be generated non-interactively by summing the entries in the corresponding rows of $[\![\tilde{\mathbf{A}}]\!]$. Finally, computing $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$, involves computing the element-wise inverse square root of the diagonal element of $\tilde{\mathbf{D}}$ via the secure inverse square root protocol.

4.4.2 Secure evaluation of GCN

The steps involved in the secure evaluation of GCN are summarised in Fig. 4.1.

Algorithm GCN evaluation

Training Phase

Input: $\hat{\mathbf{A}}$ (normalized adjacency matrix of dimension $n \times n$), $\mathbf{X}$ (feature matrix of dimension $n \times f$), $\mathbf{Y}$ (matrix of true output for labelled nodes of dimension $n \times c$), *#epochs* (number of epochs).

Output: $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ (trained weight matrices for layer 0,1, respectively).

//Initialization

- Randomly sample $\mathbf{W}^{(0)} \in \mathbb{R}^{f \times h}$, and $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times c}$, where $\mathbb{R}$ denotes the set of real numbers.
- $\beta_1 = 0.9, \beta_2 = 0.999, \mathbf{M}^{(0)} = \mathbf{M}^{(1)} = \mathbf{0}, \mathbf{V}^{(0)} = \mathbf{V}^{(1)} = \mathbf{0}$, $\epsilon = 10^{-8}$

For t in range(*#epochs*) do:

//Forward pass

- $\mathbf{H}^{(0)} = \mathsf{ReLU}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})$, $\mathbf{In} = \hat{\mathbf{A}}\mathbf{H}^{(0)}\mathbf{W}^{(1)}$, $\mathbf{Z} = \mathsf{Softmax}(\mathbf{In})$

//Backward pass

- $\left(\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(1)}}\right) = (\mathbf{H}^{(1)})^{\mathsf{T}}(\mathbf{Z} - \mathbf{Y})$
- $\left(\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(1)}}\right) = (\hat{\mathbf{A}}\mathbf{X})^{\mathsf{T}} \left(\mathsf{dReLU}(\mathbf{In}) \odot \hat{\mathbf{A}}(\mathbf{Z} - \mathbf{Y})(\mathbf{W}^{(1)})^{\mathsf{T}}\right)$

//Weights update via Adam optimizer

- $\mathbf{M}^{(i)} = \beta_1 \mathbf{M}^{(i)} + (1 - \beta_1)\left(\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(i)}}\right)$ for layer $i \in \{0, 1\}$
- $\mathbf{V}^{(i)} = \beta_2 \mathbf{V}^{(i)} + (1 - \beta_2)\left(\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(i)}}\right)^2$ for layer $i \in \{0, 1\}$
- $\hat{\mathbf{M}}^{(i)} = \frac{\mathbf{M}^{(i)}}{1-(\beta_1)^t}$ and $\hat{\mathbf{V}}^{(i)} = \frac{\mathbf{V}}{1-(\beta_2)^t}$, for layer $i \in \{0, 1\}$
- Update $\mathbf{W}^{(i)} = \mathbf{W}^{(i)} - \hat{\mathbf{M}}^{(i)}\left(\frac{\alpha}{\sqrt{\hat{\mathbf{V}}^{(i)}} + \epsilon}\right)$, where α is learning rate.

Inference Phase

Input: $\hat{\mathbf{A}}$ (normalized adjacency matrix of dimension $n \times n$), $\mathbf{X}$ (feature matrix of dimension $n \times f$), $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ (trained weight matrices of dimensions $h \times f$ and $f \times c$, respectively).

Output: $\mathbf{Z}$ (matrix of dimension $n \times c$ that captures the likelihood of unlabelled nodes belonging to each class/label).

- $\mathbf{Z} = \hat{\mathbf{A}} \cdot \mathsf{ReLU}(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})\mathbf{W}^{(1)}$

Figure 4.1: Steps involved in training and inference phase of GCN.

This secure evaluation of GCN entails servers securely computing secret shares of $\mathbf{Z}$ via the forward pass followed by computing the derivative of $\mathcal{L}$ with respect to the weight matrices, as defined in the backward pass and updating the weight matrices. Assuming the inputs are available in secret shares, servers begin by initializing random weight matrices $\mathbf{W}^{(0)}$ and $\mathbf{W}^{(1)}$, which can be done non-interactively using keys established via $\mathcal{F}_{\mathsf{Setup}}$ (Fig. 2.5). Secure

protocols are then required for matrix multiplication, ReLU and Softmax to compute shares of $\mathbf{Z}$. Additionally, a secure protocol for dReLU is required during backpropagation to compute the derivative of loss with respect to the weights $\left(\frac{\delta \mathcal{L}}{\delta \mathbf{W}}\right)$. Next, using the secret-shares of $\mathbf{Z}$ and $\left(\frac{\delta \mathcal{L}}{\delta \mathbf{W}}\right)$, servers update weight matrices. Although [133] relies on gradient descent to update weights, due to drawbacks such as slow convergence and the possibility of converging to a local minimum, we rely on the optimized alternative of Adam [132] optimizer. The weight update computations within Adam, as well as the overall steps required to evaluate a GCN, are summarised in Fig. 4.1.

To securely evaluate GCN, each step in Fig. 4.1 is realized via its secure counterpart. Elaborately, for matrix multiplication and ReLU, we rely on the secure protocols from Tetrad [138]. Although Tetrad does not give an explicit protocol for dReLU, we note that it can be computed as $\mathsf{dReLU}(x) = \mathbf{1}(x > 0)$ using the comparison protocol from Tetrad. For Softmax, we observe that Tetrad relies on an approximate variant for it. Keeping GCN accuracy as needed for real-world applications in mind, we instead compute $\mathsf{Softmax}(\mathsf{x}) = e^{\mathsf{x}}/(\sum_i e^{\mathsf{x}_i})$, which is the more accurate definition. This computation requires secure protocols for exponentiation and division. While Tetrad does not support exponentiation, for division, it relies on a garbled circuit (GC) based approach, which is known to be expensive in terms of communication cost [84, 11]. Hence, we provide efficient protocols for exponentiation as well as division. Moreover, Tetrad does not provide support for using Adam optimizer since it lacks square root primitive. Hence, we also design secure and efficient protocols for the same. We elaborate on our additions over Tetrad in §4.5. Although our approach provides efficiency and accuracy improvements in comparison to realizing GCNs via Tetrad (using SGD), we additionally incorporate the GraphSC paradigm in Entrada to further enhance efficiency, as described in §4.6.

4.5 Improvements over Tetrad

Here, we provide the building blocks that were either missing in Tetrad (double bit injection, prefix OR, exponentiation, inverse square root) or had an inefficient realization (division). The protocols for exponentiation, division and inverse square root follow from the ones in literature and are adapted to work over Tetrad while introducing optimizations where possible.

4.5.1 Double bit injection

Similar to *single* bit injection protocol of Tetrad [138], we design double bit injection or 2-bit-injection protocol ($\Pi_{\mathsf{2-bitInj}}$), which enables computing $[\![\mathsf{abv}]\!]$ given *two* Boolean shared bits

$[\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}$ and an arithmetic shared value $[\![\mathsf{v}]\!]$. Combining terms and computing them together results in our protocol having same online cost as single bit-injection, and improves online rounds and communication by $2\times$. To achieve this, observe that,

$$\begin{aligned}
\mathsf{y} &= (\mathsf{abv})^{\mathsf{R}} \\
&= (\beta_{\mathsf{a}} \oplus \alpha_{\mathsf{a}})^{\mathsf{R}} (\beta_{\mathsf{b}} \oplus \alpha_{\mathsf{b}})^{\mathsf{R}} (\mathsf{v}) \\
&= \beta_{\mathsf{a}}{}^{\mathsf{R}} \beta_{\mathsf{b}}{}^{\mathsf{R}} \beta_{\mathsf{v}} - \beta_{\mathsf{a}}{}^{\mathsf{R}} \beta_{\mathsf{b}}{}^{\mathsf{R}} \alpha_{\mathsf{v}} + \beta_{\mathsf{a}}{}^{\mathsf{R}} \left(1 - 2\beta_{\mathsf{b}}{}^{\mathsf{R}}\right) \left(\alpha_{\mathsf{b}}{}^{\mathsf{R}} \beta_{\mathsf{v}} - \alpha_{\mathsf{b}}{}^{\mathsf{R}} \alpha_{\mathsf{v}}\right) \beta_{\mathsf{b}}{}^{\mathsf{R}} \left(1 - 2\beta_{\mathsf{a}}{}^{\mathsf{R}}\right) \left(\alpha_{\mathsf{a}}{}^{\mathsf{R}} \beta_{\mathsf{v}} - \alpha_{\mathsf{a}}{}^{\mathsf{R}} \alpha_{\mathsf{v}}\right) \\
&\quad + \left(1 - 2\beta_{\mathsf{a}}{}^{\mathsf{R}}\right) \left(1 - 2\beta_{\mathsf{b}}{}^{\mathsf{R}}\right) \left(\alpha_{\mathsf{a}}{}^{\mathsf{R}} \alpha_{\mathsf{b}}{}^{\mathsf{R}} \beta_{\mathsf{v}} - \alpha_{\mathsf{a}}{}^{\mathsf{R}} \alpha_{\mathsf{b}}{}^{\mathsf{R}} \alpha_{\mathsf{v}}\right)
\end{aligned}$$

where, arithmetic equivalent of XOR, $(\mathsf{x} \oplus \mathsf{y})^{\mathsf{R}}$ is given as $\mathsf{x}^{\mathsf{R}} + \mathsf{y}^{\mathsf{R}}(1 - 2\mathsf{x}^{\mathsf{R}})$. Given that $[\![\cdot]\!]$-shares of α terms (and their products) can be generated during preprocessing (as done in Tetrad via the protocol for converting bit to its arithmetic equivalent, Π_{Bit2A}), the online phase involves generating $[\![\cdot]\!]$-shares of product terms comprising β, α via Π_{JSh} followed by local addition to generate $[\![\mathsf{y}]\!]$. The formal protocol appears in Fig. 4.2.

Protocol $\Pi_{2-\mathsf{bitInj}}\left([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{v}]\!]\right)$

Preprocessing

– Invoke preprocessing phase of Π_{Bit2A} to generate $[\mathsf{u_a}] = \left[\alpha_{\mathsf{a}}{}^{\mathsf{R}}\right]$ and $[\mathsf{u_b}] = \left[\alpha_{\mathsf{b}}{}^{\mathsf{R}}\right]$.

– Invoke Π_{MulR} to compute $[\mathsf{u_{ab}}] = \left[\alpha_{\mathsf{a}}{}^{\mathsf{R}} \alpha_{\mathsf{b}}{}^{\mathsf{R}}\right]$, $[\mathsf{u_{av}}] = \left[\alpha_{\mathsf{a}}{}^{\mathsf{R}} \alpha_{\mathsf{v}}\right]$, $[\mathsf{u_{bv}}] = \left[\alpha_{\mathsf{b}}{}^{\mathsf{R}} \alpha_{\mathsf{v}}\right]$ and $[\mathsf{u_{abv}}] = \left[\mathsf{u_{ab}} \alpha_{\mathsf{v}}{}^{\mathsf{R}}\right]$.

Online

Let $\mathsf{x_a} = \beta_{\mathsf{a}}{}^{\mathsf{R}}, \mathsf{x_b} = \beta_{\mathsf{b}}{}^{\mathsf{R}}, \mathsf{x_{ab}} = (\beta_{\mathsf{a}} \beta_{\mathsf{b}})^{\mathsf{R}}$, and $\mathsf{x_{abv}} = (\beta_{\mathsf{a}} \beta_{\mathsf{b}})^{\mathsf{R}} \beta_{\mathsf{v}}$.

– P_1, P_2 compute $\mathsf{y}_1 = \mathsf{x_{abv}} - \mathsf{x_{ab}} \mathsf{u}_{\mathsf{v}1} + \mathsf{x_a}(1 - 2\mathsf{x_b})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{b}1} - \mathsf{u}_{\mathsf{bv}1}) + \mathsf{x_b}(1 - 2\mathsf{x_a})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{a}1} - \mathsf{u}_{\mathsf{av}1}) + (1 - 2\mathsf{x_a})(1 - 2\mathsf{x_b})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{ab}1} - \mathsf{u}_{\mathsf{abv}1})$.

– P_2, P_3 compute $\mathsf{y}_2 = -\mathsf{x_{ab}} \mathsf{u}_{\mathsf{v}2} + \mathsf{x_a}(1 - 2\mathsf{x_b})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{b}2} - \mathsf{u}_{\mathsf{bv}2}) + \mathsf{x_b}(1 - 2\mathsf{x_a})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{a}2} - \mathsf{u}_{\mathsf{av}2}) + (1 - 2\mathsf{x_a})(1 - 2\mathsf{x_b})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{ab}2} - \mathsf{u}_{\mathsf{abv}2})$.

– P_3, P_1 compute $\mathsf{y}_3 = -\mathsf{x_{ab}} \mathsf{u}_{\mathsf{v}3} + \mathsf{x_a}(1 - 2\mathsf{x_b})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{b}3} - \mathsf{u}_{\mathsf{bv}3}) + \mathsf{x_b}(1 - 2\mathsf{x_a})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{a}3} - \mathsf{u}_{\mathsf{av}3}) + (1 - 2\mathsf{x_a})(1 - 2\mathsf{x_b})(\beta_{\mathsf{v}} \mathsf{u}_{\mathsf{ab}3} - \mathsf{u}_{\mathsf{abv}3})$.

– $(P_1, P_2), (P_2, P_3)$ and (P_3, P_1) invoke Π_{JSh} on $\mathsf{y}_1, \mathsf{y}_2$ and y_3, respectively, to generate $[\![\mathsf{y}_1]\!], [\![\mathsf{y}_2]\!]$ and $[\![\mathsf{y}_3]\!]$.

– Output $[\![\mathsf{y}]\!] = [\![\mathsf{y}_1]\!] + [\![\mathsf{y}_2]\!] + [\![\mathsf{y}_3]\!]$.

Figure 4.2: Double bit injection.

4.5.2 Prefix OR

This protocol, denoted as Π_{PreOr}, forms an important building block in the division, square root, and inverse square root protocols. On input Boolean shared bits $\mathsf{x}_{\ell-1}, \ldots, \mathsf{x}_0$, it outputs Boolean shared bits $\mathsf{y}_{\ell-1}, \ldots, \mathsf{y}_0$ such that $\mathsf{y}_i = \vee_{j=i}^{\ell-1} \mathsf{x}_j$. This protocol proceeds along similar lines as described in §3.4.1. Hence, we omit the details here.

4.5.3 Exponentiation

Denoted as Π_{Exp}, the protocol for exponentiation outputs $[\![e^{\mathsf{x}}]\!]$ on input $[\![\mathsf{x}]\!]$. Although our protocol is inspired from [6, 63, 126], it is much more efficient due to avoiding the need for an explicit division operation, avoiding reliance on edabits [79], and a few intermediate conversions. In the following, we first give an overview of the protocol of [6] followed by detailing our improvements over it.

To compute e^{x}, [6] proceeds as follows. It first computes the absolute value of x, denoted as $|\mathsf{x}|$, by obliviously selecting between $(\mathsf{x}, -\mathsf{x})$, depending on its sign. Then, $|\mathsf{x}|$ is split into its fractional (r) and integer (t) parts. Observe that $e^{|\mathsf{x}|} = e^{\mathsf{t}+\mathsf{r}} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$. Thus, the task reduces to computing e^{t} and e^{r}, where t, r are $[\![\cdot]\!]$-shared. e^{r} is computed using polynomial approximation. To compute e^{t}, t is decomposed into bits $\{\mathsf{t}_{\mathsf{k}-1}, \ldots, \mathsf{t}_0\}$, and bit-wise exponentiations are used as follows

$$e^{|\mathsf{t}|} = \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}} = \prod_{j=\mathsf{f}}^{\mathsf{k}-2} \left(\mathsf{t}_j \cdot \left(e^{2^{j-\mathsf{f}}} - 1\right) + 1\right)$$

where the value $e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}$ is computed by obliviously selecting between 1 and $e^{2^{j-\mathsf{f}}}$ with t_j as the selection bit.

Observe that if $\mathsf{x} > 0$, one can safely output $e^{|\mathsf{x}|} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$. However, if $\mathsf{x} < 0$, the value to be returned is $\frac{1}{e^{|\mathsf{x}|}}$. The latter additionally requires one call to secure division. Further, to ensure that no information about the sign of x is revealed, the work of [6] computes both the values $e^{|\mathsf{x}|}$ and $\frac{1}{e^{|\mathsf{x}|}}$, and then obliviously selects between them depending on the sign of x. The call to division increases the complexity of this protocol.

Unlike the approach of [6], we implicitly account for the sign of x while performing the bit-wise exponentiation. This allows us to explicitly avoid computing $\frac{1}{e^{|\mathsf{x}|}}$, and thereby the call to the secure division protocol. Elaborately, we rely on splitting x into its fractional (r) and integer (t) parts, rather than splitting $|\mathsf{x}|$, as done in [6]. The goal is to first compute $e^{\mathsf{t}}, e^{\mathsf{r}}$ and then $e^{\mathsf{x}} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$. We rely on Taylor series approximation to compute e^{r}, which implicitly

accounts for the sign of x. To compute e^{t} while accounting for the sign, we make the following observation. Let $\{\mathsf{t}_i\}_{i=0}^{\mathsf{k}-1}$ denote the bits in t. Then,

$$e^{\mathsf{t}} = \begin{cases} \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{if } \mathsf{x} \geq 0 \\ \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{-\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{otherwise} \end{cases}$$

Note that selection between the two cases in the above equation is handled obliviously using Π_{Sel} on inputs $e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}$, $e^{-\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}$ with the MSB of x as the selection bit. In this way, we avoid relying on a division protocol. Concretely, the value to be computed boils down to the following.

$$\begin{aligned} e^{\mathsf{t}} &= \prod_{j=\mathsf{f}}^{\mathsf{k}-2} \mathsf{t}_j \left(\mathsf{s} \left(\mathsf{e}^{-2^{j-\mathsf{f}}} - \mathsf{e}^{2^{j-\mathsf{f}}} \right) + \mathsf{e}^{2^{j-\mathsf{f}}} - 1 \right) + 1 \\ &= \prod_{j=\mathsf{f}}^{\mathsf{k}-2} \mathsf{t}_j \left(\mathsf{s} \left(\mathsf{e}^{-2^{j-\mathsf{f}}} - \mathsf{e}^{2^{j-\mathsf{f}}} \right) \right) + \mathsf{t}_j \left(\mathsf{e}^{2^{j-\mathsf{f}}} - 1 \right) + 1 \end{aligned}$$

where s denotes the sign bit of x and is a 1 if $\mathsf{x} < 0$. The formal protocol steps are provided in Fig. 4.3.

Protocol $\Pi_{\mathsf{Exp}}(\llbracket \mathsf{x} \rrbracket)$

- $\llbracket \mathsf{t} \rrbracket = \Pi_{\mathsf{trunc}}(\llbracket \mathsf{x} \rrbracket, \mathsf{f}) \cdot 2^{\mathsf{f}}$
- $\llbracket \mathsf{r} \rrbracket = \llbracket \mathsf{x} \rrbracket - \llbracket \mathsf{t} \rrbracket$, $\llbracket \mathsf{t} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\llbracket \mathsf{t} \rrbracket)$ and $\llbracket \mathsf{s} \rrbracket^{\mathbf{B}} = \llbracket \mathsf{t}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}$
- for $i = 0$ to $\mathsf{k} - 2$ do: $\llbracket \mathsf{t}_i \rrbracket^{\mathbf{B}} = \llbracket \mathsf{t}_i \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{s} \rrbracket^{\mathbf{B}}$
- for $j = \mathsf{f}$ to $\mathsf{k} - 2$ do:
  - $\llbracket \mathsf{e}'_j \rrbracket = \Pi_{\mathsf{BitInj}} \left(\llbracket \mathsf{t}_j \rrbracket^{\mathbf{B}}, e^{2^{j-\mathsf{f}}} - 1 \right)$
  - $\llbracket \mathsf{e}_j \rrbracket = \llbracket \mathsf{e}'_j \rrbracket + \Pi_{2-\mathsf{bitInj}} \left(\llbracket \mathsf{s} \rrbracket^{\mathbf{B}}, \llbracket \mathsf{t}_j \rrbracket^{\mathbf{B}}, e^{-2^{j-\mathsf{f}}} - e^{2^{j-\mathsf{f}}} \right) + 1$
- set $\llbracket \mathsf{d} \rrbracket = \llbracket \mathsf{e}_f \rrbracket$
- for $j = \mathsf{f} + 1$ to $\mathsf{k} - 2$ do: $\llbracket \mathsf{d} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{e}_j \rrbracket, \mathsf{f})$
- $\llbracket \mathsf{z} \rrbracket = \Pi_{\mathsf{Sel}} \left(1, 1/e, \llbracket \mathsf{s} \rrbracket^{\mathbf{B}} \right)$ and $\llbracket \mathsf{d} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{z} \rrbracket, \mathsf{f})$
- set $\llbracket \mathsf{b}_0 \rrbracket = 1$, $\llbracket \mathsf{b}_1 \rrbracket = \llbracket \mathsf{r} \rrbracket$
- for $i = 2$ to θ do: $\llbracket \mathsf{b}_i \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{b}_{i-1} \rrbracket, \llbracket \mathsf{b}_1 \rrbracket, \mathsf{f})$
- $\llbracket \mathsf{b} \rrbracket = \sum_{i=0}^{\theta} \frac{\llbracket \mathsf{b}_i \rrbracket}{i!}$
- $\llbracket \mathsf{g} \rrbracket = \Pi_{\mathsf{Mul}}(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{b} \rrbracket, \mathsf{f})$
- Return $\llbracket \mathsf{g} \rrbracket$

Figure 4.3: Exponentiation.

While the high-level approach of our protocol is similar to that of the one in [126], we note

that our optimizations, such as reliance on double bit injection, avoiding the need for an explicit bit extraction circuit for computing the msb of x, etc., further aid in improving the efficiency of our exponentiation protocol. In our work, we take $\theta = 4$ since a higher value does not aid in improving the accuracy for GCNs.

4.5.4 Division

The garbled circuit-based division in Tetrad is known to be expensive [11, 12]. Thus, we propose a division protocol, Π_{Div}, that relies on Goldschmidt's approximation and follows a similar approach as in [43]. This protocol on input $[\![\mathsf{a}]\!]$ and $[\![\mathsf{b}]\!]$, outputs $[\![\mathsf{d}]\!]$ where $\mathsf{d} \approx \mathsf{a}/\mathsf{b}$ via an iterative approach. The protocol is similar to as described in §3.4.2, and hence we omit the details here.

4.5.5 Inverse square root

The protocol Π_{InvSqrt}, on input $[\![\mathsf{a}]\!]$, outputs $[\![\mathsf{y}]\!]$ where $\mathsf{y} \approx 1/\sqrt{\mathsf{a}}$. Our protocol follows on the lines of [126, 163] and uses a polynomial approximation to compute the inverse square root of x. Similar to division, to get a good approximation of y, the input a is first normalized to $\mathsf{a}' \in (0.25, 0.5]$. Here, $\mathsf{a}' = \mathsf{a} \cdot \mathsf{v}$, and $\mathsf{v} = 2^{-(e+1)}$ is the scaling factor (where the most significant non-zero bit of a appears at index $e + \mathsf{f}$ in the bit representation of a, assuming v has f bit precision). Then $1/\sqrt{\mathsf{a}}$ is given by:

$$\frac{1}{\sqrt{\mathsf{a}}} = \left(\frac{1}{\sqrt{\mathsf{a}'}}\right) \cdot 2^{-(e+1)/2} \tag{4.5}$$

Here, $1/\sqrt{\mathsf{a}'}$ is approximated using a low degree polynomial

$$g(\mathsf{a}') = 4.63887\mathsf{a}'^2 - 5.77789\mathsf{a}' + 3.14736 \tag{4.6}$$

The protocol $\Pi_{\mathsf{PreInvSqrt}}$ computes the normalized input a'. Additionally, it computes $\mathsf{v}' = 2^{-(e+1)/2}$, as required in Equation (4.5). We now give an overview of how these two components are generated. To compute the normalized input $\mathsf{a}' = \mathsf{a} \cdot \mathsf{v}$, we first compute $\mathsf{v} = 2^{\mathsf{f}-(e+1)}$ (accounting for f bit precision). For this, the input a is decomposed to obtain its bit-wise Boolean representation. These bits are manipulated using prefix OR and local operations to compute v. Similarly, $\mathsf{v}' = 2^{\mathsf{f}-\frac{(e+1)}{2}}$ is also computed with f bits of precision. Observe that v' can be computed by shifting the bit at index $e+\mathsf{f}+1$ to index $(e+\mathsf{f}+1)/2$. This is performed obliviously by computing the XOR of every consecutive pair of bits of v. Following this, the v'

is multiplied by $2^{\frac{\mathsf{f}}{2}}$ to compute $2^{\mathsf{f}-(e+1)/2}$. Observe that when $(e+\mathsf{f}+1)$ is odd, the value is computed correctly. However, when $(e+\mathsf{f}+1)$ is even, the value computed is offset by a factor of $\sqrt{2}$. Hence, v' should be instead multiplied by $2^{(\frac{\mathsf{f}+1}{2})}$ to cancel out the offset. To handle both cases obliviously, we compute r, which denotes the parity of $(e+\mathsf{f}+1)$. This is computed as the XOR of bits in v indexed with odd numbers. Then, invoking the $\Pi_{\mathsf{Sel}}(2^{(\frac{\mathsf{f}}{2})}, 2^{(\frac{\mathsf{f}+1}{2})}, [\![\mathsf{r}]\!]^{\mathbf{B}})$ allows computing the correct inverse square root of the scaling factor obliviously. Details of $\Pi_{\mathsf{PreInvSqrt}}$ appear in Fig. 4.4.

Given a', v', the protocol Π_{InvSqrt} computes the inverse square root of the input a as follows. The approximate inverse square root of the normalized input, denoted as y, is computed using the polynomial provided in Equation (4.6). Following this, the inverse square root of the input a is given by $\frac{1}{\sqrt{\mathsf{a}}} = \mathsf{y} \cdot \mathsf{v}'$, which follows from Equation (4.5). The formal protocol appears in Fig. 4.5.

Protocol $\Pi_{\mathsf{PreInvSqrt}}([\![\mathsf{a}]\!])$

- $[\![\mathsf{a}]\!]^{\mathbf{B}} = \Pi_{\mathsf{A2B}}([\![\mathsf{a}]\!])$
- $\{[\![\mathsf{c}_i]\!]^{\mathbf{B}}\}_{i=\mathsf{k}-2}^{0} = \Pi_{\mathsf{PreOr}}\left(\{[\![\mathsf{a}_i]\!]^{\mathbf{B}}\}_{i=\mathsf{k}-2}^{0}\right)$ and $[\![\mathsf{c}_{\mathsf{k}-1}]\!]^{\mathbf{B}} = [\![0]\!]^{\mathbf{B}}$
- $[\![\mathsf{v}_{\mathsf{k}-1}]\!]^{\mathbf{B}} = [\![\mathsf{c}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$
- for $i = \mathsf{k}-2$ to 1 do: $[\![\mathsf{v}_i]\!]^{\mathbf{B}} = [\![\mathsf{c}_i]\!]^{\mathbf{B}} \oplus [\![\mathsf{c}_{i+1}]\!]^{\mathbf{B}}$
- $[\![\mathsf{v}]\!] = \Pi_{\mathsf{B2A}}\left(\{[\![\mathsf{v}_i]\!]\}_{i=2\mathsf{f}-1}^{0}\right)$ (in reverse order)
- $[\![\mathsf{a}']\!] = \Pi_{\mathsf{Mul}}([\![\mathsf{a}]\!], [\![\mathsf{v}]\!], \mathsf{f})$
- $\mathsf{k}' = \frac{\mathsf{k}}{2}, \mathsf{f}' = \frac{\mathsf{f}}{2}, c_0 = 2^{\frac{\mathsf{f}+1}{2}}, c_1 = 2^{\frac{\mathsf{f}}{2}}$
- for $i = 0$ to k' do: $[\![\mathsf{v}'_i]\!]^{\mathbf{B}} = [\![\mathsf{v}_{2i}]\!]^{\mathbf{B}} \oplus [\![\mathsf{v}_{2i+1}]\!]^{\mathbf{B}}$
- $[\![\mathsf{r}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} \oplus [\![\mathsf{v}_3]\!]^{\mathbf{B}} \oplus [\![\mathsf{v}_5]\!]^{\mathbf{B}} \oplus \ldots [\![\mathsf{v}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$
- $[\![\mathsf{v}']\!] = \Pi_{\mathsf{B2A}}\left(\{[\![\mathsf{v}'_i]\!]^{\mathbf{B}}\}_{i=2\mathsf{f}'-1}^{0}\right)$
- $[\![\mathsf{v}']\!] = \Pi_{\mathsf{Sel}}(\mathsf{c}_0, \mathsf{c}_1, [\![\mathsf{r}]\!]) \cdot [\![\mathsf{v}]\!]$
- return $[\![\mathsf{a}']\!], [\![\mathsf{v}']\!]$

Figure 4.4: Sub-protocol for inverse square root.

Protocol $\Pi_{\mathsf{InvSqrt}}([\![\mathsf{a}]\!])$

- $[\![\mathsf{a}']\!], [\![\mathsf{v}']\!] = \Pi_{\mathsf{PreInvSqrt}}([\![\mathsf{a}]\!])$
- $[\![\mathsf{y}_0]\!] = [\![\mathsf{a}']\!], [\![\mathsf{y}_1]\!] = \Pi_{\mathsf{Mul}}([\![\mathsf{a}']\!], [\![\mathsf{a}']\!], \mathsf{f})$
- $[\![\mathsf{y}]\!] = 4.63887[\![\mathsf{y}_1]\!] - 5.77789[\![\mathsf{y}_0]\!] + 3.14736$
- return $\Pi_{\mathsf{Mul}}([\![\mathsf{y}]\!], [\![\mathsf{v}']\!], \mathsf{f})$

Figure 4.5: Inverse square root.

Security

Since our constructions use primitives from Tetrad, their security follows directly from the security of the underlying protocols. Formal proofs are provided in §4.8.

Complexity of the designed primitives

In this section, we discuss the complexity of the proposed building blocks and primitives.

Building block	Online		Preprocessing
	Rounds	Comm. (in bits)	Comm. (in bits)
Joint sharing^{+}	1	2ℓ	-
Multiplication	1	3ℓ	2ℓ
3-input Multiplication	1	3	9
4-input Multiplication	1	3	24
MulR†	-	-	3ℓ
Multiplication with truncation	1	3ℓ	2ℓ
Bit to arithmetic	1	3ℓ	$3\ell + 1$
Boolean to arithmetic	1	3ℓ	$192\ell + 1$
Arithmetic to Boolean	$\log_4 \ell$	u_1	$\ell + \mathsf{u}'_1$
Bit extraction	$\log_4 \ell$	u_2	$\ell + \mathsf{u}'_2$
Bit injection	1	3ℓ	$6\ell + 1$
Comparison	$\log_4 \ell$	u_2	$\ell + \mathsf{u}'_2$
Oblivious select	1	3ℓ	$6\ell + 1$

- ℓ: size of ring in bits, instantiated with $\ell = 64$ in our work
- $\mathsf{u}'_1 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_1 = 3(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [194] with $2, 3, 4$ inputs, respectively.
- $\mathsf{u}'_2 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_2 = 3(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 41, \mathsf{n}_3 = 27, \mathsf{n}_4 = 47$ denote the number of AND gates in the optimized bit extraction circuit of [194] with $2, 3, 4$ inputs, respectively.
$^{+}$ Joint-sharing a value v in the preprocessing phase where v is held by P_0 along with another party, has a communication cost of only ℓ bits. We refer readers to [138] for further details.
† The protocol MulR is only invoked in the preprocessing phase of $\Pi_{2-\mathsf{bitInj}}$

Table 4.2: Complexity of building blocks of Tetrad [138].

Double bit injection The protocol $\Pi_{2-\mathsf{bitInj}}$ requires communication of 3ℓ bits and 1 round in the online phase and $18\ell + 2$ bits in the preprocessing phase. This cost is explained as follows. In the online phase of $\Pi_{2-\mathsf{bitInj}}$ parties compute $\mathsf{y}_1, \mathsf{y}_2$ and y_3 locally. Following this the

parties $(P_1, P_2), (P_2, P_3), (P_1, P_3)$ joint share $\mathsf{y}_1, \mathsf{y}_2$ and y_3 respectively in parallel. This requires 1 round and communication of 3ℓ bits in the online phase. The communication cost for the preprocessing phase follows from Table 4.2.

Prefix OR The protocol Π_{PreOr} requires communication of 432 bits and 3 rounds in the online phase and 1680 bits in the preprocessing phase. This cost is explained as follows. For our choice of $\mathsf{k} = 32$, the prefix OR can be computed in $\log_4(32) = 3$ rounds. Observe that each round has 16 invocations of Π_{Mul}, $\Pi_{\mathsf{3\text{-}Mul}}$, and $\Pi_{\mathsf{4\text{-}Mul}}$ that are computed in parallel. Thus, the communication cost for the online phase is 432 bits, and for the preprocessing phase is 1680 bits.

Exponentiation The protocol Π_{Exp} requires communication of $156\ell + \mathsf{u}_1$ bits and 10 rounds in the online phase and $364\ell + \mathsf{u}'_1$ bits in the preprocessing phase. Here, $\mathsf{u}'_1 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_1 = 3(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [194] with $2, 3, 4$ inputs, respectively. This cost is explained as follows. Parties first truncate the input to compute the sharing of the integer part ($[\![\mathsf{t}]\!]$) of the input, followed by a call to Π_{A2B} to get the bitwise sharing of the same. This requires 4 (i.e., 1+3) rounds and communication of $\mathsf{u}_1 + 3\ell$ bits (see Table 4.2) in the online phase. The parties then XOR the bits of $[\![\mathsf{t}]\!]$ with the MSB of $[\![\mathsf{t}]\!]$, which is computed non-interactively. Following this, the exponentiation for the integer part and the decimal part is computed in parallel. For the integer part, the bitwise exponentiation is computed in parallel. This requires one call to Π_{BitInj} and $\Pi_{\mathsf{2-bitInj}}$ with respect to each bit. For our choice of $\mathsf{k} = 32$ and $\mathsf{f} = 16$, there are 15 bits that correspond to the integer part. Hence, the total cost for this step is 1 round, and $15 \times (3\ell + 3\ell) = 90\ell$ bits of communication in the online phase. Following this, the parties multiply the computed bitwise exponentiation to obtain $[\![\mathsf{e}^{\mathsf{t}}]\!]$. This requires 15 multiplications which can be computed in 4 rounds, and communication of 45ℓ bits. To compute the exponentiation of the decimal part, the Taylor series approximation is used. We use $\theta = 4$ which gives the desired accuracy. Each iteration has one multiplication. Thus, the communication cost for computing the exponentiation of the decimal part is 12ℓ bits in the online phase. Finally, the exponentiation of the integer part and the decimal part is multiplied to get the output which requires 1 round and 3ℓ bits of communication in the online phase. Thus, the protocol Π_{Exp} requires 10 rounds and $364\ell + \mathsf{u}_1$ bits of communication in the online phase. The preprocessing cost follows from Table 4.2.

Division The protocol Π_{AppRec} requires communication of $15\ell + \mathsf{u}_1 + 432$ bits and 9 rounds in the online phase and $205\ell + \mathsf{u}'_1 + 1681$ bits in the preprocessing phase. Here, $\mathsf{u}'_1 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_1 = 3(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [194] with $2, 3, 4$ inputs, respectively. This cost is explained as follows. Parties run one instance of Π_{A2B}, which requires communication of u_1 bits (ref. Table 4.2) and 3 rounds in the online phase. This is followed by one instance of Π_{PreOr}, which requires 3 rounds and 432 bits of communication. Next, the parties run one instance of Π_{B2A}, which requires 1 round and 3ℓ communication. Next, the parties run two instances of multiplication in parallel to compute $[\![\mathsf{z}]\!]^{\mathbf{B}}$ and $[\![\mathsf{w}']\!]$ along with one instance of Π_{Sel}. This requires 1 round and has a communication cost of 9ℓ bits. Finally, the parties run one instance of multiplication to compute the initial approximation $[\![\mathsf{w}]\!]$. This constitutes one round and 3ℓ bits of communication. Thus the total online cost for Π_{AppRec} is 10 rounds and $15\ell + u_1 + 432$ bits in the online phase. The preprocessing cost follows from Table 4.2 and the cost of Π_{PreOr}.

The protocol Π_{Div} requires communication of $42\ell + \mathsf{u}_1 + 432$ bits and 14 rounds in the online phase and $223\ell + \mathsf{u}'_1 + 1681$ bits in the preprocessing phase. This cost is explained as follows. Parties invoke 1 instance of Π_{AppRec}, which has a communication cost of $15\ell + u_1 + 432$ and constitutes 9 rounds. This is followed by Goldschmidt's approximation. We observe that for our choice of $\mathsf{k} = 32$ and $\mathsf{f} = 16$, three iterations (i.e $\theta = 4$) of Goldschmidt's approximation are required to obtain good accuracy. In each iteration, the servers run 2 multiplications in parallel, which constitutes 1 round and communication of 6ℓ bits per round. Finally, the parties run another multiplication to get the final result. Thus the total number of rounds required is 14, and the communication cost is $42\ell + u_1 + 432$. The preprocessing cost follows from Table 4.2 and the cost of Π_{AppRec}.

Inverse square root The protocol $\Pi_{\mathsf{PreInvSqrt}}$ requires communication of $9\ell + \mathsf{u}_1 + 432$ bits and 8 rounds in the online phase and $387\ell + \mathsf{u}'_1 + 1682$ bits in the preprocessing phase. Here, $\mathsf{u}'_1 = 2\mathsf{n}_2 + 9\mathsf{n}_3 + 24\mathsf{n}_4$, $\mathsf{u}_1 = 3(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [194] with $2, 3, 4$ inputs, respectively. This cost is explained as follows. Parties invoke Π_{A2B} to decompose the input (a) into bits. This is followed by computing the prefix OR of the bits and XORing the bits with the MSB of the input, similar to the protocol Π_{AppRec}. This requires 4 rounds and communication $\mathsf{u}_1 + 432$ bits in the online phase. This is followed by an invocation of Π_{B2A} to compute the scaling factor v and multiplication to compute the scaled input. This requires 2 rounds and communication of 6ℓ bits in the online phase. In parallel, the parties also compute the square root of the scaling factor(v'), which also requires one call to Π_{B2A}, which requires communication of 3ℓ bits. The

bit r, which denotes the parity of the scaling factor, is computed non-interactively. Thus the protocol $\Pi_{\mathsf{PreInvSqrt}}$ requires 8 rounds and $9\ell+432+\mathsf{u}_1$ bits of communication. The preprocessing cost follows from Table 4.2 and the cost of Π_{PreOr}.

The protocol Π_{InvSqrt} requires a communication of $15\ell + \mathsf{u}_1 + 432$ bits and 10 rounds in the online phase and $391\ell + \mathsf{u}'_1 + 1682$ bits in the preprocessing phase. This cost is explained as follows. Parties invoke $\Pi_{\mathsf{PreInvSqrt}}$ to compute the scaled input and the square root of the scaling factor. Following this, parties compute the square of the scaled input, which requires one multiplication. Following this, the inverse square root of the scaled input is computed non-interactively using the polynomial provided in (4.6). Finally, the parties invoke one call to multiplications to compute the output. Thus the protocol Π_{InvSqrt} requires 10 rounds and $15\ell + 432 + \mathsf{u}_1$ bits of communication in the online phase. The preprocessing cost follows from Table 4.2 and the cost of $\Pi_{\mathsf{PreInvSqrt}}$.

4.6 GCN evaluation via GraphSC

Recall that computations via GraphSC require performing shuffles and invocations of Scatter and Gather operations across multiple rounds. Since we instantiate the MPC for GraphSC via Entrada, we first describe our shuffle protocol over the same. Following this, we discuss the components of GCN evaluation that can be cast in the message-passing paradigm and subsequently define the Scatter-Gather primitives for the same. This entails defining GCN computations in a vertex-centric manner, unlike matrix operations, as described in §4.3.3. While the forward pass of GCN can be entirely computed within GraphSC, for the backward pass, the computation of derivatives of the loss function benefits from the GraphSC paradigm. Thus, other computations, such as updating the weight matrices, are performed outside GraphSC. We conclude by discussing the steps for generating the shares of the graph list G, as required for GraphSC, from the matrix representation, which is generated as part of the input sharing phase.

4.6.1 Secure shuffle

Consider a $[\![\cdot]\!]$-shared N-sized vector $\mathbf{v}$, where each element $\mathbf{v}[i] \in \mathbf{v}$ for $i \in \{1, \ldots, \mathsf{N}\}$ is $[\![\cdot]\!]$-shared. The goal of secure shuffle protocol is to generate a $[\![\cdot]\!]$-shared vector $\mathbf{u}$ such that it comprises the elements in $\mathbf{v}$ shuffled under a secret random permutation π. We denote this operation as $\mathbf{u} = \pi(\mathbf{v})$, where $\mathbf{u}$ is the vector of elements $\mathbf{v}[\pi(1)], \mathbf{v}[\pi(2)], \ldots, \mathbf{v}[\pi(\mathsf{N})]$. To ensure that π is secret and hidden from all parties, following along the lines of [145], we define π to

be a composition of four permutations $\pi = \pi_0 \circ \pi_3 \circ \pi_1 \circ \pi_2$ such that each party P_i misses the permutation π_i, and $\circ$ denotes the composition operation. The justification for the ordering among the permutations is made clear later and follows from the construction of our shuffle protocol. Since the protocol heavily relies on the sharing semantics of Tetrad and Π_{Jmp} (enables $P_i, P_j \in \mathcal{P}$ to send v to P_k such that P_k receives the correct v, or a conflicting pair of parties among P_i, P_j, P_k is identified), Π_{JSh} (enables $P_i, P_j \in \mathcal{P}$ to generate $[\![\mathsf{v}]\!]$ where $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is held by P_i, P_j), we refer the readers to §2.4 to familiarize themselves with the same.

As per $[\![\cdot]\!]$-sharing semantics, $\mathbf{u} = \beta_{\mathbf{u}} - \alpha_{\mathbf{u}}$, where $\alpha_{\mathbf{u}}$ is $[\cdot]$-shared. Thus, to generate $[\![\mathbf{u}]\!]$, our goal is to generate $\beta_{\mathbf{u}}, \alpha_{\mathbf{u}} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ such that $\mathbf{u} = \pi(\mathbf{v}) = \pi(\beta_{\mathbf{v}} - \alpha_{\mathbf{v}}) = \pi(\beta_{\mathbf{v}}) - \pi(\alpha_{\mathbf{v}}) = \beta_{\mathbf{u}} - \alpha_{\mathbf{u}}$, and $\alpha_{\mathbf{u}}$ is $[\cdot]$-shared. We explain the generation of $[\![\mathbf{u}]\!]$ into two parts– (1) assuming that parties have generated $[\![\cdot]\!]$-shares of $\mathbf{w} = \pi'(\mathbf{v})$ where $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$, we explain how $[\![\cdot]\!]$-shares of $\mathbf{u} = \pi_0(\mathbf{w})$ can be generated (observe here that $\mathbf{u} = \pi(\mathbf{v}) = \pi_0(\mathbf{w})$ holds true since $\pi = \pi_0 \circ \pi'$), (2) we then explain how $[\![\cdot]\!]$-shares of $\mathbf{w} = \pi'(\mathbf{v})$ can be generated where $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$.

We now describe how to realize (1) assuming that $[\![\cdot]\!]$-shares of $\mathbf{w}$ are available, i.e., $\mathbf{w} = \beta_{\mathbf{w}} - \alpha_{\mathbf{w}}$ where $\alpha_{\mathbf{w}}$ is $[\cdot]$-shared and can be written as $\alpha_{\mathbf{w}} = \alpha_{\mathbf{w}1} + \alpha_{\mathbf{w}2} + \alpha_{\mathbf{w}3}$. Observe that $\mathbf{u} = \pi_0(\mathbf{w}) = \pi_0(\beta_{\mathbf{w}} - \alpha_{\mathbf{w}}) = \pi_0(\beta_{\mathbf{w}}) - \pi_0(\alpha_{\mathbf{w}1}) - \pi_0(\alpha_{\mathbf{w}2}) - \pi_0(\alpha_{\mathbf{w}3})$. Thus, $[\![\cdot]\!]$-shares of $\mathbf{u}$ can be generated by linearly combining the $[\![\cdot]\!]$-shares of $\pi_0(\beta_{\mathbf{w}}), \pi_0(\alpha_{\mathbf{w}1}), \pi_0(\alpha_{\mathbf{w}2}), \pi_0(\alpha_{\mathbf{w}3})$. To generate the $[\![\cdot]\!]$-shares of $\pi_0(\beta_{\mathbf{w}})$, observe that P_1, P_2, P_3 hold π_0 as well as $\beta_{\mathbf{w}}$. Hence, parties can generate $[\![\cdot]\!]$-shares of $\pi_0(\beta_{\mathbf{w}})$ non-interactively in the online phase by retaining the masked value as $\pi_0(\beta_{\mathbf{w}})$ and setting the mask as 0 (i.e., $[\cdot]$-shares of α are set to be 0). On the other hand, each of the remainder terms $\pi_0(\alpha_{\mathbf{w}i})$ for $i \in \{1, 2, 3\}$ is held by a distinct pair of parties in P_1, P_2, P_3. Hence, in the preprocessing phase, the corresponding pair of parties invoke Π_{JSh} to generate $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}i})$ among all the parties. This completes generation of $[\![\cdot]\!]$-shares of $\mathbf{u} = \pi_0(\mathbf{w})$.

We now explain how $[\![\cdot]\!]$-shares of $\mathbf{w} = \pi'(\mathbf{v})$ are generated where $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$ and $\mathbf{v} = \beta_{\mathbf{v}} - \alpha_{\mathbf{v}}$ such that $\alpha_{\mathbf{v}}$ is $[\cdot]$-shared. One of our main goals in realizing this is to minimize the complexity of the online phase. Towards achieving this, our high level approach to generate $[\![\cdot]\!]$-shares of $\mathbf{w}$ is to generate $[\alpha_{\mathbf{w}}]$ non-interactively, and define $\beta_{\mathbf{w}}$ as $\beta_{\mathbf{w}} = \pi'(\beta_{\mathbf{v}}) - \pi'(\alpha_{\mathbf{v}}) + \alpha_{\mathbf{w}}$. During the generation of the $[\![\cdot]\!]$-shares of $\mathbf{w}$, we maintain the invariant that every message to be communicated is held by two parties, which allows invoking the Π_{Jmp} protocol. This guarantees the correctness of the generated shares since the invocation of Π_{Jmp} ensures that any misbehaviour by a malicious party can be detected. In the following, we begin by explaining how $[\cdot]$-shares of $\alpha_{\mathbf{w}}$ can be generated followed by generation of $\beta_{\mathbf{w}}$ towards P_2, P_1, P_3. Since the composition of permutations in $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$ is non-commutative, generation of $\beta_{\mathbf{w}}$ is handled differently towards each party.

Generation of $[\cdot]$-shares of $\alpha_{\mathbf{w}}$ Relying on keys generated via $\mathcal{F}_{\mathsf{Setup}}$, parties non-interactively generate $[\cdot]$-shares of $\alpha_{\mathbf{w}}$ in the preprocessing phase, i.e., P_0, P_1, P_3 sample $\alpha_{\mathbf{w}1} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, P_0, P_2, P_3 sample $\alpha_{\mathbf{w}2} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, and P_0, P_1, P_2 sample $\alpha_{\mathbf{w}3} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$.

Generation of $\beta_{\mathbf{w}}$ towards P_2 We begin with the case of generating $\beta_{\mathbf{w}} = \pi'(\beta_{\mathbf{v}}) - \pi'(\alpha_{\mathbf{v}}) + \alpha_{\mathbf{w}}$ towards P_2. Consider the first summand $\pi'(\beta_{\mathbf{v}}) = \pi_3(\pi_1(\pi_2(\beta_{\mathbf{v}})))$. To compute this term, P_2 only misses π_2. Thus, if $\pi_2(\beta_{\mathbf{v}})$ can be made available to P_2, it can compute the first summand. Since $\pi_2(\beta_{\mathbf{v}})$ is held by both P_1, P_3, they can invoke Π_{Jmp} to send $\pi_2(\beta_{\mathbf{v}})$ to P_2. However, it is required that P_2 does not learn anything about π_2. Hence, P_1, P_3 sample a random $\mathbf{r}_2 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, and instead send $\mathbf{a}_2 = \pi_2(\beta_{\mathbf{v}} + \mathbf{r}_2)$ to P_2, who can then compute $\pi_3(\pi_1(\mathbf{a}_2)) = \pi'(\beta_{\mathbf{v}}) + \pi'(\mathbf{r}_2)$. In the computation of $\beta_{\mathbf{w}}$, to remove $\pi'(\mathbf{r}_2)$ and account for the missing summands $\pi'(\alpha_{\mathbf{v}}), \alpha_{\mathbf{w}1}$, P_2 must be provided with $\mathbf{b}_2 = \alpha_{\mathbf{w}1} - \pi'(\alpha_{\mathbf{v}}) - \pi'(\mathbf{r}_2)$. Obtaining $\mathbf{b}_2$ allows P_2 to compute $\beta_{\mathbf{w}} = \pi_3(\pi_1(\mathbf{a}_2)) + \mathbf{b}_2 + \alpha_{\mathbf{w}2} + \alpha_{\mathbf{w}3}$. Observe that since $\mathbf{b}_2$ is independent of the input, it can be generated towards P_2 in the preprocessing phase (as discussed later). Thus, generating $\beta_{\mathbf{w}}$ towards P_2 requires communication of a single message $\mathbf{a}_2$ in one round in the online phase.

Generation of $\beta_{\mathbf{w}}$ towards P_1 Similar to above, a simple way of generating $\beta_{\mathbf{w}}$ towards P_1 is to now make P_2, P_3 send $\pi_1(\mathbf{a}_2 + \mathbf{r}_1)$ to P_1, where $\mathbf{r}_1 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ is sampled randomly by P_2, P_3, and serves as a mask to hide π_1 from P_1. Party P_1 can apply π_3 on the received value and add it with analogous preprocessed terms (similar to $\mathbf{b}_2$) as described earlier to obtain $\beta_{\mathbf{w}}$. However, this approach requires an additional round of communication since P_1 requires to wait until P_2 receives $\mathbf{a}_2$ in the prior round. Instead, our approach is to enable P_1 to compute $\beta_{\mathbf{w}}$ in the same round as P_2. For this, in the preprocessing phase, we make available towards P_2, P_3 the permutation $\pi_s \circ \pi_1 \circ \pi_2$ where π_s is a random permutation known to P_1, P_3. Observe that presence of π_s in $\pi_s \circ \pi_1 \circ \pi_2$, ensures that π_2 remains hidden from P_2. Given this permutation, generation of $\beta_{\mathbf{w}}$ towards P_1 proceeds as follows. P_2, P_3 compute and send $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\beta_{\mathbf{v}} + \mathbf{r}_1)))$ via Π_{Jmp} to P_1. Given $\mathbf{b}_1 = \alpha_{\mathbf{w}2} - \pi'(\alpha_{\mathbf{v}}) - \pi'(\mathbf{r}_1)$ is generated towards P_1 in the preprocessing phase (as discussed later), P_1 can compute $\beta_{\mathbf{w}} = \pi_3(\pi_s^{-1}(\mathbf{a}_1)) + \mathbf{b}_1 + \alpha_{\mathbf{w}1} + \alpha_{\mathbf{w}3}$. Note that P_1 can compute π_s^{-1} since it has π_s on clear. In this way, generating $\beta_{\mathbf{w}}$ towards P_1 requires communicating the message $\mathbf{a}_1$ and no additional rounds in the online phase.

Generation of $\beta_{\mathbf{w}}$ towards P_3 Having generated $\beta_{\mathbf{w}}$ towards P_1, P_2, they send it to P_3 via Π_{Jmp}. Note that in scenarios such as GraphSC, which demand several shuffle invocations, sending of $\beta_{\mathbf{w}}$ towards P_3 with respect to all shuffle instances can be performed in a single

round, thereby amortizing the cost of this round across several shuffle instances[3]. In such cases, our shuffle protocol requires only a single round of interaction per shuffle instance in the online phase. A pictorial representation appears in Fig. 4.6, where arrows capture communication via Π_{Jmp}.

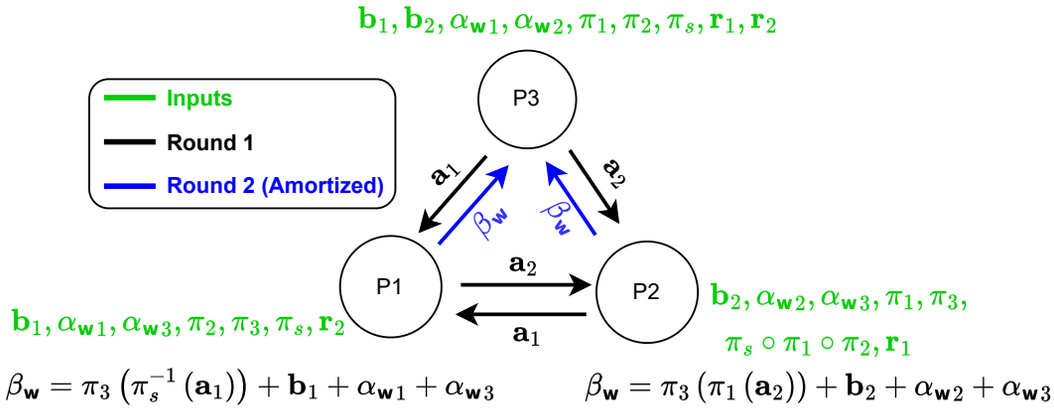


Figure 4.6: Online phase of shuffle protocol for generating $\beta_{\mathbf{w}}$ where $\mathbf{w} = \pi'(\mathbf{v})$ and $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$.

Generation of additional preprocessing data To facilitate the one-round online phase, we now discuss how additional data such as $\pi_s \circ \pi_1 \circ \pi_2$ and the terms $\mathbf{b}_1, \mathbf{b}_2$ can be generated in the preprocessing phase. For P_2, P_3 to generate $\Pi = \pi_s \circ \pi_1 \circ \pi_2$, parties P_0, P_1, P_3 randomly sample a permutation π_s[4]. P_0, P_3 can then compute Π locally since they hold π_1, π_2, and invoke Π_{Jmp} to send Π to P_2.

For generating $\mathbf{b}_1, \mathbf{b}_2$, we extend the 3-party semi-honest shuffle protocol of [13] to work in our 4-party malicious setting. The modified protocol continues to have 2 rounds as in the case of [13], which is possible due to the following observation. The protocol of [13] takes as input $[\cdot]$-shares of a vector $\mathbf{v}$ and outputs $[\cdot]$-shares of $\pi'(\mathbf{v})$. Here, we can view $\pi' = \pi_3 \circ \pi_1 \circ \pi_2$ which is shared among P_1, P_2, P_3 such that parties P_2, P_3 hold π_1, parties P_1, P_3 hold π_2, and parties P_1, P_2 hold π_3. While extending the protocol of [13] in the preprocessing phase of our 4-party setting, we note that P_0 has all the inputs held by P_1, P_2, P_3. This allows P_0 to compute all

[3]Note that with respect to the framework of Tetrad, any communication towards P_3 can be deferred until output reconstruction. We refer to [138] for further details.

[4]Steps for randomly sampling a permutation are discussed in §2.5.1.

protocol messages communicated in [13]. In this way, these messages can be communicated by two senders, one of which is P_0, by invoking Π_{Jmp}. The use of Π_{Jmp} facilitates attaining malicious security.

We next describe our concrete protocol steps for generating $\mathbf{b}_2 = \alpha_{\mathbf{w}1} - \pi'(\alpha_{\mathbf{v}} - \mathbf{r}_2)$ towards P_2. Observe that using the protocol of [13] allows generating $[\cdot]$-shares of $\pi'(\alpha_{\mathbf{v}} - \mathbf{r}_2)$. However, the requirement is to generate $\alpha_{\mathbf{w}1} - \pi'(\alpha_{\mathbf{v}} - \mathbf{r}_2)$ on clear towards P_2. Hence, we slightly modify the protocol steps to instead generate this value on clear towards P_2, which entails providing P_2 its missing $[\cdot]$-share of $\pi'(\alpha_{\mathbf{v}} - \mathbf{r}_2)$ which is masked with $\alpha_{\mathbf{w}1}$. Since the protocol takes as input $[\cdot]$-shares of $\alpha_{\mathbf{v}} - \mathbf{r}_2$, we first discuss how this is generated. Let $\boldsymbol{\gamma}_2 = \alpha_{\mathbf{v}} - \mathbf{r}_2$, where $\mathbf{r}_2 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ is sampled randomly by P_0, P_1, P_3. Parties non-interactively generate $[\cdot]$-shares of $\mathbf{r}_2$ by P_0, P_1, P_3 setting their common share as $\mathbf{r}_2$, and the other shares being set as 0. Parties then generate $[\cdot]$-shares of $\boldsymbol{\gamma}_2$ using $[\cdot]$-shares of $\alpha_{\mathbf{v}}$ and $\mathbf{r}_2$, and the linearity property of $[\cdot]$-sharing. Given $[\cdot]$-shares of $\boldsymbol{\gamma}_2$, the high-level overview of the protocol is as follows. Parties non-interactively sample $\mathbf{z}_i \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ for $i \in \{1,2,3\}$. Specifically, P_0, P_1, P_3 sample $\mathbf{z}_2$, P_0, P_2, P_3 sample $\mathbf{z}_1$, while P_0, P_1, P_2 sample $\mathbf{z}_3$. The $\mathbf{z}_i$'s serve as a random mask to ensure that parties only see random values throughout the protocol execution. Parties compute messages $\mathbf{x}_i, \mathbf{y}_i$ for $i \in \{1,2,3\}$ such that the following invariant is maintained: each $\mathbf{x}_i + \mathbf{y}_i$ is always a shuffle of $\boldsymbol{\gamma}_2$. The protocol proceeds in rounds as described in Fig. 4.7. Note that all the messages that are communicated can be computed by P_0, and hence are sent via Π_{Jmp}, which ensures security against a malicious adversary. This communication from P_0 is omitted in the figure. Correctness of $\mathbf{b}_2$ follows by opening up the corresponding values of $\mathbf{x}_i, \mathbf{y}_i$ for $i \in \{1,2,3\}$. Observe that generating $\mathbf{b}_2$ towards P_2 requires communicating 3 messages in 2 rounds.

The protocol proceeds analogously for the generation of $\mathbf{b}_1$ towards P_1, except that the computation happens with $\boldsymbol{\gamma}_1 = \alpha_{\mathbf{v}} - \mathbf{r}_1$, and in the second round P_2 sends $\mathbf{y}_3 + \alpha_{\mathbf{w}2}$ to P_1. Note that fresh random values $\mathbf{z}_i \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ for $i \in \{1,2,3\}$ are sampled to carry out this execution which is independent of the values used while generating $\mathbf{b}_2$ towards P_2. The protocol appears in Fig. 4.8. Note that the generation of $\mathbf{b}_1$ towards P_1 can be performed in parallel to generating $\mathbf{b}_2$ towards P_2, and does not incur any additional rounds.

The schematic representation of the various parts involved in the generation of $\mathbf{u}$ is provided in Fig. 4.9. The formal protocol is provided next.

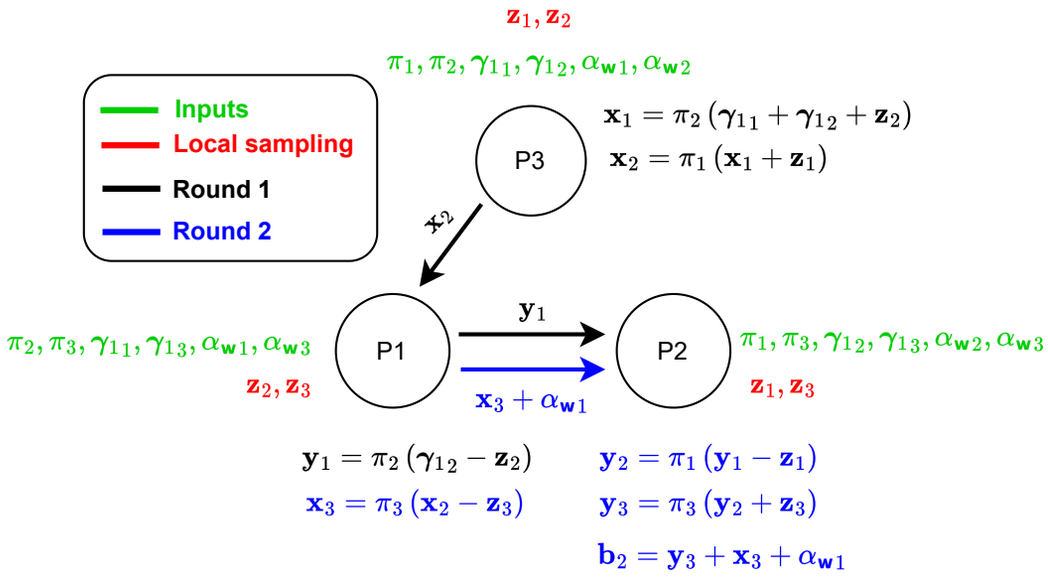


Figure 4.7: Generating $\mathbf{b}_2 = \alpha_{\mathbf{w}1} - \pi'(\alpha_{\mathbf{v}} - \mathbf{r}_2)$ towards P_2.

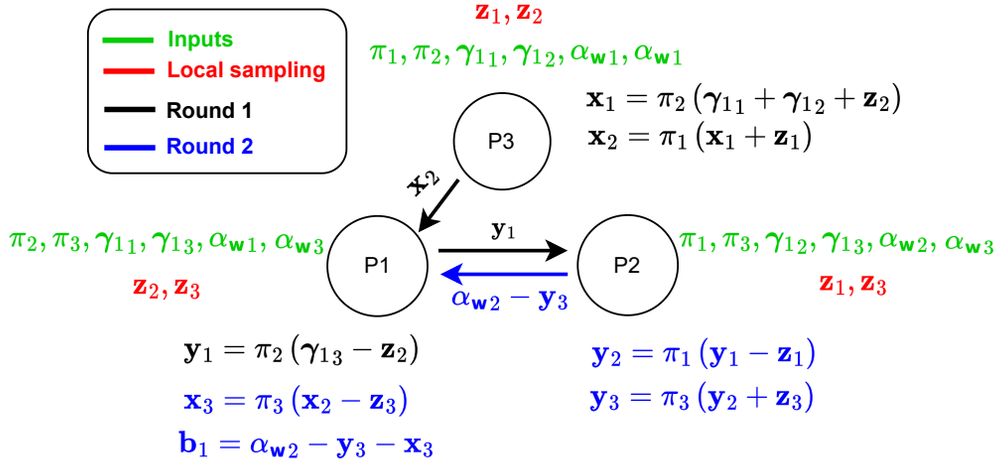


Figure 4.8: Generating $\mathbf{b}_1 = \alpha_{\mathbf{w}2} - \pi'(\alpha_{\mathbf{v}} - \mathbf{r}_1)$ towards P_1.

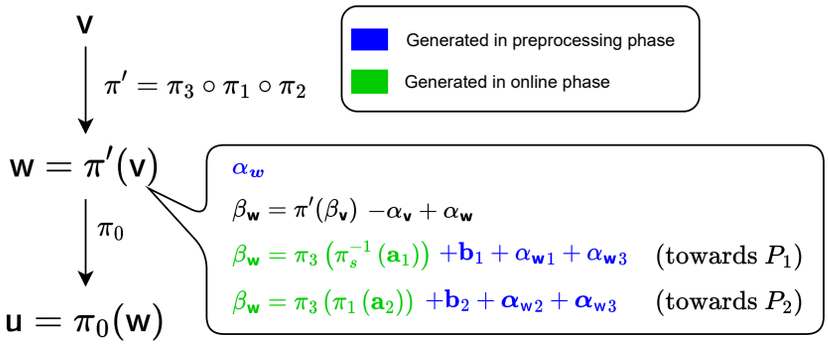


Figure 4.9: Overview of steps performed in secure shuffle.

The complete shuffle protocol Ideal functionality for secure shuffle appears in Fig. 4.10.

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

Without loss of generality, let $P_c \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Shuffle}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]$-shares of the input $\mathbf{v}$ from all parties. Let $\mathbf{u}$ denote the randomly shuffled input. $\mathcal{F}_{\mathsf{Shuffle}}$ also receives from $\mathcal{S}$ its $[\![\cdot]\!]$-shares of $\mathbf{u}$.
$\mathcal{F}_{\mathsf{Shuffle}}$ proceeds as follows.

- Reconstruct input $\mathbf{v}$ using $[\![\cdot]\!]$-shares of the honest parties.
- Sample a random permutation π from the space of all permutations and generate $\mathbf{u} = \pi(\mathbf{v})$.
- Generate $[\![\cdot]\!]$-shares of $\mathbf{u}$ while accounting for shares received from $\mathcal{S}$. Let $[\![\mathbf{u}]\!]_x$ denotes the shares held by $P_x \in \mathcal{P}$. Send $(\mathsf{Output}, [\![\mathbf{u}]\!]_x)$ to P_x.

Figure 4.10: Ideal functionality for shuffle.

The secure protocol for the online phase of shuffle appears in Fig. 4.11, while the protocol for the preprocessing phase appears in Fig. 4.12.

Protocol Π_{Shuffle}

Online

- P_1, P_3 compute and send $\mathbf{a}_2 = \pi_2(\beta_{\mathbf{v}} + \mathbf{r}_2)$ to P_2 via Π_{Jmp}. In parallel, P_2, P_3 compute and send $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\beta_{\mathbf{v}} + \mathbf{r}_1)))$ via Π_{Jmp} to P_1.
- P_2 computes $\beta_{\mathbf{w}} = \pi_3(\pi_1(\mathbf{a}_2)) + \mathbf{b}_2 + \alpha_{\mathbf{w}2} + \alpha_{\mathbf{w}3}$. P_1 computes $\beta_{\mathbf{w}} = \pi_3\left(\pi_s^{-1}(\mathbf{a}_1)\right) + \mathbf{b}_1 + \alpha_{\mathbf{w}1} + \alpha_{\mathbf{w}3}$.
- P_1, P_2 send $\beta_{\mathbf{w}}$ to P_3 via Π_{Jmp}.
- Parties non-interactively generate $[\![\cdot]\!]$-shares of $\beta_{\mathbf{w}}$.
- Compute $[\![\mathbf{u}]\!] = [\![\beta_{\mathbf{w}}]\!] - [\![\pi_0(\alpha_{\mathbf{w}1})]\!] - [\![\pi_0(\alpha_{\mathbf{w}2})]\!] - [\![\pi_0(\alpha_{\mathbf{w}3})]\!]$.

Figure 4.11: Online phase of secure shuffle protocol.

Protocol Π_{Shuffle}

Preprocessing

– P_0, P_1, P_3 sample π_2; P_0, P_2, P_3 sample π_2; P_0, P_1, P_2 sample π_3; and P_1, P_2, P_3 sample π_0, non-interactively, and define $\pi = \pi_0 \circ \pi_3 \circ \pi_1 \circ \pi_2$.

– P_0, P_1, P_3 sample $\alpha_{\mathbf{w}1} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; P_0, P_2, P_3 sample $\alpha_{\mathbf{w}2} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; and P_0, P_1, P_2 sample $\alpha_{\mathbf{w}3} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, non-interactively.

– P_i, P_3 for $i \in \{1, 2\}$ invoke Π_{JSh} to generate $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}i})$. Similarly, P_1, P_2 invoke Π_{JSh} to generate $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}3})$.

//Generation of Π *towards* P_2, P_3

– P_0, P_1, P_3 randomly sample a permutation π_s.

– P_0, P_3 compute $\Pi = \pi_s \circ \pi_1 \circ \pi_2$ locally and invoke Π_{Jmp} to send Π to P_2.

//Generation of $\mathbf{b}_1$ *towards* P_1

– P_0, P_2, P_3 non-interactively sample $\mathbf{r}_1 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$.

– Parties non-interactively generate $[\mathbf{r}_1]$, and set $[\boldsymbol{\gamma}_1] = [\alpha_{\mathbf{v}}] - [\mathbf{r}_1]$.

– P_0, P_1, P_3 sample $\mathbf{z}_{21} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; P_0, P_2, P_3 sample $\mathbf{z}_{11} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; and P_0, P_1, P_2 sample $\mathbf{z}_{31} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, non-interactively.

– P_0, P_3 compute $\mathbf{x}_{11} = \pi_2(\boldsymbol{\gamma}_{11} + \boldsymbol{\gamma}_{12} + \mathbf{z}_{21})$, and $\mathbf{x}_{21} = \pi_1(\mathbf{x}_{11} + \mathbf{z}_{11})$, where $\boldsymbol{\gamma}_{11}, \boldsymbol{\gamma}_{12}$ denote two of the three $[\cdot]$-shares of $\boldsymbol{\gamma}_1$. In parallel, P_0, P_1 compute $\mathbf{y}_{11} = \pi_2(\boldsymbol{\gamma}_{13} - \mathbf{z}_{21})$.

– P_0, P_3 invoke Π_{Jmp} to send $\mathbf{x}_{21}$ to P_1, while P_0, P_1 invoke Π_{Jmp} to send $\mathbf{y}_{11}$ to P_2.

– P_0, P_2 compute $\mathbf{y}_{21} = \pi_1(\mathbf{y}_{11} - \mathbf{z}_{11})$, and $\mathbf{y}_{31} = \pi_3(\mathbf{y}_{21} + \mathbf{z}_{31})$.

– P_0, P_2 invoke Π_{Jmp} to send $\alpha_{\mathbf{w}2} - \mathbf{y}_{31}$ to P_1.

– P_1 computes $\mathbf{x}_{31} = \pi_2(\mathbf{x}_{21} - \mathbf{z}_{31})$, and sets $\mathbf{b}_1 = \alpha_{\mathbf{w}2} - \mathbf{y}_{31} - \mathbf{x}_{31}$.

//Generation of $\mathbf{b}_2$ *towards* P_2

– P_0, P_1, P_3 non-interactively sample $\mathbf{r}_2 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$.

– Parties non-interactively generate $[\mathbf{r}_2]$, and set $[\boldsymbol{\gamma}_2] = [\alpha_{\mathbf{v}}] - [\mathbf{r}_2]$.

– P_0, P_1, P_3 sample $\mathbf{z}_{22} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; P_0, P_2, P_3 sample $\mathbf{z}_{12} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$; and P_0, P_1, P_2 sample $\mathbf{z}_{32} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, non-interactively.

– P_0, P_3 compute $\mathbf{x}_{12} = \pi_2(\boldsymbol{\gamma}_{21} + \boldsymbol{\gamma}_{22} + \mathbf{z}_{22})$, and $\mathbf{x}_{22} = \pi_1(\mathbf{x}_{12} + \mathbf{z}_{12})$, where $\boldsymbol{\gamma}_{21}, \boldsymbol{\gamma}_{22}$ denote two of the three $[\cdot]$-shares of $\boldsymbol{\gamma}_2$. In parallel, P_0, P_1 compute $\mathbf{y}_{12} = \pi_2(\boldsymbol{\gamma}_{23} - \mathbf{z}_{22})$.

– P_0, P_3 invoke Π_{Jmp} to send $\mathbf{x}_{22}$ to P_1, while P_0, P_1 invoke Π_{Jmp} to send $\mathbf{y}_{12}$ to P_2.

– P_0, P_1 compute $\mathbf{x}_{32} = \pi_3(\mathbf{x}_{22} - \mathbf{z}_{32})$, and invoke Π_{Jmp} to send $\alpha_{\mathbf{w}1} - \mathbf{x}_{32}$ to P_2.

– P_2 computes $\mathbf{y}_{22} = \pi_1(\mathbf{y}_{12} - \mathbf{z}_{12})$, $\mathbf{y}_{32} = \pi_3(\mathbf{y}_{22} + \mathbf{z}_{32})$, and sets $\mathbf{b}_2 = \alpha_{\mathbf{w}1} - \mathbf{x}_{32} - \mathbf{y}_{32}$.

Figure 4.12: Preprocessing phase of secure shuffle protocol.

Security of shuffle π remains hidden from all parties since each party misses one component of π. To ensure correct execution of the computation, observe that each message to be communicated is held by two parties, and is sent via Π_{Jmp} (or its $[\![\cdot]\!]$-shares generated via Π_{JSh}). Further, all the protocol messages are masked ($\mathbf{r}_i$ in online and $\mathbf{z}_i$ in preprocessing), guaranteeing no information leakage. The formal security proof appears in §4.8.

Communication and round complexity Observe that the online phase involves sending a message of $\mathsf{N}\ell$ bits towards P_1, P_2 via Π_{Jmp} in a single round of interaction. Since P_1, P_2 hold the required shares to evaluate the function under consideration, the Π_{Jmp} towards P_3 can be deferred. Thus, in applications such as GraphSC that entail multiple shuffle invocations, the Π_{Jmp} execution towards P_3 for the multiple shuffle instances can be performed in a single round, where each instance requires communication $\mathsf{N}\ell$ bits. In this way, the amortized online round complexity of our shuffle protocol is 1 round and has a communication of $3\mathsf{N}\ell$ bits. In the preprocessing phase, observe that generation of $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}1}), \pi_0(\alpha_{\mathbf{w}2}), \pi_0(\alpha_{\mathbf{w}3})$ requires a total communication of $3\mathsf{N}\ell$ bits. Further, generating $\mathbf{b}_1, \mathbf{b}_2$ entails a total communication of $6\mathsf{N}\ell$ bits. Finally, sending $\pi_s \circ \pi_1 \circ \pi_2$ towards P_2 requires communicating $\mathsf{N}\ell$ bits. Thus, the total communication cost in the preprocessing phase is $10\mathsf{N}\ell$ bits.

4.6.2 Scatter and gather primitives for GCN evaluation

For better readability, we provide the definitions of Scatter-Gather primitives in cleartext, while their secure versions can be obtained using the secure protocols for the operations therein.

Recall that the GCN computation in the l^{th} layer during the *forward pass* is given as $\mathbf{H}^{(l)} = g^{(l)}\left(\hat{\mathbf{A}}\mathbf{H}^{(l-1)}\mathbf{W}^{(l-1)}\right)$, where $g^{(l)}(\cdot)$ denotes the activation function. $\mathbf{H}^{(0)}$ is initialized to $\mathbf{X}$, and the final output $\mathbf{Z} = \mathbf{H}^{(2)}$. Although, our goal is to compute $\mathbf{H}^{(l)}$ via Scatter-Gather, we will discuss how $\mathbf{H}^{(1)} = g^{(1)}\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}\right)$ can be computed and define operations within Scatter-Gather for the same. Computation of $\mathbf{H}^{(2)}$ also proceeds analogously. Hence, will omit the superscript in $\mathbf{H}^{(1)}$ and $\mathbf{W}^{(0)}$ for ease of presentation.

We begin by describing the data components that are required to be stored at each entry $\mathsf{G}[i]$ in the DAG list G (all in secret shares) to facilitate the computation. For a vertex entry, $\mathsf{G}[i].\mathsf{deg}$ stores the degree of the vertex, which also accounts for the self-loop (as defined in $\tilde{\mathbf{D}}$) whereas $\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}$ stores its inverse square root. The i^{th} row of $\mathbf{X}$ (represented as $\boldsymbol{x}_i$), which denotes the feature vector associated with the i^{th} vertex, is stored at $\mathsf{G}[i].\boldsymbol{x}$. Note that these components are 0 if $\mathsf{G}[i]$ represents an edge. Additionally, vectors $\mathsf{G}[i].\mathbf{dt}, \mathsf{G}[i].\mathbf{agg}$ are used to store intermediate results. We assume that $\mathbf{W}$ is accessible to all nodes in the graph. Given this

information, the goal of Scatter-Gather is to compute the i^{th} row $\boldsymbol{h}_i$ of matrix $\mathbf{H} = g\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}\right)$ and store it at vertex entry $\mathsf{G}[i].\boldsymbol{h}$. The j^{th} component of $\boldsymbol{h}_i$ can be computed as

$$\begin{aligned}\boldsymbol{h}_i[j] = \mathbf{H}_{ij} &= g\left(\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}\right)_{ij}\right) = g\left(\sum_{k=1}^{n} \hat{\mathbf{A}}_{ik} \left(\mathbf{X}\mathbf{W}\right)_{kj}\right) \\ &= g\left(\sum_{k=1}^{n} \frac{\tilde{\mathbf{A}}_{ik}}{\mathsf{deg}_i^{\frac{1}{2}} \cdot \mathsf{deg}_k^{\frac{1}{2}}} \left(\mathbf{X}\mathbf{W}\right)_{kj}\right) \end{aligned} \tag{4.7}$$

The matrix operations in Equation (4.7) for computing $\boldsymbol{h}_i[j]$ when performed via GraphSC would involve aggregating (across the various k's) the j^{th} component of $(\boldsymbol{x}_k \cdot \mathbf{W})$, scaling the aggregated value by the degree terms, followed by application of $g(\cdot)$ on the same. Observe that this aggregation accounts only for the neighbours of node i since $\tilde{\mathbf{A}}_{ik} = 0$ otherwise. Thus, each node k can compute $\dfrac{\boldsymbol{x}_k \cdot \mathbf{W}}{\mathsf{deg}_k^{\frac{1}{2}}}$ and *scatter* it across its edges. The node i can then *gather* these vectors from its neighbors, scale it using $\mathsf{deg}_i^{-\frac{1}{2}}$, and apply $g(\cdot)$ on this vector to generate $\boldsymbol{h}_i$. In this way, one invocation of Scatter and Gather results in populating $\mathsf{G}[i].\boldsymbol{h}$ and accomplishes the computation of $\mathbf{H}^{(1)} = g^{(1)}\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}\right)$ in a vertex-centric manner. Similarly, $\mathbf{Z} = \mathbf{H}^{(2)}$ can also be computed. The formal protocols for Scatter-Gather appear in Fig. 4.13, whose secure variant can be obtained as described in §4.6.2.1. We remark that although the definitions of Scatter-Gather have a linear complexity in $|\mathsf{V}| + |\mathsf{E}|$, their sub-linear variant can be obtained using the technique of [179], recalled in §2.6.1.

Scatter(G)

for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do:
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\mathbf{v} = (\mathsf{G}[i].\boldsymbol{x}) \cdot \mathbf{W} \cdot (\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}})$
  else
    $\mathsf{G}[\mathrm{i}].\mathbf{dt} = \mathbf{v}$

Gather(G)

for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do:
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\mathsf{G}[i].\boldsymbol{h} = g\left((\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}) \cdot \mathbf{agg}\right)$
    $\mathbf{agg} = \mathbf{0}$
  else
    $\mathbf{agg} = \mathbf{agg} + \mathsf{G}[i].\mathbf{dt}$

Figure 4.13: Scatter and Gather to compute $\mathbf{H}^{(1)} = g^{(1)}\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}\right)$ in forward pass.

Recall that in the *backward pass*, the derivative of the cross-entropy loss with respect to the weight matrices is computed using the output of the forward pass, $\mathbf{Z}$, as well as the target result $\mathbf{Y}$ for the training data. This is then used to update the weight matrices $\mathbf{W}^{(0)}, \mathbf{W}^{(1)}$ via the

Adam optimizer. We will now showcase how the computation of the derivatives (Equation (4.8), Equation (4.9)) can be performed efficiently via GraphSC primitives of Scatter and Gather.

$$\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}} = {\mathbf{H}^{(1)}}^{\mathsf{T}} \hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right) \tag{4.8}$$

$$\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(1)}} = \left(\hat{\mathbf{A}} \mathbf{X} \right)^{\mathsf{T}} \left(\mathsf{dReLU}(\mathbf{In}) \bigodot \hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right) {\mathbf{W}^{(1)}}^{\mathsf{T}} \right) \tag{4.9}$$

Here, $\mathbf{In} = \hat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}$, $\mathbf{M}^{\mathsf{T}}$ denotes the matrix transpose operation, and $\bigodot$ is the element-wise multiplication operator. As in the case of the forward pass, we begin by describing the data components associated with $\mathsf{G}[i]$. Recall that as part of the forward pass, the i^{th} row of $\mathbf{Z} = \mathbf{H}^{(2)}$, denoted as $\boldsymbol{z}_i$ is already computed. Let $\mathsf{G}[i].\boldsymbol{z}$ denote this component. Similarly, the i^{th} row of $\mathbf{H}^{(1)}$ as well as $\mathbf{In}$ can be made available via the computations performed in the forward pass. In addition to data components that were a part of forward pass, the i^{th} row of $\mathbf{Y}$ that corresponds to the label for i^{th} node, can be stored as a data component, $\mathsf{G}[i].\boldsymbol{y}$.

Computing $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}}$ Note that the computation of $\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right)$ in Equation (4.8) can be performed via GraphSC, similar to the forward pass computation. Elaborately, the i^{th} vertex computes and scatters $\mathsf{v} = \left(\mathsf{G}[i].\boldsymbol{z} - \mathsf{G}[i].\boldsymbol{y} \right) \mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}$ over its edges. All the v components scattered by the neighbours of a node j are then gathered in the node while also accounting for the scaling factor of $\mathsf{G}[j].\mathsf{deg}^{-\frac{1}{2}}$ to generate the j^{th} row of $\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right)$ stored in $\mathsf{G}[j].\boldsymbol{v_1}$. The Scatter-Gather primitives for computing $\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right)$ appear in Fig. 4.14. However, computation of ${\mathbf{H}^{(1)}}^{\mathsf{T}} \left(\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right) \right)$ does not render itself well in the message-passing paradigm. This is because the multiplications performed while computing ${\mathbf{H}^{(1)}}^{\mathsf{T}} \left(\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right) \right)$ are independent of the structure of the graph, and no longer require the neighbourhood information (which is otherwise leveraged while defining the Scatter-Gather primitives). Thus, to compute ${\mathbf{H}^{(1)}}^{\mathsf{T}} \left(\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right) \right)$, we extract the matrices $\mathbf{H}^{(1)}$ and $\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right)$ from the list representation, followed by performing matrix multiplication. For this, we proceed as follows: (i) sort G such that vertex entries appear first, followed by edge entries, and (ii) extract the first $|\mathsf{V}|$ entries of $\mathsf{G}[i].\boldsymbol{h}, \mathsf{G}[i].\boldsymbol{v_1}$ to generate $\mathbf{H}^{(1)}, \hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right)$, respectively. Subsequently, we compute ${\mathbf{H}^{(1)}}^{\mathsf{T}}$ followed by multiplication with $\hat{\mathbf{A}} \left(\mathbf{Z} - \mathbf{Y} \right)$ to generate $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}}$.

Scatter(G)

for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do:
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\boldsymbol{v_1} = \mathsf{G}[i].\boldsymbol{z} - \mathsf{G}[i].\boldsymbol{y}$
    $\boldsymbol{v_2} = \mathsf{G}[i].\boldsymbol{x}$
    $\boldsymbol{v_3} = (\mathsf{G}[i].\boldsymbol{z} - \mathsf{G}[i].\boldsymbol{y}) \cdot \mathbf{W}^{(1)^\mathsf{T}}$
  else
    $\mathsf{G}[\mathrm{i}].\boldsymbol{v_1} = \boldsymbol{v_1}$
    $\mathsf{G}[\mathrm{i}].\boldsymbol{v_2} = \boldsymbol{v_2}$
    $\mathsf{G}[\mathrm{i}].\boldsymbol{v_3} = \boldsymbol{v_3}$

Gather(G)

$\mathbf{agg} = \mathbf{0}$
for $i = 1$ to $|\mathsf{V}| + |\mathsf{E}|$ do:
  if $\mathsf{G}[i].\mathsf{isV}$ then
    $\mathsf{G}[i].\boldsymbol{v_1} = \left(\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}\right) \mathbf{agg_1}$
    $\mathsf{G}[i].\boldsymbol{v_2} = \left(\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}\right) \mathbf{agg_2}$
    $\mathsf{G}[i].\boldsymbol{v_3} = \mathsf{dReLU}\left(\mathsf{G}[i].\mathbf{In}\right) \bigodot \left(\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}\right) \mathbf{agg_3}$
    for $j = 1$ to 3 do:
      $\mathbf{agg}_j = 0$
  else
    for $j = 1$ to 3 do:
      $\mathbf{agg}_j = \mathbf{agg}_j + \mathsf{G}[\mathrm{i}].\boldsymbol{v}_j$

Figure 4.14: Scatter and Gather to compute $\hat{\mathbf{A}}\left(\mathbf{Z} - \mathbf{Y}\right)$, $\hat{\mathbf{A}}\mathbf{X}$, and $\mathsf{dReLU}\left(\mathbf{In}\right) \bigodot \hat{\mathbf{A}}\left(\mathbf{Z} - \mathbf{Y}\right)$ in backward pass.

Computing $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(1)}}$ With respect to the computation of Equation (4.9), we note that $\hat{\mathbf{A}}\mathbf{X}$ can be computed via GraphSC primitives. For this vertex i scatters $\mathsf{G}[i].\boldsymbol{x}$ which is gathered in data component $\boldsymbol{v_2}$ of G. Similarly, $\hat{\mathbf{A}}\left(\mathbf{Z} - \mathbf{Y}\right)\mathbf{W}^{(1)^\mathsf{T}}$ can be computed via GraphSC by scattering $\mathbf{v} = (\mathsf{G}[i].\boldsymbol{z} - \mathsf{G}[i].\boldsymbol{y}) \cdot \mathbf{W}^{(1)^\mathsf{T}}$ followed by gathering it in data component $\boldsymbol{v_3}$ of G. Moreover, since Equation (4.9) requires computation of $\mathsf{dReLU}(\mathbf{In}) \bigodot \hat{\mathbf{A}}\left(\mathbf{Z} - \mathbf{Y}\right)\mathbf{W}^{(1)^\mathsf{T}}$, after gathering $\mathbf{v}$ from the neighbors, its element-wise multiplication with $\mathsf{dReLU}\left(\mathsf{G}[i].\mathbf{In}\right)$ can be performed because the i^{th} row of $\mathbf{In}$ is held with the i^{th} vertex in G. The formal details of Scatter-Gather appear in Fig. 4.14. Next, to multiply $\left(\hat{\mathbf{A}}\mathbf{X}\right)^\mathsf{T}$ with $\mathsf{dReLU}(\mathbf{In}) \bigodot \hat{\mathbf{A}}\left(\mathbf{Z} - \mathbf{Y}\right)\mathbf{W}^{(1)^\mathsf{T}}$, we proceed as done in the previous case, where we extract the matrices from their list representations, followed by matrix multiplication. This computation is performed outside GraphSC due to similar reasons as provided for the case of $\frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(0)}}$.

4.6.2.1 Securely realizing Scatter-Gather

The secure variants of the Scatter-Gather primitives can be obtained using the secure protocols for the operations therein. We note that the primitives are defined such that the number of iterations in each looping construct is dependent on a publicly known value ($|\mathsf{V}| + |\mathsf{E}|$ in this

case). Further, all the private values, such as the entries in G and other intermediate variables used in Scatter-Gather, are operated on as secret shares. The primitives additionally have branching statements such as the *if-else* construct. We rely on the Π_{Sel} primitive (see Table 2.2) to obliviously evaluate only those steps within the correct branch of the construct. More specifically, every assignment operation within both branches is evaluated via Π_{Sel}. However, based on the branch condition provided as input to Π_{Sel}, only those assignment statements where the condition is met are updated, while the others will remain unchanged.

4.6.3 Generation of shares of G

Given $[\![\cdot]\!]$-shares of $\mathbf{A}, \mathbf{X}$ and $\mathbf{Y}$, generating G, entails generating $[\![\cdot]\!]$-shares of (i) $\mathsf{G}[i].\mathsf{isV}$ to denote if the i^{th} tuple is a vertex or an edge, (ii) $\mathsf{G}[i].\mathsf{deg}$ and $\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}$ to store the degree and inverse degree, and (iii) $\mathsf{G}[i].\mathbf{dt}$ to store the data elements, some of which comprise a row of the feature matrix, a row of $\mathbf{Y}$, intermediate results, etc. For this, we proceed as follows. When the i^{th} entry is a vertex, we set $[\![\mathsf{G}[i].\mathsf{isV}]\!] = [\![1]\!]$, $[\![\mathsf{G}[i].\boldsymbol{x}]\!]$ as the i^{th} row of $\mathbf{X}$, $[\![\mathsf{G}[i].\mathsf{deg}]\!]$ and $[\![\mathsf{G}[i].\mathsf{deg}^{-\frac{1}{2}}]\!]$ as the $(i,i)^{\text{th}}$ entry of $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{D}}^{-\frac{1}{2}}$, respectively, while remaining $\mathsf{G}[i].\mathbf{dt}$ are initialized to 0 vectors. Since there may only be $|\mathsf{E}|$ edges but $[\![\mathbf{A}]\!]$ consists of $|\mathsf{V}|^2$ possibilities, the challenge arises in generating their corresponding entries in G while leaking no information. For this, we generate a list $[\![\mathsf{G}']\!]$ comprising of all possible edges (i.e. every element in $\mathbf{A}$). We set $[\![\mathsf{G}'[i].\mathsf{isV}]\!] = [\![\mathbf{A}_{ij}]\!]$, and all other data components to $[\![0]\!]$. To extract the valid $|\mathsf{E}|$ edges from $|\mathsf{V}|^2$ entries in G', we sort $[\![\mathsf{G}']\!]$ based on the values in $[\![\mathsf{isV}]\!]$ in descending order, extract the first $|\mathsf{E}|$ entries, and append these to $[\![\mathsf{G}]\!]$. Since, isV should be 1 only for vertices, values of $[\![\mathsf{isV}]\!]$ are set $1 - [\![\mathsf{isV}]\!]$ for edges before appending them to $[\![\mathsf{G}]\!]$.

4.7 Benchmarks

Benchmark environment and parameters Benchmarks are performed over LAN using Google Cloud instances with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors, 64vCPUs, 120GB of RAM memory and a bandwidth of 16Gbps. We implement all protocols in Python. We use the Crypto library for AES and hashlib for generating SHA256 hash. We note that our code is developed for benchmarking, is not optimized for industry-grade use, and a C++-based implementation can give better performance. We consider run time for one epoch as a benchmark parameter for efficiency comparison and report the online and preprocessing cost. However, when reporting accuracy, we consider a maximum of 200 epochs and stop the training earlier if the test loss does not change for 10 consecutive epochs.

4.7.1 Comparison of primitives

The overall performance of Entrada is heavily dependent on the underlying primitives. Hence, we first analyze the same to showcase the efficiency and accuracy improvements brought in by the new primitives in comparison to the ones in the literature. For a fair comparison, all algorithms are realized using the MPC of Tetrad. Since our improvements are in exponentiation and division, which are also brought in by our improved double bit injection and prefix OR protocols in addition to other optimizations described in §7.3.4, we focus on comparing exponentiation with the protocol of [126], and division with GC-based division of Tetrad, in Table 4.3. We also report the relative error in the accuracy of the secure protocols with respect to floating-point operations. We note that for both, exponentiation and division, our efficiency improvements come without compromising accuracy.

Operation	Reference	Communication(KB)		Run time(ms)		Relative Error (%)
		Preprocessing	online	Preprocessing	Online	
Exp	[126]	3.91	0.91	8.21	15.29	0.24
	Entrada	1.33	0.66	5.47	11.90	0.24
Div	Tetrad [138]	11028.61	125.44	1704.09	1665.66	3.72
	Entrada	2.94	1.99	48.57	136.45	3.72

Table 4.3: Comparison of primitives.

4.7.2 GCN

Since the secure computation framework relies on fixed-point arithmetic (FPA), which is known to have lesser accuracy than the floating-point counterpart, we first demonstrate how much is the accuracy loss of the GCN in moving from cleartext floating-point to FPA. Moreover, due to operations such as truncation, and the approximations used within the secure protocols for exponentiation, division, and inverse square root, the accuracy of the secure GCN model may be affected. Hence, we analyze the accuracy of secure GCN (FPA) and demonstrate that it is on par with the cleartext (FPA) variant and that it improves in comparison to Tetrad. We also showcase that Entrada outperforms Tetrad in terms of efficiency. Note that we do not compare against the protocol of [208] since it does not consider the GCN of [133], and only provides support for inference for a relatively old graph neural network. Moreover, [208] operates in a 3PC setting and provides semi-honest security (with only privacy against a malicious adversary), as opposed to our setting where we consider 4PC for efficiency reasons

and attain stronger security notions of fairness/robustness. Finally, we showcase improvements brought in via GraphSC.

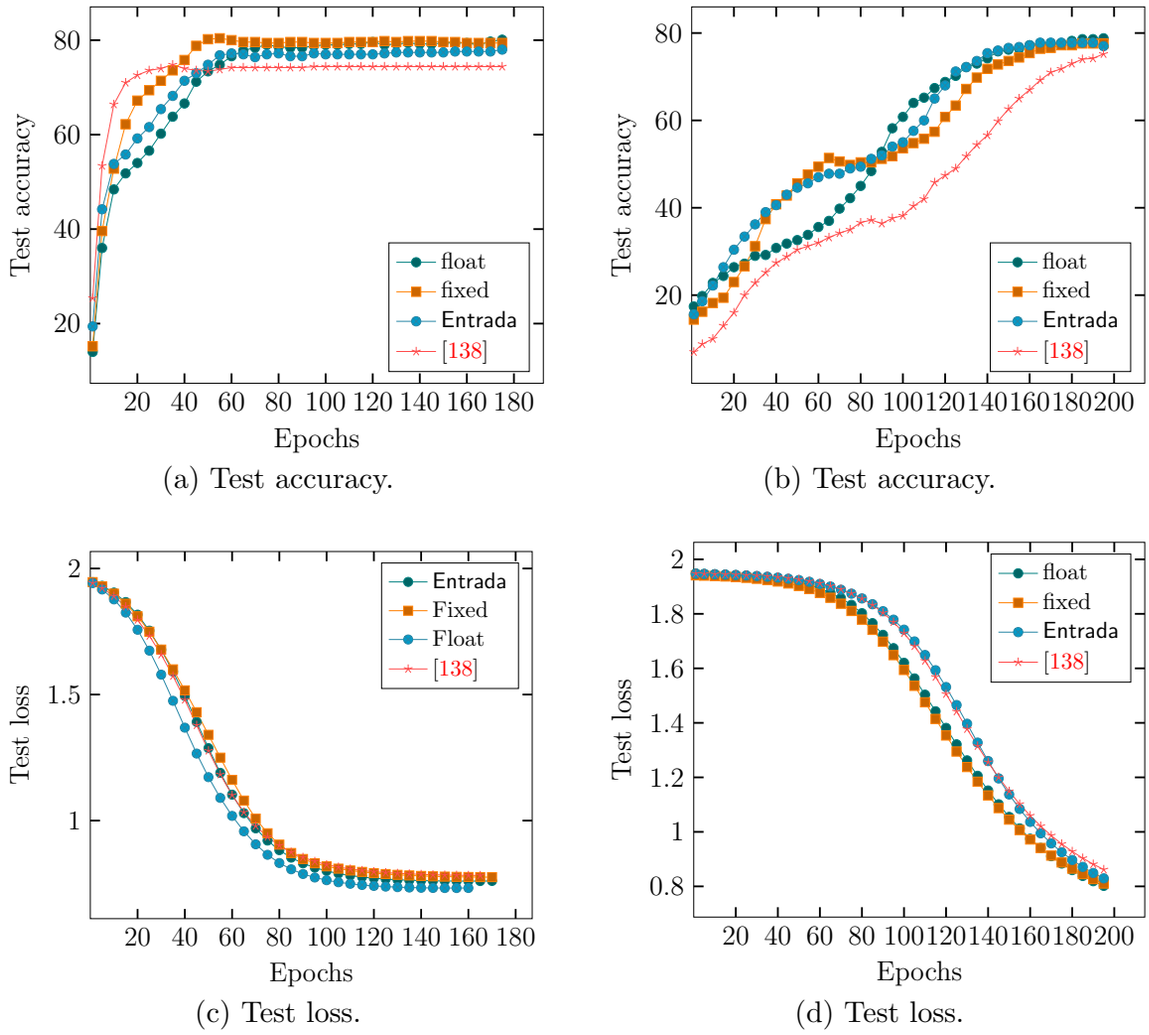
Dataset We use the Cora dataset to benchmark the performance. It contains 2708 documents that are treated as nodes, and 4732 citation links between documents, that are treated as undirected edges. Each document has a bag of words which is treated as the feature vector associated with the node. The documents are considered to be classified into seven classes (or labels).

Accuracy We report the accuracy of GCN using the Cora dataset in Table 4.4. A pictorial representation of variation in accuracy and test loss with the number of epochs appears in Fig. 4.15. As evident from Table 4.4, moving from cleartext floating-point to cleartext FPA representation witnesses a slight drop in accuracy. Keeping the cleartext FPA accuracy as the benchmark for the secure variants, we observe that our protocol loses out on accuracy by only 0.4%. This is very small compared to the loss in accuracy witnessed by Tetrad, which is around 5.6%. On the contrary, realizing secure GCN via SGD in Terad results in an accuracy which is 76.6%. We note that the drop in accuracy in Tetrad while using Adam stems due to the use of the approximate softmax function (ASM), which degrades the accuracy of the overall model.

Model	Cleartext		Secure	
	Float	Fixed	Tetrad (fixed)	Entrada (fixed)
GCN [133]	80.2%	79.7%	74.1%	79.3%

Table 4.4: GCN accuracy on Cora using Adam—Tetrad is enhanced with inverse square root protocol to support Adam.

To showcase improvement brought in by Entrada over Tetrad, Table 4.5 reports impact on accuracy when sequentially replacing each of the following primitives—division, square root, exponentiation—in Tetrad with the newly designed ones. Elaborately, we begin with reporting the accuracy of GCN via Tetrad (version v1), which relies on GC-based division, ASM, and the SGD optimizer. In the next version, v2, we replace the GC-based division with our division protocol. As noted earlier in Table 4.3, our division protocol has the same relative error as the GC-based division of Tetrad and hence does not impact the accuracy of GCN. This is followed by version v3, where ASM in v2 is replaced by the accurate computation of softmax, resulting in improved accuracy. Finally, in version v4, which constitutes our framework Entrada, SGD in v3 is replaced with Adam optimizer, which drastically improves accuracy by over 2%.



(a) Test accuracy. (b) Test accuracy.

(c) Test loss. (d) Test loss.

Figure 4.15: Variation in GCN test accuracy and loss with number of epochs on Cora dataset. float and fixed denote cleartext variants. (a),(c) use Adam and (b),(d) use SGD.

Version	Accuracy
v1:= Tetrad (GC division + ASM + SGD)	76.6%
v2:= v1 - GC division + our division	76.6%
v3:= v2 - ASM + accurate softmax	77.4%
v4:= Entrada (v3 - SGD + Adam optimizer)	79.3%

Table 4.5: GCN accuracy improvements (Cora dataset) when replacing primitives in Tetrad in a step-by-step manner.

Efficiency We compare the efficiency of evaluating GCN via Entrada and Tetrad. We first report the performance of Entrada for GCN inference. Note that the performance of Entrada is on par with Tetrad's, since inference does not require any of the newly designed primitives. Specifically, it has a run time of 2.4 seconds and 5.6 seconds in the preprocessing and online

phase, respectively, when evaluated on the Cora dataset.

With respect to GCN training, recall that Entrada is designed to support the Adam optimizer, and to also leverage the GraphSC paradigm to yield an efficient solution. However, Entrada can be modified to use SGD instead of the Adam optimizer. One can also avoid reliance on GraphSC, depending on the application scenario. Further, recall as discussed in §4.7.2, that Tetrad originally only supports SGD evaluation. Hence, to provide a fair comparison, we also report the performance Entrada when using only the SGD optimizer. In fact, we begin by analyzing Entrada's performance in comparison to Tetrad, while excluding GraphSC. As seen from Table 4.6, Entrada (SGD, w/o GraphSC) outperforms Tetrad in training. The overhead in Tetrad can be mainly attributed to the use of the GC. Entrada (SGD, w/o GraphSC) not only has better efficiency, but also outperforms Tetrad in terms of accuracy (see v3 in Table 4.5). To further improve the accuracy, we switch to the Adam optimizer, which results in Entrada having an increased online time in comparison to Tetrad. This is due to its reliance on additional operations required for supporting Adam. Interestingly, the overall efficiency of Entrada is still better than Tetrad's by a factor of around 30×. The use of GraphSC helps to further improve the efficiency of GCN training, as evident from Table 4.6.

Variant	Preprocessing	Online
Tetrad	7285.076	121.577
Entrada (SGD, w/o GraphSC)	14.988	71.578
Entrada (w/o GraphSC)	30.915	211.160
Entrada (SGD)	1.269	29.601
Entrada	19.572	189.885

Table 4.6: Comparison of GCN performance (training).

GCN evaluation via GraphSC Observe that it is only the training of GCNs that leverages the GraphSC paradigm. One may be misled to believe that performing inference via GraphSC may also lead to improved efficiency. However, the justification for why this is not the case is as follows. Recall that inference is the forward pass sans the Softmax, which comprises matrix multiplications. Hence, relying on the matrix representation allows performing inference in 1 round of interaction. This is because the dot products that are required for matrix multiplication can all be performed in parallel and require just 1 round of computation. On the contrary, this round-efficient approach comes at the cost of prohibitively high memory, as required to store the underlying adjacency matrix. For instance, operating on the Yelp dataset (see §4.7.3) would

require 50GB of memory at a processor when executing the MPC protocol for matrix multiplication. This being the case, leveraging the multiprocessor setting to enhance the efficiency by performing the computations in parallel via m processors would require $m \times 50$GB memory which is prohibitively high. In fact, we observe that for the system configuration specified in the benchmarks, evaluating the GCN under the matrix representation for even two processors runs into insufficient memory issues. Thus, to capitalize on the multiprocessor setting, it is important to have a memory-efficient representation of the underlying graph. Moreover, the representation must also facilitate leveraging the multiprocessor setting. Since the GraphSC paradigm satisfies both the above requirements, we can rely on the same. That is, operating over the list representation of the graph as required by GraphSC only needs 47MB of memory at each processor when executing the MPC protocol. This amounts to a total of 3GB for the 64 vCPUs considered in the current setting. Further, GraphSC provides a way to translate matrix operations into Scatter-Gather operations on the list that are designed to leverage the multiprocessor setting. However, unlike the matrix representation that allows matrix multiplications to be performed in a single round, the GraphSC framework requires $\mathcal{O}(\log(|\mathsf{V}| + |\mathsf{E}|))$ number of rounds. Thus, performing inference via GraphSC would incur additional overhead in comparison to directly operating on the matrices. On the contrary, training witnesses improvements via GraphSC (see Table 4.6, Table 4.10). The reason for the same is as follows. Recall that training comprises invocations of Softmax and the backward pass in addition to the steps of inference. The improvements brought in by the multiprocessor setting of GraphSC in the computation of Softmax significantly overpower the inefficiencies introduced during the inference phase. This is corroborated by the numbers reported in Table 4.6, Table 4.10.

GCN evaluation with 3PC Depending on the application scenario, one may wish to evaluate GCNs in the 3PC setting. Hence, for completeness, we also showcase the practicality of GCN evaluation in the 3PC setting by relying on the state-of-the-art robust framework of [136]. For this, we adapt the primitives designed in this chapter (exponentiation and inverse square root) as well as rely on the primitives from Chapter 3 (prefix OR, division and shuffle) to enhance the 3PC of [136] to support GCN evaluation. The performance is reported in Table 4.7. As expected, observe that Entrada fares better than 3PC.

4.7.3 Fraud detection

We use Entrada to securely realize the application of fraud detection via GCN from the work of [223] and [158], where the former performs fraud detection in online review platforms us-

Variant	Preprocessing (s)	Online (s)
Inference	4.16	7.88
Training w/ Adam w/o GraphSC	57.97	277.04
Training w/ Adam w/ GraphSC	29.60	240.23

Table 4.7: GCN performance in 3PC.

ing the dataset from Tencent App Store, and the latter detects fraudulent accounts in online payment network of Alipay. Given the unavailability of the datasets considered in each of the works, we benchmark their performance on alternative datasets, i.e. [223] is evaluated on the Yelp dataset while [158] is evaluated on the DBLP dataset. As done for the case of vanilla GCN, we first evaluate the accuracy loss, followed by analyzing the performance of the secure protocols. Note that our analysis of these alternative datasets is meant to establish the relative accuracy/performance of Entrada in comparison to the cleartext computation. We believe similar accuracy/performance trends will hold true when Entrada is evaluated on the actual fraud detection datasets. Further, a comparison with Tetrad is omitted since §4.7.2 establishes that we outperform it.

Dataset The work of [223] is evaluated on the Yelp dataset that contains 45,954 reviews, each of which is treated as a node. An edge between two nodes indicates that the corresponding reviews were posted by the same user, and there exist 3,846,979 edges. Each node/review is classified as fake or real. Since the work of [158] operates on a heterogeneous graph, we consider the DBLP dataset that is known to be similar to the original dataset of Alipay since both consider graphs having heterogeneous edges (i.e., multiple edges between the same two nodes may indicate different relations between these nodes). The DBLP dataset consists of 14,328 papers that are treated as nodes. There are three different types of edges, each of which relates two papers (nodes) if they– (i) appear in the same conference, (ii) have the same authors, and (iii) use common terms. There are 1,70,794 edges in total. Further, each node has an associated bag of words that are treated as its feature vector. Each paper/node is classified into 4 different classes (labels) that include database, data mining, machine learning, and information retrieval.

Accuracy and efficiency comparison The results appear in Table 4.8, Table 4.9 and Table 4.10. For accuracy, we observe similar trends as seen in the case of vanilla GCN, where the accuracy of secure variant is comparable to that of cleartext. Regarding efficiency, we

observe up to 4× gain when adapting GCNs to work with GraphSC, thereby corroborating our claim of witnessing efficiency improvements when using GraphSC.

Algorithm	Metric	Cleartext		Secure variant
		Float	Fixed	Entrada (Fixed)
[223]	Recall	0.513	0.507	0.507
	Precision	0.681	0.676	0.669
	F1	0.585	0.579	0.576
[158]	Accuracy	68.5%	67.1%	66.3%

Table 4.8: Accuracy comparison of fraud detection algorithms.

Algorithm	Preprocessing	Online
[223]	34.574	73.678
[158]	21.350	55.511

Table 4.9: Fraud detection algorithms on Entrada (inference).

Algorithm	Preprocessing	Online
[223] (w/o GraphSC)	131.433	686.272
[158] (w/o GraphSC)	73.329	350.468
[223]	32.753	431.398
[158]	32.756	425.707

Table 4.10: Fraud detection algorithms on Entrada (training).

4.8 Security proofs

The simulation-based security proofs for the designed primitives are presented in this section. At a high level, observe that the designed protocols rely on invoking protocols given in Tetrad [138] whose security was established therein in the standard real-world/ideal-world simulation paradigm. Hence, the security of the designed protocols follows directly from the security of the underlying protocols of Tetrad. We let the following denote the ideal functionalities for the protocols provided by Tetrad.

1. $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$: This functionality takes as input $[\![\cdot]\!]$-shares of x, y and outputs $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ by truncated by f bits using probabilistic truncation.

2. $\mathcal{F}_{\mathsf{A2B}}$: This functionality takes as input $[\![\cdot]\!]$-shares of a value x, and outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares for its equivalent Boolean representation.

3. $\mathcal{F}_{\mathsf{B2A}}$: This functionality takes as input $[\![\cdot]\!]^{\mathbf{B}}$-shares of the Boolean representation of a value x, and outputs $[\![\cdot]\!]$-shares for its equivalent arithmetic representation.

4. $\mathcal{F}_{\mathsf{BitInj}}$: This functionality takes as input $[\![\cdot]\!]$-shares of x and $[\![\cdot]\!]^{\mathbf{B}}$-shares of a bit b and outputs $[\![\cdot]\!]$-shares of $\mathsf{b} \cdot \mathsf{x}$.

5. $\mathcal{F}_{\mathsf{Sel}}$: This functionality takes as input $[\![\cdot]\!]$-shares of $\mathsf{x}_0, \mathsf{x}_1$ and $[\![\cdot]\!]^{\mathbf{B}}$-shares of a bit b, and outputs $[\![\cdot]\!]$-shares of x_b.

We use the simulation strategy as described in Tetrad [138], where we simulate the end-to-end computation of a function f for which the designed primitives serve as building blocks. The simulation begins with the simulator $\mathcal{S}$ emulating the shared-key setup $\mathcal{F}_{\mathsf{Setup}}$ functionality (Fig. 2.5) and giving the respective keys to the adversary $\mathcal{A}$. This is followed by the input sharing phase in which $\mathcal{S}$ extracts the input of $\mathcal{A}$, using the known keys, and sets the inputs of the honest parties to be 0 (see simulator for input sharing in [138]). This allows $\mathcal{S}$ to have access to the shares of the honest parties. Since $\mathcal{S}$ knows all the inputs, it can honestly carry out the computation and compute all the intermediate values as required for simulating the view of $\mathcal{A}$. $\mathcal{S}$ proceeds to simulate the various sub-protocols required to compute f in topological order using the aforementioned values. Observe that since $\mathcal{S}$ knows $\mathcal{A}$'s inputs, it can detect any malicious behaviour carried out by $\mathcal{A}$. Finally, depending on $\mathcal{A}$'s behaviour, $\mathcal{S}$ invokes the ideal functionality for the function f with $\mathcal{A}$'s input, obtains the function output and forwards the same to $\mathcal{A}$ during the output reconstruction phase. For simplicity of presentation, we stick to a modular approach of providing simulation steps for each of the (newly designed) sub-protocols, as done in [138]. Note that carrying out these simulation steps in respective topological order (starting from $\mathcal{F}_{\mathsf{Setup}}$, the input sharing phase, all the intermediate sub-protocols, and output reconstruction) results in simulating the computation of the desired function f.

Exponentiation:

The ideal functionalities for exponentiation appears in Fig. 4.16.

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

$\mathcal{F}_{\mathsf{Exp}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

- Receive as input the $[\![\cdot]\!]$-shares of x from all parties.
- Reconstruct x using the shares of honest parties.
- Split x into its integer part t and the fractional part r such that $\mathsf{x} = \mathsf{t} + \mathsf{r}$.
- Compute e^{r} using Taylor series approximation up to θ terms.
- Let $\{\mathsf{t}_i\}_{i=0}^{\mathsf{k}-1}$ denote the bits in $|\mathsf{t}|$ (absolute value of t). Compute the following using probabilistic truncation after multiplication.

$$e^{\mathsf{t}} = \begin{cases} \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{if } \mathsf{x} \geq 0 \\ \prod_{j=\mathsf{f}}^{\mathsf{k}-2} e^{-\mathsf{t}_j \cdot 2^{j-\mathsf{f}}}, & \text{otherwise} \end{cases}$$

- Compute $\mathsf{g} = e^{\mathsf{x}} = e^{\mathsf{t}} \cdot e^{\mathsf{r}}$.
- Generate $[\![\cdot]\!]$-shares of g and send $(\mathsf{Output}, [\![\mathsf{g}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 4.16: Ideal functionality for exponentiation.

Lemma 4.1 (Security) *Protocol Π_{Exp} (Fig. 4.3) securely realizes $\mathcal{F}_{\mathsf{Exp}}$ (Fig. 4.16) in computational 4PC setting against a malicious adversary $\mathcal{S}$ in $(\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{BitInj}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}})$-hybrid model.*

Proof: The simulator for Π_{Exp} appears in Fig. 4.17. The simulator emulates $\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{BitInj}}, \mathcal{F}_{\mathsf{Sel}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from the simulator. This is indistinguishable from its view in the real world, where it sees random values.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{Exp}}$ honestly executes the protocol steps and proceeds as follows.

- Generate $[\![\cdot]\!]$-shares of the integer part of x, denoted as t, on behalf of the honest parties.
- Set $[\![\mathsf{r}]\!] = [\![\mathsf{x}]\!] - [\![\mathsf{t}]\!]$ on behalf of the honest parties.
- Emulate $\mathcal{F}_{\mathsf{A2B}}$ on input $[\![\mathsf{t}]\!]$ and output $[\![\mathsf{t}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- Set $[\![\mathsf{s}]\!]^{\mathbf{B}} = [\![\mathsf{t}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$ on behalf of the honest parties.
- Compute $[\![\mathsf{t}_i]\!]^{\mathbf{B}} = [\![\mathsf{t}_i]\!]^{\mathbf{B}} \oplus [\![\mathsf{s}]\!]^{\mathbf{B}}$ for $i = 0$ to $\mathsf{k}-2$ on behalf of the honest parties.
- for $j = \mathsf{f}$ to $\mathsf{k}-2$ do:
  - Emulate $\mathcal{F}_{\mathsf{BitInj}}$ on inputs $[\![\mathsf{t}_j]\!]^{\mathbf{B}}, e^{2^{j-\mathsf{f}}} - 1$ and output $[\![\mathsf{e}'_j]\!]$ to $\mathcal{A}$.
  - Emulate $\mathcal{F}_{\mathsf{2-bitInj}}$ on inputs $[\![\mathsf{s}]\!]^{\mathbf{B}}, [\![\mathsf{t}_j]\!]^{\mathbf{B}}, e^{-2^{j-\mathsf{f}}} - e^{2^{j-\mathsf{f}}}$ and output $[\![\mathsf{v}_j]\!]$ to $\mathcal{A}$. Compute $[\![\mathsf{e}_j]\!] = [\![\mathsf{e}'_j]\!] + [\![\mathsf{v}_j]\!] + 1$ on behalf of the honest parties.
- Set $[\![\mathsf{d}]\!] = [\![\mathsf{e}_f]\!]$ on behalf of the honest parties.
- For $j = \mathsf{f}+1$ to $\mathsf{k}-2$: emulate $\mathcal{F}_{\mathsf{Mul-Tr}}$ on inputs $[\![\mathsf{d}]\!], [\![\mathsf{e}_j]\!], \mathsf{f}$ and output $[\![\mathsf{d}]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $1, 1/e, [\![\mathsf{s}]\!]^{\mathbf{B}}$ and output $[\![\mathsf{z}]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Mul-Tr}}$ on inputs $[\![\mathsf{d}]\!], [\![\mathsf{z}]\!], \mathsf{f}$ and output $[\![\mathsf{d}]\!]$ to $\mathcal{A}$.
- Set $[\![\mathsf{b}_0]\!] = 1$, $[\![\mathsf{b}_1]\!] = [\![\mathsf{r}]\!]$ on behalf of teh honest parties.
- Emulate $\mathcal{F}_{\mathsf{Mul-Tr}}$ on inputs $[\![\mathsf{b}_{i-1}]\!], [\![\mathsf{b}_1]\!], \mathsf{f}$ and output $[\![\mathsf{b}_i]\!]$ to $\mathcal{A}$ for $i = 2$ to θ.
- Set $[\![\mathsf{b}]\!] = \sum_{i=0}^{\theta} \frac{[\![\mathsf{b}_i]\!]}{i!}$ on behalf of the honest parties.
- Emulate $\mathcal{F}_{\mathsf{Mul-Tr}}$ on inputs $[\![\mathsf{d}]\!], [\![\mathsf{b}]\!], \mathsf{f}$ and output $[\![\mathsf{g}]\!]$ to $\mathcal{A}$.

Figure 4.17: Simulator for Π_{Exp}.

□

Inverse square root:

The ideal functionalities for $\Pi_{\mathsf{PreInvSqrt}}$ and inverse square root appear in Fig. 4.18, Fig. 4.19, respectively.

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

$\mathcal{F}_{\mathsf{PreInvSqrt}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

- Receive as input the $[\![\cdot]\!]$-shares of a from all parties.
- Reconstruct a using the shares of honest parties.
- Compute $\mathsf{a}' = \mathsf{a} \cdot \mathsf{v}$, which is a normalized to lie in $(0.25, -.5]$ in fixed-point arithmetic representation as follows using probabilistic truncation when performing multiplication.
  - Compute the scaling factor $\mathsf{v} = 2^{-(e+1)}$, where the most significant non-zero bit of a appears at index $e + \mathsf{f}$ in the bit representation of a, (with v having f bit precision).
  - Compute $\mathsf{a}' = \mathsf{a} \cdot \mathsf{v}$.
  - Set $\mathsf{v}' = 2^{-(e+1)/2}$.
- Generate $[\![\cdot]\!]$-shares of a', v' and send $(\mathsf{Output}, [\![\mathsf{a}']\!]_s, [\![\mathsf{v}']\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 4.18: Ideal functionality for $\Pi_{\mathsf{PreInvSqrt}}$.

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

$\mathcal{F}_{\mathsf{InvSqrt}}$ interacts with parties in $\mathcal{P}$ and ideal world adversary $\mathcal{S}$, and proceeds as follows.

- Receive as input the $[\![\cdot]\!]$-shares of a from all parties.
- Reconstruct a using the shares of honest parties.
- Normalize a to $\mathsf{a}' \in (0.25, 0.5]$ and compute square root of the scaling factor v' as described in Fig. 4.18.
- Compute $\mathsf{y}' = 4.63887\mathsf{a}'^2 - 5.77789\mathsf{a}' + 3.14736$ and $\mathsf{y} = \mathsf{y}' \cdot \mathsf{v}'$ using probabilistic truncation after multiplication.
- Generate $[\![\cdot]\!]$-shares of y and send $(\mathsf{Output}, [\![\mathsf{y}]\!]_s)$ to $P_s \in \mathcal{P}$.

Figure 4.19: Ideal functionality for inverse square root.

Lemma 4.2 (Security) *Protocol $\Pi_{\mathsf{PreInvSqrt}}$ (Fig. 4.4) securely realizes $\mathcal{F}_{\mathsf{PreInvSqrt}}$ (Fig. 4.18) in the computational 4PC setting against a malicious adversary $\mathcal{S}$ in the $(\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}})$-hybrid model.*

Proof: The simulator for $\Pi_{\mathsf{PreInvSqrt}}$ appears in Fig. 4.20. The simulator emulates $\mathcal{F}_{\mathsf{A2B}}, \mathcal{F}_{\mathsf{B2A}}, \mathcal{F}_{\mathsf{PreOr}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}}, \mathcal{F}_{\mathsf{Sel}}$ in the order in which they appear in the protocol. In this way, the simulation proceeds by simulating the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from

the simulator. This is indistinguishable from its view in the real world, where it sees random values.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{PreInvSqrt}}$ honestly executes the protocol steps and proceeds as follows.

- Emulate $\mathcal{F}_{\mathsf{A2B}}$ on input $[\![\mathsf{a}]\!]$ and output $[\![\mathsf{a}]\!]^{\mathbf{B}}$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{PreOr}}$ on input $[\![\mathsf{a}_{\mathsf{k}-2}]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{a}_0]\!]^{\mathbf{B}}$ and outputs $[\![\mathsf{c}_{\mathsf{k}-2}]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{c}_0]\!]^{\mathbf{B}}$ to $\mathcal{A}$ and set $[\![\mathsf{c}_{\mathsf{k}-1}]\!]^{\mathbf{B}} = [\![0]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}_{\mathsf{k}-1}]\!]^{\mathbf{B}} = [\![\mathsf{c}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$ on behalf of the honest parties.
- Set $[\![\mathsf{v}_i]\!]^{\mathbf{B}} = [\![\mathsf{c}_i]\!]^{\mathbf{B}} \oplus [\![\mathsf{c}_{i+1}]\!]^{\mathbf{B}}$ for $i = \mathsf{k}-2$ to 1 on behalf of the honest parties.
- Emulate $\mathcal{F}_{\mathsf{B2A}}$ on input $\{[\![\mathsf{v}_i]\!]\}_{i=2\mathsf{f}-1}^{0}$ and output $[\![\mathsf{v}]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{a}]\!], [\![\mathsf{v}]\!], \mathsf{f}$ and output $[\![\mathsf{a}']\!]$ to $\mathcal{A}$.
- Set $\mathsf{k}' = \frac{\mathsf{k}}{2}, \mathsf{f}' = \frac{\mathsf{f}}{2}, c_0 = 2^{\frac{\mathsf{f}+1}{2}}, c_1 = 2^{\frac{\mathsf{f}}{2}}$ on behalf of the honest parties.
- Compute $[\![\mathsf{v}'_i]\!]^{\mathbf{B}} = [\![\mathsf{v}_{2i}]\!]^{\mathbf{B}} \oplus [\![\mathsf{v}_{2i+1}]\!]^{\mathbf{B}}$ for $i = 0$ to k' on behalf of the honest parties.
- Compute $[\![\mathsf{r}]\!]^{\mathbf{B}} = [\![\mathsf{v}_1]\!]^{\mathbf{B}} \oplus [\![\mathsf{v}_3]\!]^{\mathbf{B}} \oplus [\![\mathsf{v}_5]\!]^{\mathbf{B}} \oplus \ldots [\![\mathsf{v}_{\mathsf{k}-1}]\!]^{\mathbf{B}}$ on behalf of the honest parties
- Emulate $\mathcal{F}_{\mathsf{B2A}}$ on inputs $\{[\![\mathsf{v}'_i]\!]^{\mathbf{B}}\}_{i=2\mathsf{f}'-1}^{0}$ and output $[\![\mathsf{v}']\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Sel}}$ on inputs $\mathsf{c}_0, \mathsf{c}_1, [\![\mathsf{r}]\!]$ and output $[\![\mathsf{x}]\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Mul}}$ on inputs $[\![\mathsf{x}]\!], [\![\mathsf{v}]\!]$ and output $[\![\mathsf{v}']\!]$ to $\mathcal{A}$.

Figure 4.20: Simulator for $\Pi_{\mathsf{PreInvSqrt}}$.

□

Lemma 4.3 (Security) *Protocol* Π_{InvSqrt} *(Fig. 4.5) securely realizes* $\mathcal{F}_{\mathsf{InvSqrt}}$ *(Fig. 4.19) in the computational 4PC setting against a malicious adversary* $\mathcal{S}$ *in the* $(\mathcal{F}_{\mathsf{InvSqrt}}, \mathcal{F}_{\mathsf{Mul\text{-}Tr}})$*-hybrid model.*

Proof: The simulator for Π_{InvSqrt} appears in Fig. 4.21.

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

Let $P^\star \in \mathcal{P}$ be the party corrupted by $\mathcal{A}$. $\mathcal{S}_{\mathsf{InvSqrt}}$ honestly executes the protocol steps and proceeds as follows.

- Emulate $\mathcal{F}_{\mathsf{InvSqrt}}$ on input $[\![\mathsf{a}]\!]$ and output $[\![\mathsf{a}']\!], [\![\mathsf{v}']\!]$ to $\mathcal{A}$.
- Emulate $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ on inputs $[\![\mathsf{a}']\!], [\![\mathsf{a}']\!], \mathsf{f}$ and output $[\![\mathsf{y}_1]\!]$ to $\mathcal{A}$.
- Set $[\![\mathsf{y}_0]\!] = [\![\mathsf{a}']\!]$ and compute $[\![\mathsf{y}]\!] = 4.63887[\![\mathsf{y}_1]\!] - 5.77789[\![\mathsf{y}_0]\!] + 3.14736$ on behalf of honest parties.

Figure 4.21: Simulator for Π_{InvSqrt}.

The simulator begins by emulating $\mathcal{F}_{\mathsf{PreInvSqrt}}$. Following this, it emulates $\mathcal{F}_{\mathsf{Mul\text{-}Tr}}$ as per its invocation in the real-world protocol. In this way, the simulation proceeds by simulating

the steps of the underlying protocols. Note that since the simulator carries out the protocol steps honestly, $\mathcal{A}$'s view comprises only random values received from the simulator. This is indistinguishable from its view in the real world, where it sees random values. □

Lemma 4.4 (Security) *The shuffle protocol,* Π_{Shuffle} *(Fig. 4.12, Fig. 4.11) securely realizes the functionality* $\mathcal{F}_{\mathsf{Shuffle}}$ *(Fig. 4.10) against a malicious adversary that corrupts at most one party in* $\mathcal{P}$*, in the* $\mathcal{F}_{\mathsf{Setup}}$*-hybrid model.*

Proof: Let $\mathcal{S}$ represent the ideal-world adversary and $\mathcal{A}$ represent the adversary in the real world. On a high level, $\mathcal{S}$ starts by simulating $\mathcal{F}_{\mathsf{Setup}}$, where common keys are established with $\mathcal{A}$. These keys are used to sample the common randomness needed throughout the protocol. As a result, $\mathcal{S}$ is aware of all the randomness that $\mathcal{A}$ uses (more precisely, $\mathcal{S}$ is aware of shares of $\alpha_{\mathbf{v}}$, and $\beta_{\mathbf{v}}$ held by $\mathcal{A}$) and can extract the input from $\mathcal{A}$ as well as to confirm the correctness of messages sent by $\mathcal{A}$. After this, it simulates the steps of the shuffle protocol. Since P_0 is active only in the preprocessing phase and it possesses all the input-independent data, security against P_0 follows easily. The simulation steps for a corrupt P_2 are provided below, where the corresponding simulator is denoted as $\mathcal{S}^{P_2}$. The simulation for a corrupt P_1 follows along the same lines as P_2.
$\mathcal{S}^{P_2}$ proceeds as follows.

Preprocessing:

– Using the keys established via $\mathcal{F}_{\mathsf{Setup}}$, sample the common randomness with $\mathcal{A}$.

– Simulate the steps of Π_{JSh} acting as the sender together with $\mathcal{A}$ to generate $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}i})$ for $i \in \{2,3\}$.

– Simulate the steps of Π_{JSh} with $\mathcal{A}$ as the receiver to generate $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}1})$.

Simulation of generation of $\mathbf{b}_1$ *towards* P_1

– Sample a random π_s, π_2 and compute $\Pi = \pi_s \circ \pi_1 \circ \pi_2$, where π_1 is held by $\mathcal{A}$. Simulate the steps of Π_{JSh} acting as the sender to send Π to $\mathcal{A}$.

– Sample a random $\mathbf{z}_{21} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and compute $\mathbf{y}_{11} = \pi_2(\boldsymbol{\gamma}_{13} - \mathbf{z}_{21})$ as per the protocol. Simulate steps of Π_{Jmp} acting as the sender to send $\mathbf{y}_{11}$ to $\mathcal{A}$.

– Compute $\mathbf{y}_{21} = \pi_1(\mathbf{y}_{11} - \mathbf{z}_{11})$ and $\mathbf{y}_{31} = \pi_3(\mathbf{y}_{21} + \mathbf{z}_{31})$ as per the protocol. Simulate steps of Π_{Jmp} acting as the sender together with $\mathcal{A}$ to send $\alpha_{\mathbf{w}2} - \mathbf{y}_{31}$ to P_1.

Simulation of generation of $\mathbf{b}_2$ *towards* P_2

– Sample a random $\mathbf{z}_{22} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and compute $\mathbf{y}_{12} = \pi_2(\boldsymbol{\gamma}_{23} - \mathbf{z}_{22})$ as per the protocol. Simulate steps of Π_{Jmp} acting as the sender to send $\mathbf{y}_{12}$ to $\mathcal{A}$.

– Compute $\mathbf{x}_{32} = \pi_3(\mathbf{x}_{22} - \mathbf{z}_{32})$ as per the protocol. Sample a random $\alpha_{\mathbf{w}1} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and simulate steps of Π_{Jmp} acting as the sender to send $\alpha_{\mathbf{w}1} - \mathbf{x}_{32}$ to $\mathcal{A}$.

Online:

– Sample a random $\mathbf{r}_2 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and compute $\mathbf{a}_2 = \pi_2(\beta_{\mathbf{v}} + \mathbf{r}_2)$. Simulate the steps of Π_{Jmp} acting as the sender to send $\mathbf{a}_2$ to $\mathcal{A}$.

– Compute $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\beta_{\mathbf{v}} + \mathbf{r}_1)))$ as per the protocol, and simulate steps of Π_{Jmp} acting as the sender together with $\mathcal{A}$ to send $\mathbf{a}_1$ to P_1.

– Compute $\beta_{\mathbf{w}}$ as per the protocol, and simulate steps of Π_{Jmp} acting as the sender together with $\mathcal{A}$ to send $\beta_{\mathbf{w}}$ to P_3.

Observe that the messages received by $\mathcal{A}$ in the real-world comprise of the random $\mathbf{a}_2$ in the online phase. In the preprocessing phase, P_2 receives a random permutation Π, and $\mathbf{y}_{11}$, $\mathbf{y}_{12}$, $\alpha_{\mathbf{w}1} - \mathbf{x}_{32}$ which come from a uniform random distribution. Observe that in the above simulation, these messages received by $\mathcal{A}$ continue to come from a uniform distribution. This is because each of these messages is generated by using randomness sampled from a uniform distribution by $\mathcal{S}^{P_2}$. Hence, the real-world and ideal world views for $\mathcal{A}$ are indistinguishable.

The simulation steps for a corrupt P_3 are provided below. $\mathcal{S}^{P_3}$ proceeds as follows.

Preprocessing:

– Using the keys established via $\mathcal{F}_{\mathsf{Setup}}$, sample the common randomness with $\mathcal{A}$.

– Simulate the steps of Π_{JSh} acting as the sender together with $\mathcal{A}$ to generate $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}i})$ for $i \in \{1, 2\}$.

– Simulate the steps of Π_{JSh} with $\mathcal{A}$ as the receiver to generate $[\![\cdot]\!]$-shares of $\pi_0(\alpha_{\mathbf{w}3})$.

Simulation of generation of Π *towards* P_2

– Compute $\Pi = \pi_s \circ \pi_1 \circ \pi_2$ as per the protocol. Simulate the steps of Π_{Jmp} acting as the sender together wit $\mathcal{A}$ to send Π to P_1.

Simulation of generation of $\mathbf{b}_1$ *towards* P_1

– Compute $\mathbf{x}_{21} = \pi_1(\mathbf{x}_{11} + \mathbf{z}_{11})$ as per the protocol. Simulate steps of Π_{Jmp} acting as the sender along with $\mathcal{A}$ to send $\mathbf{x}_{21}$ to P_1.

Simulation of generation of $\mathbf{b}_2$ *towards* P_2

– Compute $\mathbf{x}_{22} = \pi_1(\mathbf{x}_{12} + \mathbf{z}_{12})$ as per the protocol. Simulate steps of Π_{Jmp} acting as the sender along with $\mathcal{A}$ to send $\mathbf{x}_{22}$ to P_1.

Online:

– Compute $\mathbf{a}_1 = \pi_s(\pi_1(\pi_2(\beta_{\mathbf{v}} + \mathbf{r}_1)))$ as per the protocol, and simulate steps of Π_{Jmp} acting as the sender together with $\mathcal{A}$ to send $\mathbf{a}_1$ to P_1.

– Compute $\mathbf{a}_2 = \pi_2(\beta_{\mathbf{v}} + \mathbf{r}_2)$ as per the protocol, and simulate steps of Π_{Jmp} acting as the other sender together with $\mathcal{A}$ to send $\mathbf{a}_2$ to P_2.

– Compute $\beta_{\mathbf{w}}$ as per the protocol, and simulate steps of Π_{Jmp} acting as the sender to send $\beta_{\mathbf{w}}$ to $\mathcal{A}$.

Observe that the messages received by $\mathcal{A}$ in the real-world comprise of the random $\beta_{\mathbf{w}}$ in the online phase. In the preprocessing phase, P_3 does not receive any values. Observe that in the above simulation, these messages received by $\mathcal{A}$ continue to come from a uniform distribution. This is because each of these messages is generated by using randomness sampled from a uniform distribution by $\mathcal{S}^{P_3}$. Hence, the real-world and ideal world views for $\mathcal{A}$ are indistinguishable.

□

Chapter 5

Secure Dark Pools

This chapter discusses (1,1)-FaF secure 5-party computation (5PC) protocols that consider one malicious and one semi-honest corruption and constitutes the optimal setting for attaining an honest majority. We then discuss the application of a secure dark pool which is realized using the (1,1)-FaF secure 5PC protocols. The results in this chapter have led to a publication at ACM CCS 2022 [137].

5.1 Overview

The recent work of [5] identified the shortcomings of traditional MPC and defined a Friends-and-Foes (FaF) security notion to address the same. We showcase the need for FaF security in real-world applications such as dark pools. This subsequently necessitates designing concretely efficient FaF-secure protocols. Towards this, keeping efficiency at the centre stage, we design ring-based FaF-secure MPC protocols in the small-party honest-majority setting. Specifically, we provide (1,1)-FaF secure 5PC protocols that consider one malicious and one semi-honest corruption and constitutes the optimal setting for attaining an honest majority. To facilitate having FaF-secure variants for several applications, we design a variety of building blocks optimized for our FaF setting. The practicality of the designed (1,1)-FaF secure 5PC framework is showcased via the application of dark pools, where not only do our protocols improve in terms of security guarantees provided but also in terms of efficiency over the existing traditionally secure protocols for the same. This improvement is witnessed as a gain of up to 62× in throughput compared to the existing ones. Finally, to demonstrate the versatility of our framework, we also benchmark popular deep neural networks. Detailed results in this chapter are as follows.

$(1,1)$-FaF secure 5PC

Departing from traditionally secure MPC protocols providing GOD, we design GOD protocols in the FaF-secure model. Towards this, with efficiency in mind, we work over the ring $\mathbb{Z}_{2^\ell}$, both arithmetic and Boolean ($\ell = 1$) and design (1,1)-FaF secure 5PC protocols. The protocols are cast in the preprocessing model since it offloads heavy input-independent computations to a preprocessing phase, resulting in a fast input-dependent online phase. The highlight here is the multiplication which requires—(i) *just three* parties to be online for most of the computation and (ii) requires one round (amortized) and eight ring elements of communication in the online phase. The efficiency and resource management (involvement of only 3 parties for most of the computation) of the multiplication results in a concretely efficient 5PC framework. We concretely showcase the benefit of having a reduced number of online parties over a naive solution (all parties online) as well as the traditional $(5, 2)$ maliciously secure protocol.

Building blocks and generality

We resort to a modular approach to design various building blocks, as shown in Fig. 5.1, where protocols in each layer build on those in the previous layers. Layer 0 forms the core MPC, with layers above it providing the building blocks. This constitutes our generic and comprehensive framework since it provides support for a wide range of building blocks that suffice for various applications. While these building blocks have been well studied in the literature, our contribution lies in designing and optimizing these for the 5PC (1,1)-FaF setting.

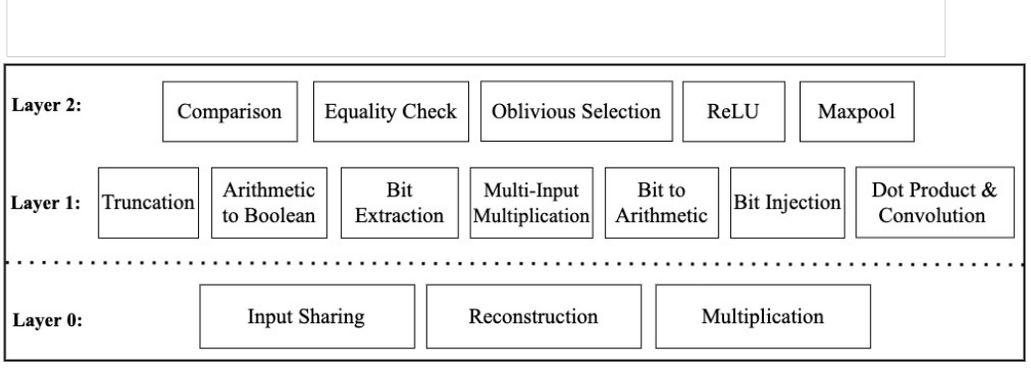


Figure 5.1: Designed (1, 1) FaF-secure 5PC framework.

The designed building blocks have been extensively used in realizing privacy-preserving machine learning (PPML) [49, 193, 50, 38, 136, 173, 174], albeit in the traditional security model. Since PPML in itself is suitable for a wide range of application scenarios, we demonstrate the versatility and practicality of the designed (1,1)-FaF secure 5PC by also considering PPML

as another application. For this, we benchmark the performance of the designed protocols for secure inference using popular deep neural networks such as LeNet [147] and VGG16 [213].

Secure dark pools

Although secure dark pools have been considered in the traditional MPC setting, we design improved protocols for the same in the (1, 1)-FaF setting for 5PC. We optimize the continuous double auction (CDA) and volume-based matching algorithms. We identify several aspects of the matching algorithms that can be performed in parallel, which improves the efficiency of the designed protocols. We benchmark the performance of these secure matching algorithms and observe a throughput improvement of up to 62× in comparison to [40].

5.2 Related work

The work of [5] focuses on extending the standard security notion of MPC to the FaF-setting. In this regard, they provide both, a full-security as well as fairness variants in this new setting. They further provide a detailed investigation of various feasibility results and limitations in the FaF-setting. The (1,1)-FaF secure 5PC protocol designed in this chapter forms the first concrete instantiation of a FaF-secure protocol, particularly as the optimal case for an honest-majority setting for a small number of parties. We, therefore, next discuss relevant secret sharing-based MPC works that provide GOD in a small-party setting under the traditional security model. A concretely efficient protocol for achieving GOD was provided in [136], both for 3PC and 4PC setting, which improved over the 4PC of [38] and the 3PC of [30]. Note that [38], in turn, improved upon the GOD protocols in [91]. The work in [61] proposed 4PC protocols on par with [136], albeit with the security of *private robustness*. However, the security guarantees of both SWIFT and [61] are known to be theoretically equivalent. The recent work of [138] provides an improved multiplication protocol over [136] in the 4PC setting. The improvement is seen in the preprocessing phase, where [138] requires only 2 ring elements as opposed to 3. While there are no protocols explicitly designed for 5PC that attain GOD, [31] provides protocols for the n-party setting, from which a 5PC protocol can be derived. The work of [37] attains GOD in the 5PC setting, albeit relying on garbled circuits. With respect to the primitives, note that these have been extensively studied in the literature and our contribution lies in adapting these for the FaF setting, while incorporating improvements wherever possible. The relevant literature with respect to the primitives appears in §3.2, §4.2.

5.3 Preliminaries

5.3.1 System model

We design protocols that comprise five parties $\mathcal{P} = \{P_1, P_2, \ldots, P_5\}$ that are connected via pairwise private and authentic channels in a synchronous network. Our protocols are FaF-secure with a static, malicious probabilistic polynomial time (PPT) adversary that can corrupt up to one party, and a *different* semi-honest adversary that can corrupt at most one other party. The set of computing parties $\mathcal{P}$ may be equivalently represented as $\mathcal{P} = \{P_i, P_j, P_k, P_l, P_m\}$ for ease of presentation.

5.3.2 Joint message passing (Jmp)

This primitive enables two parties to send a common message to a third party such that the recipient either receives the correct message or in case of an inconsistency in the received messages, a trusted third party (TTP) is identified [136]. The protocol involves one sender sending the value, while the other sending the hash to the receiver, who then compares the received values; in case of an inconsistency, the parties proceed to identify a TTP, who then completes the computation of MPC on the clear after receiving inputs from the parties. As opposed to the protocol of SWIFT [136], we cannot use TTP in the same way in the $(1, 1)$-FaF setting as the TTP learns all the inputs. Thus, we modify the Jmp protocol in [136] to adapt it to the $(1, 1)$-FaF setting as follows—in case of an inconsistency, we modify Jmp to output a pair of parties in conflict, one of which is guaranteed to be maliciously corrupt, instead of identifying a TTP. Note that the Jmp protocol consists of two phases (*send, verify*). The *send* phase consists of one of the two senders, denoted as the *speaker* party sending the message to the receiver, while the other sender party, referred to as the *silent* party, keeps quiet. This distinction between the *speaker* and a *silent* party is made only for the *send* phase. *Verify* phase comprises all the other steps of the Jmp protocol, and either confirms that message delivery to the recipient was a success or identifies a conflict pair, CP. Looking ahead, our protocols rely on several invocations of Jmp. Hence, to leverage amortization, in most cases, the *send* phase is executed on the flow, and the *verify* phase is deferred to a later stage. This deferring of *verify* brings significant challenges in our protocols, such as multiplication. A part of our novelty comes from handling these challenges.

We say P_i, P_j Jmp-send msg to P_k when they invoke the *send* phase of $\Pi_{\mathsf{Jmp}}(P_i, P_j, P_k, \mathsf{msg})$. Without loss of generality, we let P_i be the *speaker* and P_j be the *silent* party. Since verification can be deferred, we say that P_i, P_j Jmp-vrfy towards P_k when they invoke only the deferred

verify phase corresponding to $\Pi_{\mathsf{Jmp}}(P_i, P_j, P_k, \mathsf{msg})$. Finally, we say P_i, P_j Jmp-sv msg to P_k when they invoke the complete $\Pi_{\mathsf{Jmp}}(P_i, P_j, P_k, \mathsf{msg})$ protocol and execute both the *send* and *verify* phases together.

The modified protocol for Jmp appears in Fig. 5.2. The protocol is described with respect to a single message v for a fixed-ordered pair of senders and a given receiver. However, we note that *verify* phase across several messages for the same ordered pair of senders and receiver can be bundled together, thereby amortizing this cost. This would involve party P_j (silent party) sending a single hash corresponding to all the messages under consideration and performing the verification accordingly.

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

Each party P_s for $s \in \{i, j, k\}$ initializes bit $\mathsf{b}_s = 0$. Let CP denote the conflict pair, which is the pair of parties in conflict, one of which is guaranteed to be corrupt. Let P_i, P_j denote the senders who wish to send v to receiver P_k. Let H denote a collision-resistant hash function.

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

– P_k broadcasts (accuse, P_i), if P_i is silent, and all take $\mathsf{CP} = (P_i, P_k)$ as the conflict pair. Analogously for P_j. If P_k accuses both P_i, P_j, then $\mathsf{CP} = (P_i, P_k)$. Otherwise, P_k receives some $\tilde{\mathsf{v}}$, and either sets $\mathsf{b}_k = 0$ when the value and the hash are consistent or sets $\mathsf{b}_k = 1$. P_k then sends b_k to P_i, P_j and terminates if $\mathsf{b}_k = 0$.

– If P_i does not receive a bit from P_k, it broadcasts (accuse, P_k) and $\mathsf{CP} = (P_i, P_k)$. Analogously for P_j. If both P_i, P_j accuse P_k, then $\mathsf{CP} = (P_i, P_k)$. Otherwise, P_s for $s \in \{i, j\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.

– P_i, P_j exchange their bits with each other. If P_i does not receive b_j from P_j, it broadcasts (accuse, P_j) and $\mathsf{CP} = (P_i, P_j)$. 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 $\mathsf{b}_s = 1$, 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 $\mathsf{CP} = (P_i, P_j)$. Otherwise,

- If $\mathsf{H}_i \neq \mathsf{H}_j$: $\mathsf{CP} = (P_i, P_j)$.
- Else if $\mathsf{H}_i \neq \mathsf{H}_k$: $\mathsf{CP} = (P_i, P_k)$.
- Else if $\mathsf{H}_i = \mathsf{H}_j = \mathsf{H}_k$: $\mathsf{CP} = (P_j, P_k)$.

Figure 5.2: Joint message passing.

5.3.3 Secret sharing semantics

In the 5PC $(1,1)$-FaF setting, a (semi-honest) adversary may be entitled to the view of at most two parties (itself and the malicious party). Thus, to ensure that the view of the two parties does not leak any additional information, we rely on a $(5,2)$ replicated secret sharing (RSS) scheme and its variants. A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be RSS-shared among 5 parties with threshold 2 if for every subset of two parties, say $\{P_i, P_j\}$, the residual three parties hold share $\mathsf{v}_{ij} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = \sum_{1 \le i < j \le 5} \mathsf{v}_{ij}$. Observe that since any set of two parties in $\mathcal{P}$ always miss one share of v, they cannot reconstruct the value, whereas any three parties can. The total number of shares of a value is thus $\binom{5}{2} = 10$ and the RSS-share possessed by $P_s \in \mathcal{P}$ is a tuple of $\binom{4}{2} = 6$ shares v_{ij} where $s \neq i, s \neq j$ and $1 \le i < j \le 5$. With this background, we define our sharing semantics below.

– $[\cdot]$-sharing: A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared among parties in $\mathcal{P}$ if it is $(5,2)$ RSS-shared among them. We let $[\mathsf{v}]_s$ denote P_s's $[\cdot]$-shares of v. Note that $[\mathsf{v}]_s$ is a tuple of 6 elements, and $[\mathsf{v}]$ is a tuple of 10 elements.

– $\langle\cdot\rangle$-sharing: A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $\langle\cdot\rangle$-shared among parties in $\mathcal{P}$ if there exists $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_5 \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2 + \mathsf{v}_3 + \mathsf{v}_4 + \mathsf{v}_5$ and $P_s \in \mathcal{P}$ possess v_s. Let $\langle\mathsf{v}\rangle = (\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_5)$.

– $[\![\cdot]\!]$-sharing: A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}$, if
  - there exists $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared among parties in $\mathcal{P}$, and
  - there exists $\beta_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ such that $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ is held by all parties in $\mathcal{P}$.

For a set of n values $\{\mathsf{v}_1, \ldots \mathsf{v}_n\}$, we let $\beta_{\mathsf{v}_1 \ldots \mathsf{v}_n} = \Pi_{i=1}^n \beta_{\mathsf{v}_i}$ and $\alpha_{\mathsf{v}_1 \ldots \mathsf{v}_n} = \Pi_{i=1}^n \alpha_{\mathsf{v}_i}$. We use the superscript $\mathbf{B}$ to denote the Boolean sharing over $\mathbb{Z}_2$, while the absence of it implies arithmetic sharing over $\mathbb{Z}_{2^\ell}$. All the above sharing schemes are linear, i.e., given shares of $\mathsf{v}_1, \mathsf{v}_2$, and public constants c_1, c_2, parties can locally compute the shares of $c_1\mathsf{v}_1 + c_2\mathsf{v}_2$.

Conversion between $[\![\cdot]\!]$ and $[\cdot]$ shares during preprocessing. Given $[\mathsf{v}]$, protocol $\Pi_{[\cdot]\to[\![\cdot]\!]}$ generates $[\![\mathsf{v}]\!]$ from it by setting $\beta_\mathsf{v} = 0$ and $[\alpha_\mathsf{v}] = -[\mathsf{v}]$. Conversely, $\Pi_{[\![\cdot]\!]\to[\cdot]}$ generates $[\mathsf{v}]$ from $[\![\mathsf{v}]\!]$. For this, parties set $\mathsf{v}_{12} = \beta_\mathsf{v} - \alpha_{\mathsf{v}12}$ while $\mathsf{v}_{ij} = -\alpha_{\mathsf{v}ij}$ for $1 \le i < j \le 5, (i,j) \neq (1,2)$.

Shared key setup Following several recent works [11, 12, 173, 50, 38, 193, 136], to enable non-interactive communication between the parties, a one-time setup is performed that establishes common random keys for a pseudo-random function (PRF) F. Here $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ is a secure PRF, with co-domain X being $\mathbb{Z}_{2^\ell}$. The key setup is modelled via a functionality $\mathcal{F}_{\mathsf{Setup}}$ (Fig. 5.3) that can be realized using any FaF-secure MPC protocol. The goal is to establish a common key between every set of 2, 3, 4, and all parties. To sample a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$

among a set of 3 parties P_i, P_j, P_k non-interactively, each of these parties invoke $F_{k_{ijk}}(id_{ijk})$ and obtain r. Here, id_{ijk} denotes a counter maintained by these three parties and is updated after every PRF invocation. The appropriate keys used to sample the common randomness are implicit from the context and from the identities of the parties that sample.

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

$\mathcal{F}_{\mathsf{Setup}}$ interacts with parties in $\mathcal{P}$ and adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$. $\mathcal{F}_{\mathsf{Setup}}$ picks the following keys.

- A common random key $k_{\mathcal{P}}$ for all the parties.
- A common key k_{ij} between every pair of parties P_i, P_j where $1 \leq i < j \leq 5$.
- A common key k_{ijk} between every set of 3 parties P_i, P_j, P_k where $1 \leq i < j < k \leq 5$.
- A common key k_{ijkl} between every set of 4 parties P_i, P_j, P_k, P_l where $1 \leq i < j < k < l \leq 5$.

Output: Keys $\{k_{\mathcal{P}}, k_{si}, k_{js}, k_{sjk}, k_{isk}, k_{ijs}, k_{sjkl}, k_{iskl}, k_{ijsl}, k_{ijks}\}$, generated as above, are output to every $P_s \in \mathcal{P}$.

Figure 5.3: Ideal functionality for shared-key setup.

5.3.4 Notations

The notations used in this chapter are summarized in Table 5.1.

Notation	Description
TTP	Trusted third party
CP	Conflicting pair of parties
Jmp	Joint message passing primitive
Jmp-send	Send phase of Jmp
Jmp-vrfy	Deferred verification phase of Jmp
Jmp-sv	Send phase together with verify phase of Jmp
$\mathcal{B}$	Sorted list of M buy orders
$\mathcal{S}$	Sorted list of N sell orders
$name^b$	Name of the client in a buy order
b	Number of units to be bought in a buy order
q	Buying price in a buy order, also known as 'bid'
$name^s$	Name of the client in a sell order
s	Number of units to be sold in a sell order
p	Selling price in a sell order, also know as 'offer'

Table 5.1: Notations used in this chapter.

5.4 Input sharing

Protocol Π_{Sh} (Fig. 5.4) enables $P_i \in \mathcal{P}$ holding a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to generate $[\![\mathsf{v}]\!]$. For this, parties generate $[\cdot]$-shares of a random value $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ in the preprocessing phase, non-interactively, using their shared key setup such that the dealer P_i learns all the $[\cdot]$-shares of α_{v}. This enables P_i to compute $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ in the online phase and Jmp-sv it to all the parties.

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

Preprocessing:

– Parties non-interactively generate $[\cdot]$-shares of a random $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that P_i learns all shares of α_{v}, using the shared-keys.

Online:

– P_i computes and sends $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to one other party, say P_j.

– P_i, P_j then Jmp-sv β_{v} to all other parties.

Figure 5.4: Generating $[\![\mathsf{v}]\!]$ by party P_i.

The protocol for generating $[\cdot]$-shares of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is similar to above and formal details appear in Fig. 5.5.

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

Let v_{lm} be a share of v held by $P_i, P_j, P_k \in \mathcal{P}$.

– Parties in $\mathcal{P} \setminus \{P_p, P_q\}$ for $1 \le p < q \le 5$ and $p \neq l, q \neq m$, non-interactively generate $\mathsf{v}_{pq} \in \mathbb{Z}_{2^\ell}$ together with P_i, using the shared-key setup.

– P_i computes and sends $\mathsf{v}_{lm} = \mathsf{v} - \sum_{1 \le p < q \le 5, p \neq l, q \neq m} \mathsf{v}_{pq}$ to P_j, following which P_i, P_j Jmp-sv v_{lm} to P_k.

Figure 5.5: Generating $[\mathsf{v}]$ by party P_i.

Protocol Π_{JSh2} is a variant of input sharing Π_{Sh}, which enables two parties P_i, P_j to jointly generate $[\![\cdot]\!]$-shares of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ known to both. Looking ahead, this protocol is heavily used in designing the building blocks and is similar to Π_{Sh}. Here, during the preprocessing phase parties generate $[\alpha_{\mathsf{v}}]$ for $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that P_i, P_j learn all its shares. Following this, P_i, P_j generate and Jmp-send $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ towards all the other parties with its Jmp-vrfy deferred. The protocol appears in Fig. 5.6. We next describe a few optimizations that can be performed on Π_{JSh2}.

Protocol $\Pi_{\mathsf{JSh2}}(P_i, P_j, \mathsf{v})$

Preprocessing:

– Parties non-interactively generate $[\cdot]$-shares of a random $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that P_i, P_j learn all shares of α_{v}, using the shared keys.

Online:

– P_i, P_j compute and Jmp-sv $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to all other parties.

Figure 5.6: Joint sharing of v by P_i, P_j.

When the value to be shared is available with P_i, P_j in the preprocessing phase, the protocol can be optimized as follows. All parties set $\beta_{\mathsf{v}} = 0$. P_i, P_j, P_k non-interactively sample a random $\mathsf{r}_{lm} \in \mathbb{Z}_{2^\ell}$ and set the common $[\cdot]$-share of α_{v} they possess as $\alpha_{\mathsf{v}_{\mathsf{lm}}} = \mathsf{r}_{lm}$. Similarly, P_i, P_j, P_l non-interactively sample a random $\mathsf{r}_{km} \in \mathbb{Z}_{2^\ell}$ and set the common $[\cdot]$-share of α_{v} they possess as $\alpha_{\mathsf{v}_{\mathsf{km}}} = \mathsf{r}_{km}$. P_i, P_j set the common share of α_{v} held together with P_m as $\alpha_{\mathsf{v}_{\mathsf{kl}}} = -(\mathsf{v} + \mathsf{r}_{lm} + \mathsf{r}_{km})$ and Jmp-sv $\alpha_{\mathsf{v}_{\mathsf{kl}}}$ to P_m. The other $[\cdot]$-shares of α_{v} are set as 0.

When the value to be shared is held by three parties, say P_i, P_j, P_k, the protocol proceeds similarly to Π_{JSh2}, with the following difference—in the preprocessing phase, α_{v} will be also be learned by P_k, and in the online phase, only two Jmp-sv are required. We call the resultant protocol Π_{JSh3}, and omit the formal protocol due to its close resemblance to Π_{JSh2}. Moreover, when the value is available with these three parties in the preprocessing phase, the protocol can be made completely non-interactive. For this, similar to the previous case, β_{v} is set as 0, and the common $[\cdot]$-share of α_{v} held by P_i, P_j, P_k is set as $-\mathsf{v}$ and all other shares are set as 0.

Finally, when all parties hold a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, they can generate $[\![\mathsf{v}]\!]$ by setting $\beta_{\mathsf{v}} = \mathsf{v}$ and all $[\cdot]$-shares of α_{v} as 0.

5.5 Reconstruction

Protocol Π_{Rec} enables robust reconstruction of a $[\![\cdot]\!]$-shared value v towards P_i. For this, observe that each party misses 4 shares, and each such share is held by three other parties. Thus, to reconstruct v towards P_i, parties can send the missing shares to P_i. For each share, P_i uses the value which appears in the majority to reconstruct v. As on optimization, we let two parties send the value while the third send its hash to P_i. The protocol appears in Fig. 5.7.

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

Let the missing shares at P_i be $\mathsf{v}_{ij}, \mathsf{v}_{ik}, \mathsf{v}_{il}, \mathsf{v}_{im}$.

– Let P_k, P_l, P_m possess v_{ij}. P_k, P_l send v_{ij} to P_i while P_m sends its hash to P_i. Analogous steps are carried out for the other three shares.

– P_i uses the value which appears in the majority for the received missing shares, together with its own shares, for reconstructing v as $\mathsf{v} = \sum_{1 \leq p < q \leq 5} \mathsf{v}_{pq}$.

Figure 5.7: Reconstruction of v towards P_i.

5.6 Multiplication

The multiplication protocol Π_{Mul} allows parties to compute $[\![\mathsf{z}]\!] = [\![\mathsf{a} \cdot \mathsf{b}]\!]$, where $\mathsf{a}, \mathsf{b} \in \mathbb{Z}_{2^\ell}$ are $[\![\cdot]\!]$-shared. The highlight of our protocol is that it requires a single *online* round for evaluating a multiplication gate and requires active participation from only three parties for most of the computation. The protocol proceeds as follows. In the preprocessing phase, parties first generate $[\alpha_\mathsf{z}] \in \mathbb{Z}_{2^\ell}$, non-interactively, using their shared key setup. To generate $[\![\mathsf{z}]\!]$, parties need to compute β_z which can be written as follows: $\beta_\mathsf{z} = \mathsf{z} + \alpha_\mathsf{z} = \mathsf{ab} + \alpha_\mathsf{z} = (\beta_\mathsf{a} - \alpha_\mathsf{a})(\beta_\mathsf{b} - \alpha_\mathsf{b}) + \alpha_\mathsf{z} = \beta_\mathsf{ab} - \beta_\mathsf{a}\alpha_\mathsf{b} - \beta_\mathsf{b}\alpha_\mathsf{a} + \alpha_\mathsf{ab} + \alpha_\mathsf{z}$, where $\beta_\mathsf{ab} = \beta_\mathsf{a}\beta_\mathsf{b}$ and $\alpha_\mathsf{ab} = \alpha_\mathsf{a}\alpha_\mathsf{b}$. Observe that parties already possess $\beta_\mathsf{a}, \beta_\mathsf{b}$ and $[\cdot]$-shares of $\alpha_\mathsf{a}, \alpha_\mathsf{b}, \alpha_\mathsf{z}$. Assuming that $[\alpha_\mathsf{ab}]$ is also made available, parties can compute $[\beta_\mathsf{z}]$, leveraging the linearity of $[\cdot]$-sharing. We discuss how to generate $[\alpha_\mathsf{ab}]$ in the preprocessing phase later and focus on the remaining steps assuming that $[\alpha_\mathsf{ab}]$ is given. Now, β_z can be reconstructed towards all the parties, thereby generating $[\![\mathsf{z}]\!]$. This reconstruction towards $P_i \in \mathcal{P}$ can be performed using just two invocations of Π_{Jmp} as follows. The four shares missing at P_i, which include $\{\beta_{\mathsf{z_{ij}}}, \beta_{\mathsf{z_{ik}}}, \beta_{\mathsf{z_{il}}}, \beta_{\mathsf{z_{im}}}\}$ are sent to it as—P_j, P_k Jmp-sv $\{\beta_{\mathsf{z_{il}}} + \beta_{\mathsf{z_{im}}}\}$ while P_l, P_m Jmp-sv $\{\beta_{\mathsf{z_{ij}}} + \beta_{\mathsf{z_{ik}}}\}$.

5.6.1 Towards an efficient online phase

The above approach requires all parties to be online. However, observe that P_1, P_2, P_3 possess the required shares to compute the entire function. Hence, to reduce the number of active parties in the online phase, whenever multiplication is invoked, we restrict the reconstruction of β_z only towards the *online parties*, P_1, P_2, P_3 (but without the correctness guarantee), and defer reconstruction towards P_4, P_5 to a later point. Thus, only the Jmp-send with respect to following 6 Jmps are invoked—$\Pi_{\mathsf{Jmp}}(P_2, P_4, P_1, \beta_{\mathsf{z}13} + \beta_{\mathsf{z}15})$, $\Pi_{\mathsf{Jmp}}(P_3, P_5, P_1, \beta_{\mathsf{z}12} + \beta_{\mathsf{z}14})$, $\Pi_{\mathsf{Jmp}}(P_1, P_4, P_2, \beta_{\mathsf{z}23} + \beta_{\mathsf{z}25})$, $\Pi_{\mathsf{Jmp}}(P_3, P_5, P_2, \beta_{\mathsf{z}12} + \beta_{\mathsf{z}24})$, $\Pi_{\mathsf{Jmp}}(P_1, P_4, P_3, \beta_{\mathsf{z}23} + \beta_{\mathsf{z}35})$, $\Pi_{\mathsf{Jmp}}(P_2, P_5, P_3, \beta_{\mathsf{z}13} + \beta_{\mathsf{z}34})$. Recall that since only the *send* of Jmp is performed, the *silent*

parties, P_4, P_5, can remain offline. To complete the generation of $[\![\mathsf{z}]\!]$, and enable P_4, P_5 to obtain β_{z}, we let P_1, P_2 Jmp-sv β_{z} to P_4, P_5. We can defer this step until the output reconstruction stage, where βs corresponding to all the invocations of multiplication until output reconstruction are sent in a *single* round. Deferring the send of β_{z} to after output reconstruction, may result in incorrectly reconstructing z. With this approach, evaluating multiplications requires participation from only the online parties (P_1, P_2, P_3) for most of the computation and *offline parties* (P_4, P_5) become active only before output reconstruction. Further, only a single round (owing to the *send* phase of Jmp where only *speaker* party communicates) is needed for reconstructing each β among the online parties. Observe that the following two issues may arise while executing the above approach

(a) *correctness:* β_{z} reconstructed among online parties P_1, P_2, P_3 may be incorrect;

(b) *agreement:* online parties may not be in agreement with respect to the β_{z} they hold, let alone hold the correct β_{z}.

Both issues arise since only the *send* phase of Jmp is executed among the online parties while reconstructing β_{z}, which may lead to incorrect reconstruction among them. We next describe how both these issues can be addressed in the verification phase. Looking ahead, resolution for both issues either results in successfully completing the protocol or identification of CP. In the latter case, parties switch to 3PC (after share conversion) for the rest of the computation.

(a) Ensuring correctness. Correctness of the β_{z} reconstructed towards the online parties can be enforced by executing the Jmp-vrfy towards them, and requires P_4, P_5. For this, P_4, P_5 should possess the correct inputs used for generating β_{z}, which may themselves be outputs, $\beta_{\mathsf{a}}, \beta_{\mathsf{b}}$, of multiplications. As mentioned earlier, P_4, P_5 receive all these βs in a single invocation of Jmp-sv from P_1, P_2 just before output reconstruction. However, P_1, P_2 may not be in *agreement* with respect to these βs due to incorrect reconstruction of the same. Performing Jmp when the senders are not in agreement with respect to the value being sent may result in incorrectly identifying a pair of honest parties as a CP (conflict pair). This necessitates a consistency check to ensure that P_1, P_2 are in agreement, and is discussed later. Hence, assuming P_1, P_2 are in agreement after this consistency check, they proceed to Jmp-sv βs for all these multiplications to P_4, P_5. If this Jmp-sv towards P_4, P_5 succeeds (i.e., no CP identified), verification of βs, reconstructed among the online parties, is performed. This is done by invoking deferred Jmp-vrfy corresponding to all the Jmp-send performed among the online parties. The success of all the *verify* phases guarantees the correctness of βs. In case if any *verify* fails, a CP is identified.

(b) Ensuring agreement. We now describe the consistency check mentioned above. In order to ensure agreement among online parties P_1, P_2, P_3, they exchange the hash of βs for all the multiplications among themselves. If these are consistent, then they proceed with the correctness check as described above. If the consistency check fails, the goal is to identify a CP. Observe that the check may fail due to one of the following reasons: (i) an incorrect β was reconstructed towards some honest online party which led to sending an incorrect hash during the check, or (ii) an incorrect hash was deliberately sent. Note that case (i) arises if a malicious online party misbehaved during a Jmp-send performed at some level (layer in the circuit comprising addition and multiplication gates.) during circuit evaluation. Hence, performing the Jmp-vrfy of this particular Jmp-send can identify a CP and address case (i). A keen observer would note the circularity involved in addressing the agreement issue by relying on *verify* of Jmp (to identify a CP). The circularity arises due to the following reason. Jmp-vrfy towards P_1, P_2, P_3 requires P_4, P_5 to hold consistent β. Since P_4, P_5 receive the β via Jmp-sv from P_1, P_2, it requires the latter to already be in agreement, and hence the circularity. To break the circularity, online parties rely on a binary search of levels within the circuit. The search identifies consecutive levels L_p, L_{p+1} such that all βs up to level L_p are consistently held among P_1, P_2, P_3 while L_{p+1} onwards they are inconsistent. The consistency of βs up to L_p thus enables usage of Jmp. For the binary search, parties exchange the hash with respect to βs in the first (top) half of the circuit, say $\mathsf{max}/2$ levels (where max denotes the maximum levels in the circuit). If the hash is inconsistent, they recursively proceed with the first half (L_1 to $L_{\mathsf{max}/2}$), else if consistent, they proceed with the second half ($L_{\mathsf{max}/2+1}$ to L_{max}). In this way, they recursively operate on the appropriate half that has inconsistent hash to identify L_p, L_{p+1}. Note that one is guaranteed to identify such a L_p, L_{p+1} since the above recursion would terminate and, at least at the first level in the circuit, is guaranteed to be consistent and correct, owing to the correctness of the input sharing.

On identifying levels L_p and L_{p+1}, P_1, P_2 Jmp-sv all the βs up to level L_p to P_4, P_5. This is followed by the deferred Jmp-vrfy towards P_1, P_2, P_3 for all βs up to level L_{p+1}. If no CP is identified during any of the *verify* phases, it implies case (ii), and hence, honest online parties will be guaranteed to be in agreement with respect to the *correct* βs up to level L_{p+1}. Thus, the correct hash that should have been sent at level L_{p+1} in the binary search can be computed locally (using the β) and matched against the hash received from others. This determines the corrupt party that deliberately sent an incorrect hash. This corrupt party, together with another honest party, is identified as a CP. Note that the binary search can terminate with the last level being identified as L_p. This happens when circuit evaluation is correct, but the malicious party deliberately sends an incorrect hash in consistency check and behaves honestly

in binary search. L_p being the last level implies that honest parties are guaranteed to be in agreement with respect to *all* βs. Hence, a corrupt party can be identified as the party who sent the incorrect hash in the consistency check. Similar to the above, the CP can thus be formed.

5.6.2 Generating $[\alpha_{\mathsf{ab}}]$

Since $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ are available in the preprocessing phase, $[\alpha_{\mathsf{ab}}]$ can be computed there. For obtaining $[\alpha_{\mathsf{ab}}]$, we rely on a robust $(1,1)$-FaF secure multiplication protocol for 5PC which works on $[\cdot]$-shares (RSS shares), and is abstracted out as a functionality, $\mathcal{F}_{\mathsf{MulPre}}$, in Fig. 5.15. To leverage amortization, we preprocess several multiplication triples in a single shot. Hence, $\mathcal{F}_{\mathsf{MulPre}}$ is defined with respect to several triples. We instantiate $\mathcal{F}_{\mathsf{MulPre}}$ using a variant of the protocol of [31] for the 5-party setting. Similar to the original protocol, the modified protocol involves performing a 5PC semi-honest multiplication followed by a verification phase to check the correctness of the semi-honest execution. The difference lies in the steps performed when verification fails, and it outputs a pair of conflicting parties. In such a case, we eliminate the pair of parties, and the computation proceeds via semi-honest 3PC,unlike the malicious 3PC used in the original protocol. The verification phase relies on distributed zero-knowledge proof system [29] and is designed such that its communication cost gets amortized over multiple instances of multiplication. Thus, amortized communication cost of this 5PC protocol is same as that of the semi-honest protocol. [31] is secure as per standard security definition. We prove that the modified variant, for 5PC, is secure in the $(1,1)$-FaF model in §5.6.3. Our multiplication protocol appears in Fig. 5.8.

Protocol $\Pi_{\mathsf{Mul}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!])$

Preprocessing: Non-interactively generate $[\cdot]$-shares of a random $\alpha_{\mathsf{z}} \in \mathbb{Z}_{2^\ell}$, using the shared-key setup. Invoke $\mathcal{F}_{\mathsf{MulPre}}$ on $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ (Fig. 5.15) to generate $[\alpha_{\mathsf{ab}}]$.

Online:

- Compute $[\beta'] = -\beta_{\mathsf{a}}[\alpha_{\mathsf{b}}] - \beta_{\mathsf{b}}[\alpha_{\mathsf{a}}] + [\alpha_{\mathsf{ab}}] + [\alpha_{\mathsf{z}}]$.
- Send missing $[\beta']$-shares to P_1, P_2, P_3: (a) P_2, P_5 Jmp-send ${\beta'}_{13} + {\beta'}_{14}$ to P_1, while P_3, P_4 Jmp-send ${\beta'}_{12} + {\beta'}_{15}$ to P_1, (b) P_1, P_5 Jmp-send ${\beta'}_{23} + {\beta'}_{24}$ to P_2, while P_3, P_4 Jmp-send ${\beta'}_{12} + {\beta'}_{25}$ to P_2, (c) P_1, P_4 Jmp-send ${\beta'}_{23} + {\beta'}_{35}$ to P_3, while P_2, P_5 Jmp-send ${\beta'}_{13} + {\beta'}_{34}$ to P_3.
- P_1, P_2, P_3 reconstruct β' and compute $\beta_{\mathsf{z}} = \beta' + \beta_{\mathsf{ab}}$.

One-time Verification (for entire circuit): Follow the steps described in Fig. 5.9.

Figure 5.8: Multiplication.

Let M be the set of all β_{z}s where each z is the output of multiplication in the circuit. Parties do the following.

– $P_i \in \{P_1, P_2, P_3\}$ computes the hash, $\mathsf{H}_i = \mathsf{H}\left(\beta_{\mathsf{z}_1}, \ldots, \beta_{\mathsf{z}_{|\mathsf{M}|}}\right)$ where $\beta_{\mathsf{z}_j} \in \mathsf{M}$ and mutually exchange it among themselves.

– $P_i \in \{P_1, P_2, P_3\}$ broadcasts an inconsistency bit b to indicate whether all the obtained hashes are consistent ($\mathsf{b} = 0$) or not ($\mathsf{b} = 1$).

– If all parties in $\{P_1, P_2, P_3\}$ broadcast $\mathsf{b} = 0$, then–(a) P_1, P_2 Jmp-sv all $\beta_{\mathsf{z}} \in \mathsf{M}$ to P_4, P_5. (b) If this Jmp-sv succeeds, (i.e., no CP is identified), then parties perform the deferred Jmp-vrfy with with respect to all $\beta_{\mathsf{z}} \in \mathsf{M}$.

– Else, if some $P_i \in \{P_1, P_2, P_3\}$ broadcasts $\mathsf{b} = 1$, then

- Each $P_i \in \{P_1, P_2, P_3\}$ broadcasts H_i.
- If for any $P_i \in \{P_1, P_2, P_3\}$, the hash sent via broadcast does not match the hash received on point-to-point communication by some P_j, then P_i broadcasts its complaint against P_j. Parties set $\mathsf{CP} = (P_i, P_j)$ where P_i is party with the least index that complained, and terminate.
- If a CP was not identified via a complaint, then

  - Let $\mathsf{H}_i^{L_s}$ denote the hash computed by $P_i \in \{P_1, P_2, P_3\}$ on all β_{z} up to level L_s in circuit.

  - P_1, P_2, P_3 perform a binary search to identify a pair of consecutive levels L_p, L_{p+1} in the circuit such that $\mathsf{H}_i^{L_p}$ is consistent, but $\mathsf{H}_i^{L_{p+1}}$ is inconsistent.

  - P_1, P_2 Jmp-sv β_{z} up to level L_p to P_4, P_5.

  If the Jmp-sv is a success, then parties perform deferred Jmp-vrfy with respect to all β_{z} up to level L_{p+1}. If the Jmp-vrfy is a success, P_i matches its hash $\mathsf{H}_i^{L_{p+1}}$ against the hashes received to identify the party that sent an incorrect $\mathsf{H}^{L_{p+1}}$. P_i broadcasts the identity of this corrupted party $P^\star$ to all parties in $\mathcal{P}$. All parties set $\mathsf{CP} = (P_i, P^\star)$ where P_i is the party with the least index.

  - If L_p is the same as the last level in the segment, then $P_i \in \{P_1, P_2, P_3\}$ matches its hash H_i against the hashes received to identify the party that sent an incorrect H in the first consistency check. P_i broadcasts the identity of this corrupted party $P^\star$ to all parties in $\mathcal{P}$. All parties set $\mathsf{CP} = (P_i, P^\star)$ where P_i is the party with the least index.

Figure 5.9: One-time Verification (for entire circuit).

To showcase all the cases handled and improve the readability of our algorithm, we also provide a flowchart of the verification phase in Fig. 5.10. The green arrows denote the steps that lead to successful circuit evaluation and also showcase the correctness of our protocol. The flow where one of the offline parties is malicious is trivial and follows from the verify of jmp towards parties P_1, P_2, P_3 in the verification phase.

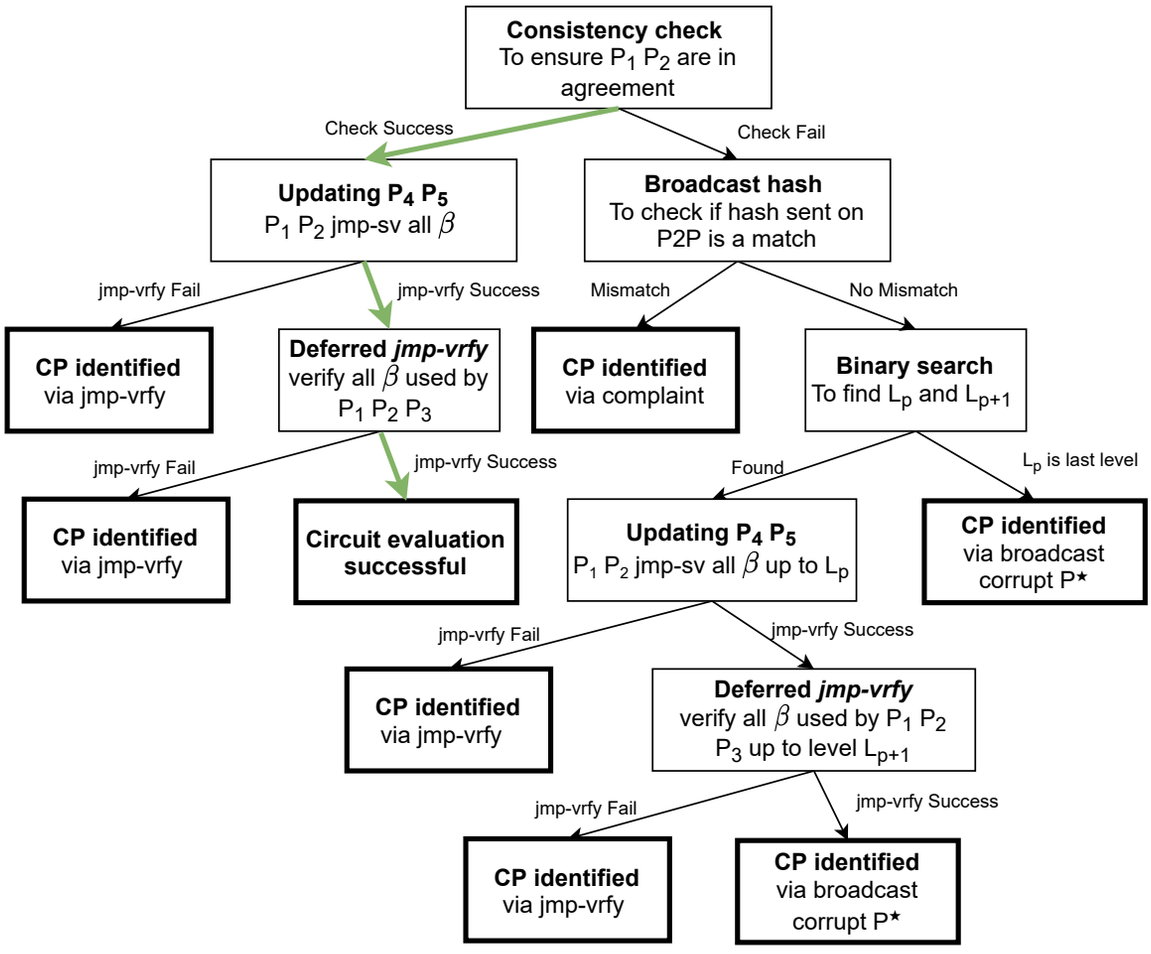


Figure 5.10: Flow of verification phase when the online party is malicious.

5.6.3 Preprocessing phase of multiplication

We next discuss the protocol carried out in the preprocessing phase to perform multiplication. The protocol is similar to the one proposed in [31], where first, the semi-honest protocol is executed, followed by verifying the correctness of the semi-honest execution. The difference lies in the steps performed when the verification fails, and a pair of conflicting parties is output. In such a case, owing to the presence of at most one malicious party in our setting, we eliminate the pair of parties in conflict, and the computation proceeds via *semi-honest* 3PC, unlike the malicious 3PC used in the original protocol. Further, we do not require the use of tags (or message authentication codes) to ensure a consistent share conversion, due to the presence of only a single malicious party. The verification protocol has a communication cost which is sublinear in the number of multiplication triples to be verified, and thus, its cost can be amortized away for multiple multiplications. Thus, the cost of the preprocessing phase boils

down to the cost of the semi-honest 5PC protocol, which is 6 ring elements. While the protocol of [31] is proven to be secure according to the standard security definition, we prove that the variant described above is $(1,1)$-FaF secure in the 5PC setting. We provide the details of the protocol here (mostly follows from [31]) for ease of understanding of the proof.

The verification of the semi-honest execution can be reduced to the problem of verifying the correctness of multiplications (several degree-2 equations). We begin with discussing the protocol for verifying the correctness of degree-2 equations (realized by the ideal functionality $\mathcal{F}_{\mathsf{CheatIdentify}}$). This protocol serves as the basis for the verification protocol (realized by the ideal functionality $\mathcal{F}_{\mathsf{Verify}}$), which is discussed subsequently. The verification protocol relies on 5 invocations (one for each party in $\mathcal{P}$) of $\mathcal{F}_{\mathsf{CheatIdentify}}$ to verify the correctness of the multiplication triples. Due to the top-down approach of explaining the functionalities, the use of $\mathcal{F}_{\mathsf{CheatIdentify}}$ may not be evident until the details of $\mathcal{F}_{\mathsf{Verify}}$ are described. We request a reader to read §5.6.3.1 as an independent section. Finally, we discuss the main protocol Π_{MulPre}, which involves executing a semi-honest 5PC protocol followed by an invocation of $\mathcal{F}_{\mathsf{Verify}}$. We prove that these protocols are $(1,1)$-FaF secure in the 5PC setting in §5.12.

5.6.3.1 Checking correctness of degree-2 relations

We first discuss a protocol that allows parties to prove the correctness of a degree-2 computation carried out on their shares. The protocol follows along the lines of the protocol in [31] and we demonstrate that it is secure in the $(1,1)$-FaF model for 5PC. We begin with the protocol for fields and discuss how it can be extended to work over rings, as shown in [31]. Specifically, party P_i wants to prove the correctness of the following equation:

$$c - \sum_{k=1}^{L} (\boldsymbol{a}_k \diamond \boldsymbol{b}_k) = 0 \tag{5.1}$$

where $c, \{\boldsymbol{a}_k\}_{k=1}^{L}, \{\boldsymbol{b}_k\}_{k=1}^{L}$ are known to P_i and $[\cdot]$-shared among parties in $\mathcal{P}$. Further, we assume that P_i knows all $[\cdot]$-shares of c. Looking ahead, $\{\boldsymbol{a}_k\}_{k=1}^{L}, \{\boldsymbol{b}_k\}_{k=1}^{L}$ represent P_i's $[\cdot]$-shares of $\{a_k\}_{k=1}^{L}, \{b_k\}_{k=1}^{L}$, while c represents P_i's additive share ($\langle\cdot\rangle$-share) of $\sum_{k=1}^{L} \mathsf{a}_k \cdot \mathsf{b}_k$ obtained by operating on its shares $\{\boldsymbol{a}_k\}_{k=1}^{L}, \{\boldsymbol{b}_k\}_{k=1}^{L}$, which is denoted by the operation $\diamond$[1]. Note here that we abuse the vector notation to mean $[\cdot]$-sharing. By virtue of $[\cdot]$-sharing, given $[\cdot]$-sharing of $\{\mathsf{a}_k, \mathsf{b}_k\}_{k=1}^{L}$ (which will be the case in the final protocol), parties can locally

[1] $[a]$ consists of 10 shares $\{a_{1,2}, a_{1,3}, \ldots, a_{4,5}\}$. Similar is the case with $[b]$. The product $c = a \cdot b$ can thus be written as the sum of products of the form $a_{i,j}b_{k,l}$ $\forall 1 \leq i \leq j \leq 5$ and $1 \leq k \leq l \leq 5$. Thus, the additive shares of c can be obtained by splitting each term $a_{i,j}b_{k,l}$ contributed by some party who has both shares. This operation of obtaining additive shares of c using local shares of a, b is captured by the $\diamond$ operator.

generate $[\cdot]$-sharing of the $[\cdot]$-shares $(\{\boldsymbol{a}_k\}_{k=1}^L, \{\boldsymbol{b}_k\}_{k=1}^L)$ held by P_i. This holds because for every share held by P_i, 2 other parties also possess it. Hence, it is possible to define a sharing where the share of one subset of 3 parties is P_i's share itself, while the other shares are 0. For instance, if v is $[\cdot]$-shared and $\boldsymbol{v} = (\mathsf{v}_{23}, \mathsf{v}_{24}, \mathsf{v}_{25}, \mathsf{v}_{34}, \mathsf{v}_{35}, \mathsf{v}_{45})$ denote the tuple of shares held by P_1 (where the subscript denotes the pair of parties which does not possess this share), then $[\boldsymbol{v}] = ([\mathsf{v}_{23}], [\mathsf{v}_{24}], [\mathsf{v}_{25}], [\mathsf{v}_{34}], [\mathsf{v}_{35}], [\mathsf{v}_{45}])$, where $[\mathsf{v}_{jk}]$ is generated by setting all but one of its shares as 0, and the non-zero share being v_{jk} (which is held by all 3 parties in $\mathcal{P} \setminus \{P_j, P_k\}$).

Relying on the distributed zero-knowledge proof system from [29] allows to prove the correctness of Equation (5.1) with sublinear communication complexity. Note that in the scenario that the proof is rejected due to one of the parties' misbehaviour, the prover will be able to identify the cheating party. In this case, the prover, together with this party, are regarded as a pair of conflicting parties, one of which is guaranteed to be corrupt. This is captured by the ideal functionality $\mathcal{F}_{\mathsf{CheatIdentify}}$, which checks for correctness of Equation (5.1) and either outputs an accept, or a pair of parties that are in conflict with each other (one among which is guaranteed to be corrupt). The functionality is defined in Fig. 5.11.

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

Let $\mathcal{S}_{\mathcal{A}}$ be an ideal world malicious adversary and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world, semi-honest adversary. Let honest parties hold consistent $[\cdot]$-sharings $[c], \{[\boldsymbol{a}_k]\}_{k=1}^L, \{[\boldsymbol{b}_k]\}_{k=1}^L$. The functionality is invoked by an index i sent by honest parties and works as follows.

1. $\mathcal{F}_{\mathsf{CheatIdentify}}$ receives from honest parties their shares of $c, \{\boldsymbol{a}_k\}_{k=1}^L, \{\boldsymbol{b}_k\}_{k=1}^L$.
2. $\mathcal{F}_{\mathsf{CheatIdentify}}$ computes $c, \{\boldsymbol{a}_k\}_{k=1}^L, \{\boldsymbol{b}_k\}_{k=1}^L$. It computes the corrupted party's shares of these values and sends them to $\mathcal{S}_{\mathcal{A}}$. If P_i is corrupted, then it also sends $[\cdot]$-shares of c, and $\{\boldsymbol{a}_k\}_{k=1}^L, \{\boldsymbol{b}_k\}_{k=1}^L$ to $\mathcal{S}_{\mathcal{A}}$. $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends $P_{\mathcal{H}}$'s shares of $c, \{\boldsymbol{a}_k\}_{k=1}^L, \{\boldsymbol{b}_k\}_{k=1}^L$ to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$, where $P_{\mathcal{H}}$ is controlled by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.
3. $\mathcal{F}_{\mathsf{CheatIdentify}}$ checks that Equation (5.1) holds.
   - If it holds then $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends accept to $\mathcal{S}_{\mathcal{A}}$, and receives $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$ from it. $\mathcal{F}_{\mathsf{CheatIdentify}}$ forwards out to honest parties.
   - If it does not hold then $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends reject to honest parties.
4. If honest parties received reject:
   - If P_i is corrupt, $\mathcal{S}_{\mathcal{A}}$ sends an index $j \in \{1, 2, \ldots, 5\}$ to $\mathcal{F}_{\mathsf{CheatIdentify}}$.
   - If P_i is honest, $\mathcal{S}_{\mathcal{A}}$ sends an index $j \in \{1, 2, \ldots, 5\}$ to $\mathcal{F}_{\mathsf{CheatIdentify}}$, where P_j is corrupt.
   - $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends the pair (i, j) to honest parties.
5. $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.11: Ideal functionality for proving correctness of degree-2 equation by prover P_i.

The concrete protocol for $\mathcal{F}_{\mathsf{CheatIdentify}}$ We begin with a high-level idea of the protocol. Given a g-gate which is defined as follows:

$$g(\boldsymbol{v_1}, \ldots, \boldsymbol{v_L}) = \sum_{l=1}^{L/2} \boldsymbol{v_{2l-1}} \diamond \boldsymbol{v_{2l}}$$

where $\diamond$ denotes the operation of obtaining additive shares of $\mathsf{v}_i \cdot \mathsf{v}_j$ given their $[\cdot]$-shares $(\boldsymbol{v_i}, \boldsymbol{v_j})$. Equation (5.1) can be written as: $c - g(\boldsymbol{a_1}, \boldsymbol{b_1}, \ldots, \boldsymbol{a_{L/2}}, \boldsymbol{b_{L/2}}) - g(\boldsymbol{a_{L/2+1}}, \boldsymbol{b_{L/2+1}}, \ldots, \boldsymbol{a_L}, \boldsymbol{b_L}) = 0$. The prover, knowing all inputs, can compute the output of the two g-gates and $[\cdot]$-share them among parties in $\mathcal{P}$. Let $g_1 = g(\boldsymbol{a_1}, \boldsymbol{b_1}, \ldots, \boldsymbol{a_{L/2}}, \boldsymbol{b_{L/2}})$ and $g_2 = g(\boldsymbol{a_{L/2+1}}, \boldsymbol{b_{L/2+1}}, \ldots, \boldsymbol{a_L}, \boldsymbol{b_L})$. Thus, parties can compute $[b] = [c] - [g_1] - [g_2]$ and check if $b = 0$ by reconstructing b. To ensure that a corrupt P_i did not cheat while generating $[\cdot]$-shares of g_1, g_2, parties perform an additional test. For this, parties define polynomials $\boldsymbol{f_1}, \ldots, \boldsymbol{f_L}$ as follows: for each $e \in \{1, 2, \ldots, L\}$, $\boldsymbol{f}_e(1)$ is the eth input vector to the 1st g-gate and $\boldsymbol{f}_e(2)$ is the eth input vector to the 2nd g-gate. $\boldsymbol{f}_e$ is thus a linear function. Next, define polynomial $q(x) = g(\boldsymbol{f}_1(x), \ldots, \boldsymbol{f}_L(x))$. Thus, $q(1), q(2)$ are the outputs of the first and second g-gate, respectively, and q is of degree 2 (since the multiplicative depth of g-gate is 1 and degree of $\boldsymbol{f}_e$ is 1). To ensure that P_i shared the correct $g(1), g(2)$, it suffices for the parties to check if $q(\mathsf{r}) = g(\boldsymbol{f}_1(\mathsf{r}), \ldots, \boldsymbol{f}_L(\mathsf{r}))$ for a random r in the ring/field. For this, parties compute $[\cdot]$-shares of $q(\mathsf{r}), \boldsymbol{f}_1(\mathsf{r}), \ldots, \boldsymbol{f}_L(\mathsf{r})$ via Lagrange interpolation on their local shares and check for the equality on clear. This also requires P_i to share $q(3)$ so that parties have sufficient points on q. To reduce the cost from L shares which is linear in L, to logarithmic in L, P_i is made to prove that

$$q(\mathsf{r}) - g(\boldsymbol{f}_1(\mathsf{r}), \ldots, \boldsymbol{f}_L(\mathsf{r})) = 0 \tag{5.2}$$

by repeating the same process (since Equation (5.2) has the same form as that of Equation (5.1)). Parties repeat the process $\log L$ times until a constant number of inputs are left, which are verified on clear. Since $\boldsymbol{f}_e(\mathsf{r})$ is a linear combination of the inputs, to avoid leaking any information about the inputs, in the final step, the $\boldsymbol{f}$ polynomials are randomized by adding one additional random point one each polynomial. This increases the degree of $\boldsymbol{f}$ to 2 and that of q to 4, and requires P_i to generate and share additional points on q. In case parties reject the proof, the prover is asked to identify the cheating party. The pair of parties, including the prover and the party identified by the prover, are then regarded as the corrupted pair of parties. For this, observe that every message sent by a party other than the prover is a function of (i) the messages received from the prover, (ii) the inputs to the protocol, and (iii) the randomness used. Since the prover knows all these, it can compute the message that should have been sent by other parties and identify inconsistencies, if any. Formal protocol steps appear in Fig. 5.12.

Protocol $\Pi_{\mathsf{CheatIdentify}}\left(\mathcal{P}, P_i, [c], \{[\boldsymbol{a}]_k\}_{k=1}^{L}, \{[\boldsymbol{b}]_k\}_{k=1}^{L}\right)$

1. Parties set $\bar{L} = L$ and for $l = 1$ to $\log \bar{L} - 1$ do the following:
   - Parties define linear polynomials $\boldsymbol{f}_1, \boldsymbol{f}_2, \ldots, \boldsymbol{f}_L$ such that for each $e \in \{1, 2, \ldots, L\}$, polynomial $\boldsymbol{f}_e$ is defined by the following two points:

$$\boldsymbol{f}_e(1) = \begin{cases} \boldsymbol{a}_{\lceil e/2 \rceil} & \text{if } e \bmod 2 = 1 \\ \boldsymbol{b}_{e/2} & \text{if } e \bmod 2 = 0 \end{cases} \quad \boldsymbol{f}_e(2) = \begin{cases} \boldsymbol{a}_{L/2+\lceil e/2 \rceil} & \text{if } e \bmod 2 = 1 \\ \boldsymbol{b}_{L/2+e/2} & \text{if } e \bmod 2 = 0 \end{cases}$$

   - Let $q(x) = g(\boldsymbol{f}_1(x), \boldsymbol{f}_2(x), \ldots, \boldsymbol{f}_L(x))$ be a degree-2 polynomial where

$$g(\boldsymbol{f}_1(x), \boldsymbol{f}_2(x), \ldots, \boldsymbol{f}_L(x)) = \sum_{j=1}^{L/2} \boldsymbol{f}_{2j-1}(x) \diamond \boldsymbol{f}_{2j}(x)$$

   P_i computes $q(1), q(2), q(3)$ and shares them among parties in $\mathcal{P}$ (via the field equivalent of $\Pi_{\mathsf{RSS-Sh}}$ protocol in Fig. 5.5).
   - Parties locally compute $[b_l] = [c] - [q(1)] - [q(2)]$ and store the result.
   - Parties generate a random $\mathsf{r} \in \mathbb{F}$ non-interactively using their shared key setup.
   - Parties locally compute $[q(\mathsf{r})]$ and $[\boldsymbol{f}_1(\mathsf{r})], [\boldsymbol{f}_2(\mathsf{r})], \ldots, [\boldsymbol{f}_L(\mathsf{r})]$ via Lagrange interpolation.
   - Parties set $c \leftarrow q(\mathsf{r})$, and $\forall k \in \{1, 2, \ldots, L/2\} : \boldsymbol{a}_k \leftarrow \boldsymbol{f}_{2k-1}(\mathsf{r})$, $\boldsymbol{b}_k \leftarrow \boldsymbol{f}_{2k}(\mathsf{r})$ and $L \leftarrow L/2$.

2. Parties exit the loop with $L = 2$ and inputs $c, \boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{b}_1, \boldsymbol{b}_2$ that are known to P_i and secret shared among other parties. Next,
   - Parties non-interactively generate $[\boldsymbol{w}_1], [\boldsymbol{w}_2]$ where $\boldsymbol{w}_1, \boldsymbol{w}_2 \in \mathbb{F}^{\mathsf{g}}$ are known to P_i. Here, g is the number of components in $[\cdot]$-sharing held by each party. Parties define polynomials $\boldsymbol{f}_1, \boldsymbol{f}_2$ of degree 2 such that $\boldsymbol{f}_1(0) = \boldsymbol{w}_1, \boldsymbol{f}_1(1) = \boldsymbol{a}_1, \boldsymbol{f}_1(2) = \boldsymbol{a}_2$, $\boldsymbol{f}_2(0) = \boldsymbol{w}_2, \boldsymbol{f}_2(1) = \boldsymbol{b}_1, \boldsymbol{f}_2(2) = \boldsymbol{b}_2$.
   - P_i defines the degree-4 polynomial $q(x) = g(\boldsymbol{f}_1(x), \boldsymbol{f}_2(x))$ where $g(\boldsymbol{f}_1(x), \boldsymbol{f}_2(x)) = \boldsymbol{f}_1(x) \diamond \boldsymbol{f}_2(x)$, and computes $q(0), q(1), \ldots, q(4)$.
   - P_i shares $q(0), q(1), \ldots, q(4)$ among parties in $\mathcal{P}$ (via $\Pi_{\mathsf{RSS-Sh}}$ protocol in Fig. 5.5).
   - Parties locally compute $[b_{\log L}] = [c] - [q(1)] - [q(2)]$.
   - Parties non-interactively generate $\mathsf{r}, \gamma_1, \ldots, \gamma_{\log L} \in \mathbb{F}$, and compute $[b] = \sum_{l=1}^{\log L} \gamma_l \cdot [b_l]$.
   - Parties locally compute $[\boldsymbol{f}_1(\mathsf{r})], [\boldsymbol{f}_2(\mathsf{r})]$ and $[q(\mathsf{r})]$ via Lagrange interpolation.
   - Parties reconstruct $b, q(\mathsf{r}), \boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r})$ towards each party where each missing share is broadcast. If reconstruction has an inconsistency, or if $q(\mathsf{r}) \neq g(\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}))$ of if $b \neq 0$, then parties output reject. Else, parties output accept.
   - If parties output reject, P_i identifies a party P_j who sent incorrect messages in the previous step, and broadcasts j to all the parties. Parties output the conflict pair (i, j).

Figure 5.12: Realizing $\mathcal{F}_{\mathsf{CheatIdentify}}$.

Cheating probability over finite fields There are two cases which lead to the parties outputting accept even when Equation (5.1) does not hold—(i) the linear combination of the b values yields a 0, and (ii) when P_i cheats during sharing points on q and thus $q \neq g(\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L)$ and $h(x) = q(x) - g(\boldsymbol{f}_1(x), \ldots, \boldsymbol{f}_L(x))$ is a non-zero polynomial. While (i) happens with probability $\frac{1}{\mathbb{F}}$, for (ii), the probability that $h(\mathsf{r}) = 0$ for a random $\mathsf{r} \in \mathbb{F} \setminus \{1, 2, 3\}$ is bounded by $\frac{2}{\mathbb{F}-2}$ (since degree of polynomial h is 2) in the first $\log L - 1$ rounds and $\frac{4}{\mathbb{F}-5}$ in the last round (since degree of h is now 4). Thus, the overall cheating probability is bounded by

$$\frac{2(\log L - 1)}{\mathbb{F} - 3} + \frac{4}{\mathbb{F} - 5} < \frac{2\log L + 4}{\mathbb{F} - 5}$$

Extension to rings While the protocol described works over fields, using the extension techniques from [29, 30, 31], the protocol can be extended to work over rings. The challenge lies in performing interpolation where not all elements have an inverse over the ring $\mathbb{Z}_{2^\ell}$. To overcome this, the solution is to work over the extended ring $\mathbb{Z}_{2^\ell}[x]/f(x)$, i.e. the ring of all polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ working modulo a polynomial f that is of the right degree and irreducible over $\mathbb{Z}_2$. When working over these extension rings, the number of roots of a polynomial is greater than its degree and thus changes the error probability. For a protocol which verifies L values, the error is roughly $\frac{2\log L+4}{2^d}$, where d is the extension degree. We refer readers to [30, 29] for more details.

Communication cost In the first $\log L - 1$ iterations, the prover shares 3 elements each. In the last round, it shares 5 elements, followed by public reconstruction of 4 elements via broadcast. Generation of randomness can be done non-interactively and does not incur any cost. Thus, the total communication cost is

$$6(\log L - 1) + 10 + 7 \text{ elements.}$$

Thus, the per-party cost is approximately $\log L + 3$ elements.

5.6.3.2 The verification protocol

Using $\mathcal{F}_{\mathsf{CheatIdentify}}$, we next provide the protocol for verification of m multiplication triples with sublinear communication complexity in the number of multiplication triples. A multiplication triple is a shared triple $[\mathsf{x}], [\mathsf{y}], [\mathsf{z}]$ such that $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$. The ideal functionality for the same appears in Fig. 5.13. When verification fails, the functionality either obtains a pair of conflicting parties, one of which is guaranteed to be corrupt, from the adversary, or it identifies this pair by itself.

In the latter case, the functionality obtains the inputs, randomness and views of honest parties when computing some incorrect multiplication triple, and uses this information to identify a pair of conflicting parties.

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

Let $\mathcal{S}_{\mathcal{A}}$ be an ideal world malicious adversary and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world, semi-honest adversary. The functionality is invoked by honest parties sending their $[\cdot]$-shares of m multiplication triples $\{(\mathsf{x}_k, \mathsf{y}_k, \mathsf{z}_k)_{k=1}^m\}$ to $\mathcal{F}_{\mathsf{Verify}}$.

1. $\mathcal{F}_{\mathsf{Verify}}$ computes all secrets and corrupted party's shares and sends these shares to $\mathcal{S}_{\mathcal{A}}$. $\mathcal{F}_{\mathsf{Verify}}$ sends $P_{\mathcal{H}}$'s shares to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$, where $P_{\mathcal{H}}$ is controlled by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

2. $\mathcal{F}_{\mathsf{Verify}}$ verifies if $\mathsf{z}_k = \mathsf{x}_k \cdot \mathsf{y}_k$ for all $k \in \{1, 2, \ldots, m\}$.
   - If it holds, it sends accept to $\mathcal{S}_{\mathcal{A}}$.
   - Else, it sends reject to $\mathcal{S}_{\mathcal{A}}$ and $\mathsf{d}_k = \mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k$ for each $k \in \{1, 2, \ldots, m\}$ such that $\mathsf{d}_k \neq 0$.

3. If $\mathcal{F}_{\mathsf{Verify}}$ sent accept, it receives $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$ from $\mathcal{S}_{\mathcal{A}}$, which is forwarded to the honest parties and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.
   - If $\mathsf{out} = \mathsf{reject}$, $\mathcal{S}_{\mathcal{A}}$ send a pair of indices (i, j) to $\mathcal{F}_{\mathsf{Verify}}$, where at least one among P_i, P_j is corrupt.
   - $\mathcal{F}_{\mathsf{Verify}}$ forwards (i, j) to honest parties and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

4. If $\mathcal{F}_{\mathsf{Verify}}$ sent reject, then $\mathcal{S}_{\mathcal{A}}$ does one of the following.

   (1) $\mathcal{S}_{\mathcal{A}}$ sends a pair of indices (i, j) to $\mathcal{F}_{\mathsf{Verify}}$, where at least one among P_i, P_j is corrupt. $\mathcal{F}_{\mathsf{Verify}}$ forwards (i, j) to honest parties and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

   (2) $\mathcal{S}_{\mathcal{A}}$ asks $\mathcal{F}_{\mathsf{Verify}}$ to find a pair of conflicting parties in $\bar{k}^{\text{th}}$ multiplication, $1 \leq \bar{k} \leq m$. Next, $\mathcal{F}_{\mathsf{Verify}}$ asks the honest parties to send their inputs, randomness and views in the execution to compute the $\bar{k}^{\text{th}}$ triple. Based on the received information, $\mathcal{F}_{\mathsf{Verify}}$ computes the messages that should have been sent by the corrupted party and finds a pair of parties P_i, P_j, where P_j received an incorrect message. $\mathcal{F}_{\mathsf{Verify}}$ sends (i, j) to honest parties, $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

5. $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.13: Ideal functionality for verifying semi-honest protocol.

The protocol for $\mathcal{F}_{\mathsf{Verify}}$ To compute the functionality, the parties take a random linear combination

$$\beta = \sum_{k=1}^{m} \theta_k \cdot (\mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k)$$

where θ_k is randomly chosen by all the parties and want to check if $\beta = 0$. Since β is a degree-2 function of $\{(\mathsf{x}_k, \mathsf{y}_k, \mathsf{z}_k)_{k=1}^m\}$ which are $[\cdot]$-shared, parties can compute an additive sharing ($\langle\cdot\rangle$-sharing) of β. Using the $\langle\cdot\rangle$-shares, parties can reconstruct β and check for equality with 0. However, since $\langle\cdot\rangle$-sharing does not allow for robust reconstruction, the parties first $[\cdot]$-share their additive shares of $\psi = \sum_{k=1}^m \theta_k \cdot \mathsf{x}_k \cdot \mathsf{y}_k$. Let ψ^i denote the additive share of ψ held by P_i. The consistency check in the $[\cdot]$-sharing protocol ensures that all receive consistent $[\cdot]$-shares of ψ^i. In case of a failure, the dealer broadcasts the share for which pairwise inconsistency exits. Given $[\psi^i]$ for $i \in \{1, \ldots, 5\}$, parties can compute

$$[\beta] = \sum_{k=1}^{m} \theta_k \cdot [\mathsf{z}_k] - \sum_{i=1}^{5} [\psi^i]$$

and reconstruct β. It is, however, required to ensure that every party P_i shares the correct value ψ^i. Towards realizing this, the property of $[\cdot]$-sharing, which allows parties to locally convert from $[\mathsf{x}_k], [\mathsf{y}_k]$ to $[\boldsymbol{x}_k^i], [\boldsymbol{y}_k^i]$, where $\boldsymbol{x}_k^i, \boldsymbol{y}_k^i$ are the vector of $[\cdot]$-shares of $\mathsf{x}_k, \mathsf{y}_k$, held by P_i, respectively, is used. Parties now want to verify if

$$\forall i \in \{1, \ldots, 5\} : \sum_{k=1}^{m} \theta_k \cdot \left([\boldsymbol{x}_k^i] \diamond [\boldsymbol{y}_k^i] \right) - [\psi^i] = 0$$

Letting $[c^i] = [\psi^i], [\boldsymbol{a}_k^i] = \theta_k \cdot [\boldsymbol{x}_k^i]$ and $[\boldsymbol{b}_k^i] = [\boldsymbol{y}_k^i]$, one needs to verify that $[c^i] - \sum_{k=1}^m [\boldsymbol{a}_k^i] \diamond [\boldsymbol{b}_k^i] = 0$ for $i \in \{1, \ldots, 5\}$. This can be verified using $\mathcal{F}_{\mathsf{CheatIdentify}}$. In case of a reject, $\mathcal{F}_{\mathsf{CheatIdentify}}$ outputs a pair of conflicting parties. Otherwise, parties proceed with reconstructing β. If reconstruction fails due to inconsistency, a pairwise consistency check of $[\cdot]$-sharing is used to identify a pair of conflicting parties, where the consistency check is carried out over a broadcast channel. Finally, if $\beta \neq 0$, then it implies that no one cheated in the verification protocol (with high probability), and one of the multiplication triples is incorrect. Parties localize the fault by running a binary search on the multiplication triples to identify a triple where $\mathsf{z}_k \neq \mathsf{x}_k \cdot \mathsf{y}_k$. In each search step, the verification protocol is carried out on half the number of triples until one incorrect triple is identified. Finally, parties check the execution of the multiplication protocol for this triple to find a pair of disputing parties. This is done by invoking a functionality

$\mathcal{F}_{\mathsf{MiniMPC}}$, which takes the inputs, randomness and view of parties in the multiplication protocol as input and outputs the pair of parties for which the incoming and outgoing messages do not match. The protocol appears in Fig. 5.14.

Protocol $\Pi_{\mathsf{Verify}}\left(\mathcal{P}, \{([\mathsf{x}_k], [\mathsf{y}_k], [\mathsf{z}_k])\}_{k=1}^{m}\right)$

1. Parties generate random values $\theta_1, \ldots, \theta_m \in \mathbb{F}$, and locally compute

$$\langle \psi \rangle = \left\langle \sum_{k=1}^{m} \theta_k \cdot \mathsf{x}_k \cdot \mathsf{y}_k \right\rangle = \sum_{k=1}^{m} \theta_k \cdot ([\mathsf{x}_k] \diamond [\mathsf{y}_k])$$

2. Let the $\langle \cdot \rangle$-share of ψ held by P_i be ψ^i. Each party P_i shares ψ^i among other parties.

3. For each $i \in \{1, 2, \ldots, 5\}$:
   - Parties locally convert $[\mathsf{x}_k], [\mathsf{y}_k]$ to $\left[\mathbf{x}_k^i\right], \left[\mathbf{y}_k^i\right]$ for each $k \in \{1, 2, \ldots, m\}$.
   - Parties define $\left[c^i\right] = \left[\psi^i\right]$, $\left[\boldsymbol{a}_k^i\right] = \theta_k \cdot \left[\mathbf{x}_k^i\right]$ and $\left[\boldsymbol{b}_k^i\right] = \theta_k \cdot \left[\mathbf{y}_k^i\right]$.
   - Parties send $\left[c^i\right]$ and $\left(\left[\boldsymbol{a}_k^i\right], \left[\boldsymbol{b}_k^i\right]\right)_{k=1}^{m}$ to $\mathcal{F}_{\mathsf{CheatIdentify}}$.
   - If parties receive $\mathsf{reject}, (i, j)$ from $\mathcal{F}_{\mathsf{CheatIdentify}}$, then they output it and halt.

4. If parties received accept from $\mathcal{F}_{\mathsf{CheatIdentify}}$ in all five invocations, they proceed to the next step.

5. Parties locally compute $[\beta] = \sum_{k=1}^{m} \theta_k \cdot [\mathsf{z}_k] - \sum_{i=1}^{5} \left[\psi^i\right]$.

6. Parties robustly reconstruct β by sending their shares via broadcast.
   - If parties see inconsistent shares, they output $\mathsf{reject}, (i, j)$, where P_i, P_j is the first pair of parties for which pair-wise inconsistency exists.
   - If $\beta = 0$, parties output accept.
   - If $\beta \neq 0$, parties perform a fault localization procedure to identify the first incorrect triple by running a binary search on the input triples. For this search, parties run the above protocol on two half-sized sets of input triples and proceed as follows.
     - If parties output accept in both executions, they output accept and halt.
     - If any execution ends with parties holding a pair of conflicting parties (i, j), parties output $\mathsf{reject}, (i, j)$ and halt.
     - If $\beta \neq 0$ in both executions, they continue the search on one of the sets.
     - If $\beta \neq 0$ in one of the executions, they continue the search on the set for which $\beta \neq 0$.

   If parties didn't receive any output, then they reach a triple k for which $\mathsf{z}_k \neq \mathsf{x}_k \cdot \mathsf{y}_k$. Then, parties send their inputs, randomness and view when computing z_k to $\mathcal{F}_{\mathsf{MiniMPC}}$, which returns a pair of conflicting parties (i, j) with conflicting views. Parties output $\mathsf{reject}, (i, j)$.

Figure 5.14: Realizing $\mathcal{F}_{\mathsf{Verify}}$.

Cheating probability over finite fields Assume that there is an incorrect triple. If the adversary does not cheat in the verification protocol, then there will be at most $\log m$ executions. In each execution, the probability that the test will pass is $\frac{1}{\mathbb{F}}$, which happens when the random linear combination outputs a value 0. Thus, the overall cheating probability is bounded by $\log m \cdot \frac{1}{\mathbb{F}}$.

Communication cost Protocol Π_{Verify} is recursive. In the jth step, parties secret share one element each, reconstruct one element and call $\mathcal{F}_{\mathsf{CheatIdentify}}$ for every party over a set of triples of size $m/2^j$. Thus, the total communication cost in the jth step is

$$\begin{aligned} &5 \cdot 2 + 7 + 5 \cdot \big(6(\log(m/2^j) - 1) + 17\big) \\ &= 97 + 30 \cdot \log(m/2^j) \text{ elements.} \end{aligned}$$

In the worst case, there are $\log m$ steps, and the total cost is

$$97 \cdot \log m + 30 \cdot \sum_{j=1}^{\log m} \log(m/2^j)$$

Since $\sum_{j=1}^{\log m} \log(m/2^j) \leq \log m \cdot \log \sqrt{m}$, the total communication cost is

$$97 \cdot \log m + 30 \cdot \log m \cdot \log \sqrt{m} \text{ elements.} \tag{5.3}$$

Note that while working over extended rings, the cost gets multiplied by a factor d, which is the degree of the extension.

Similar to $\Pi_{\mathsf{CheatIdentify}}$, the protocol Π_{Verify} can also be extended to work over the ring $\mathbb{Z}_{2^\ell}$ (see 5.6.3.1).

5.6.3.3 The main protocol

We now provide details of the main protocol for computing the multiplication triples in the preprocessing phase. The ideal functionality for the same appears in Fig. 5.15. We remark that operating in the preprocessing model, we can generate a large number of multiplication triples at the same time, which also helps in amortizing the cost due to verification. The main protocol begins with executing a semi-honest 5PC protocol, followed by a verification phase to check the correctness of the multiplication triples generated during the semi-honest execution.

Verification completes with it either being a success or outputting a pair of conflicting parties (in which case a semi-honest 3PC is executed). The protocol appears in Fig. 5.16.

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

Let $\mathcal{S}_{\mathcal{A}}$ be an ideal world malicious adversary and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world, semi-honest adversary.

1. $\mathcal{F}_{\mathsf{MulPre}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$. $\mathcal{F}_{\mathsf{MulPre}}$ receives $[\cdot]$-shares of $\{(\mathsf{x}_k, \mathsf{y}_k)_{k=1}^m\}$ from honest parties.
2. $\mathcal{F}_{\mathsf{MulPre}}$ receives $([\mathsf{x}_k \cdot \mathsf{y}_k]_i)_{k=1}^m$ from $\mathcal{S}_{\mathcal{A}}$ where P_i is controlled by $\mathcal{S}_{\mathcal{A}}$. It also receives $\mathsf{continue}$ or $(\mathtt{abort}, j)$ from $\mathcal{S}_{\mathcal{A}}$. If received $\mathtt{abort}$, $\mathcal{F}_{\mathsf{MulPre}}$ sends (i, j) to all. Else, it does the following.
   - $\mathcal{F}_{\mathsf{MulPre}}$ reconstructs $\mathsf{x}_k, \mathsf{y}_k$ using the honest parties' shares and computes $\mathsf{x}_k \cdot \mathsf{y}_k$ for $k \in \{1, \ldots, m\}$.
   - $\mathcal{F}_{\mathsf{MulPre}}$ generates $[\mathsf{x}_k \cdot \mathsf{y}_k]$, for $k \in \{1, 2, \ldots, m\}$, using $\mathsf{x}_k \cdot \mathsf{y}_k$ and $[\mathsf{x}_k \cdot \mathsf{y}_k]_i$ received from $\mathcal{S}_{\mathcal{A}}$.
   - $\mathcal{F}_{\mathsf{MulPre}}$ sends $(\mathsf{Output}, [\mathsf{x}_k \cdot \mathsf{y}_k]_s)$ to $P_s \in \mathcal{P}$.
3. $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.15: Ideal functionality for computing multiplication triples in the preprocessing.

Protocol $\Pi_{\mathsf{MulPre}}(\mathcal{P}, \{[\mathsf{x}_k], [\mathsf{y}_k]\}_{k=1}^m)$

1. Parties generate $[\cdot]$-shares of random values $\mathsf{r}_1, \mathsf{r}_2, \ldots, \mathsf{r}_m$, non-interactively using their shared key setup. They locally convert $[\cdot]$-shares to $\langle\cdot\rangle$-shares.
2. Parties locally compute $\langle \mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k \rangle = [\mathsf{x}_k] \diamond [\mathsf{y}_k] - \langle \mathsf{r}_k \rangle$ for each $k \in \{1, 2, \ldots, m\}$ and send it to P_1.
3. P_1 reconstructs $\mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k$ for each $k \in \{1, 2, \ldots, m\}$ and generates $[\mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k]$ using $\Pi_{\mathsf{RSS-Sh}}$ (Fig. 5.5).
4. Parties compute $[\mathsf{x}_k \cdot \mathsf{y}_k] = [\mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k] + [\mathsf{r}_k]$ for $k \in \{1, 2, \ldots, m\}$.
5. Parties invoke $\Pi_{\mathsf{Verify}}(\mathcal{P}, \{([\mathsf{x}_k], [\mathsf{y}_k], [\mathsf{x}_k \cdot \mathsf{y}_k])\}_{k=1}^m)$ to verify the correctness of the multiplication triples.
6. If parties receive accept from Π_{Verify}, they proceed with the online phase. Else, parties obtain a pair of parties (P_i, P_j) to eliminate from Π_{Verify}.

Figure 5.16: $(1, 1)$-FaF secure protocol for 5PC preprocessing phase of multiplication.

Communication cost The communication cost follows from the cost of the semi-honest protocol and the cost of the verification protocol. The semi-honest protocol requires communicating 6 ring elements. The cost due to the verification phase can be amortized by preprocessing a large number of multiplication triples. Concretely, for verifying 2^{25} multiplication triples, the cost for verification is only 0.003 ring elements for an extension degree $d = 46$ (see Table 4 of full (eprint) version of [31]). Table 5.2 summarizes the communication cost for a various number of multiplication triples to be verified.

m	Cost (per party per multiplication)
2^{10}	22.1914
2^{20}	0.0696
2^{25}	0.0032
2^{30}	0.0001

Table 5.2: Cost of verification in terms of the number of ring elements communicated per party per multiplication, and 40 bits of statistical security. Here, m - #multiplication triples to be verified and degree of extension $d = 46$ to achieve statistical security of 2^{-40}.

5.7 The complete 5PC

We give an overview of the execution of 5PC for computing any function. The complete protocol can be divided into three stages: input sharing, evaluation, and output reconstruction. Each stage is further cast in the preprocessing model, which comprises a preprocessing phase and an online phase. The protocol execution is preceded by a one-time shared key setup and begins by executing the preprocessing phase for each of the three stages. Note that protocols in each of these stages rely on several invocations of Jmp. Thus, they either complete successfully or, in case of misbehaviour, a conflict pair CP is identified. To leverage amortization, only the *send* of all Jmps are run on the flow while all *verify* steps are deferred until output reconstruction. Recall that identification of CP calls for rerunning of the protocol via 3PC. Thus, deferring verification until output reconstruction would result in the worst-case cost of executing 5PC and 3PC. To avoid this, a possible optimization is to divide the computation of the circuit into segments[2], with a checkpoint placed at the end of each segment. Computation carried out in a segment can be verified at each checkpoint. In this way, if a CP is identified in any segment,

[2]A circuit is sliced depth-wise into segments comprising multiple levels/layers.

the computation of this segment restarts with a 3PC execution. For this, a share conversion is performed to convert shares from 5PC to 3PC. The details of the same are provided next. All the subsequent segments can now be evaluated via the 3PC. The complete protocol appears in Fig. 5.17 and proofs in §5.12.

Protocol $\mathsf{5PC - FaF}$

A one-time shared key setup is performed to generate common PRF keys, which can be used to generate correlated randomness.

Preprocessing Phase:

- For each input gate u, parties execute preprocessing phase of Π_{Sh} to obtain $[\alpha_{\mathsf{u}}]$.
- For each addition gate with input wires u, v and output wire w, parties locally compute $[\alpha_{\mathsf{w}}] = [\alpha_{\mathsf{u}}] + [\alpha_{\mathsf{v}}]$.
- For each multiplication gate with input wires u, v and output wire w, parties execute preprocessing phase of Π_{Mul} to obtain $[\alpha_{\mathsf{w}}], [\alpha_{\mathsf{uv}}]$.

Online Phase:

- For each input v held by a party, parties invoke the online phase of Π_{Sh} to generate $[\![\mathsf{v}]\!]$.
- For each addition gate with input wires u, v and output wire w, parties locally compute $[\![\mathsf{w}]\!] = [\![\mathsf{u}]\!] + [\![\mathsf{v}]\!]$.
- For each multiplication gate with input wires u, v and output wire w, parties execute the online phase of Π_{Mul} to generate $[\![\mathsf{w}]\!]$.
- For each output gate, parties execute Π_{Rec} to reconstruct output w towards the designated party.

Semi-honest 3PC: If a CP is identified at any step, perform share conversion and continue computation with semi-honest 3PC.

Figure 5.17: 5PC FaF Protocol.

Share conversion We describe the (3, 1) replicated secret sharing (RSS) semantics for a 3PC protocol followed by the steps for share conversion, where the latter is similar to that described in [31] optimized for our setting. Let $\mathcal{P}' = \{P'_0, P'_1, P'_2\}$ denote the three parties. Let $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1 + \mathsf{v}_2$ where $(\mathsf{v}_i, \mathsf{v}_{(i+1)\%3})$ are the shares held by P'_i that define a (3, 1) RSS scheme. Observe here that a value is split into three shares, each of which is held by two parties. Our goal is to convert from the 5PC $[\![\cdot]\!]$-sharing (which is an augmented $(5, 2)$-RSS sharing with an additional β held by all parties) to a (3, 1)-RSS sharing. The conversion proceeds as follows. Let P_i, P_j be parties to be eliminated. The residual three parties are arbitrarily assigned roles of P'_0, P'_1, P'_2. To generate the (3, 1)-RSS shares among parties in $\mathcal{P}' = \mathcal{P} \setminus \{P_i, P_j\}$, consider

the following types of shares.

1. Shares that are known to either P_i or P_j: Such shares are already held by two other parties in $\mathcal{P} \setminus \{P_i, P_j\}$, which is what is needed for the (3, 1)-RSS sharing.

2. Shares that are not known to both P_i, P_j: Such shares are known to all the three residual parties. Since exactly two parties should hold each share, we let the party with the lowest index remove this share from its possession.

3. Shares that are known to both P_i and P_j: Such shares are known to exactly one other party, say P_k, in $\mathcal{P} \setminus \{P_i, P_j\}$. To enable one other party to hold this share to complete the (3, 1)-RSS sharing, we let P_k send this share to the remainder party, say P_l.

4. Shares that are held by all (β): We let parties enacting the role of P'_1, P'_2 incorporate this share in its set of common shares, and let P'_0 remove this share from its possession.

We explain the share conversion steps with a concrete example. Let P_1, P_2 be the parties to be eliminated, and let $P'_0 = P_3, P'_1 = P_4, P'_2 = P_5$. Consider the conversion of $[\![\mathsf{v}]\!]$ to a (3,1)-RSS share. For type 1 shares, shares that are held by P_1 or P_2 include $\alpha_{\mathsf{v}_{13}}, \alpha_{\mathsf{v}_{23}}, \alpha_{\mathsf{v}_{14}}, \alpha_{\mathsf{v}_{24}}, \alpha_{\mathsf{v}_{15}}, \alpha_{\mathsf{v}_{25}}$, where every consecutive pair of shares is held by $\{P_4, P_5\}, \{P_3, P_5\}, \{P_3, P_4\}$, respectively. With respect to type 2 shares, shares that are not known to both P_1, P_2 include $\alpha_{\mathsf{v}_{12}}$. These are included by $\{P_4, P_5\}$ in their set of shares. For type 3 shares, shares that are known to both P_1, P_2 include $\alpha_{\mathsf{v}_{34}}, \alpha_{\mathsf{v}_{35}}, \alpha_{\mathsf{v}_{45}}$, which are held by P_5, P_4, P_3, respectively. Let P_3 send $\alpha_{\mathsf{v}_{45}}$ to P_4, let P_4 send $\alpha_{\mathsf{v}_{35}}$ to P_5, and let P_5 send $\alpha_{\mathsf{v}_{34}}$ to P_3. Finally, for the last type of share, we let P_4, P_5 include β_{v} in its set of shares. The (3, 1)-RSS shares of v are now defined as $\mathsf{v}_0 = -\alpha_{\mathsf{v}_{25}} - \alpha_{\mathsf{v}_{15}} - \alpha_{\mathsf{v}_{45}}$ which is held by P_3, P_4, $\mathsf{v}_2 = -\alpha_{\mathsf{v}_{24}} - \alpha_{\mathsf{v}_{14}} - \alpha_{\mathsf{v}_{34}}$ which is held by P_3, P_5, and $\mathsf{v}_1 = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}_{23}} - \alpha_{\mathsf{v}_{13}} - \alpha_{\mathsf{v}_{35}} - \alpha_{\mathsf{v}_{12}}$ which is held by P_4, P_5. This generates the (3, 1)-RSS shares of v from $[\![\mathsf{v}]\!]$.

5.8 Building blocks

In this section, we discuss 5PC $(1, 1)$-FaF realizations of building blocks (Table 5.3) required for the applications considered. Most of these are well studied in the literature [173, 194, 136, 138]. Hence, we only highlight those which were challenging to achieve in the 5PC $(1, 1)$-FaF setting.

Multi-input multiplication To reduce the online communication cost as well as the round complexity, we design protocols to enable the multiplication of 3 and 4 inputs in a single

shot [138, 194, 191]. Compared to the naive approach of performing sequential multiplications to multiply 3 and 4 inputs, the multi-input multiplication protocol enjoys the benefit of having the same online phase complexity as that of the 2-input multiplication protocol. This brings in a $2\times$ improvement in the online round complexity, while also improving the online communication cost. We extend the ideas of [138] to achieve this in our setting. For instance, the goal of 3-input multiplication is to generate $[\![\mathsf{z}]\!]$ given $[\![\cdot]\!]$-shares of $\mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathbb{Z}_{2^\ell}$ where $\mathsf{z} = \mathsf{abc}$. Observe that, $\beta_\mathsf{z} = \mathsf{abc} + \alpha_\mathsf{z} = \beta_\mathsf{abc} - \beta_\mathsf{ac}\alpha_\mathsf{b} - \beta_\mathsf{bc}\alpha_\mathsf{a} - \beta_\mathsf{ab}\alpha_\mathsf{c} + \beta_\mathsf{a}\alpha_\mathsf{bc} + \beta_\mathsf{b}\alpha_\mathsf{ac} + \beta_\mathsf{c}\alpha_\mathsf{ab} - \alpha_\mathsf{abc} + \alpha_\mathsf{z}$. Thus, parties generate $[\alpha_\mathsf{ab}], [\alpha_\mathsf{ac}], [\alpha_\mathsf{bc}], [\alpha_\mathsf{abc}]$ during preprocessing by invoking $\mathcal{F}_{\mathsf{MulPre}}$ (Fig. 5.15), and proceed with a similar online phase as in 2-input multiplication. Similarly, for 4-input multiplication $[\cdot]$-shares of $\alpha_\mathsf{ab}, \alpha_\mathsf{ac}, \alpha_\mathsf{ad}, \alpha_\mathsf{bc}, \alpha_\mathsf{bd}, \alpha_\mathsf{cd}, \alpha_\mathsf{abc}, \alpha_\mathsf{abd}, \alpha_\mathsf{acd}, \alpha_\mathsf{bcd}, \alpha_\mathsf{abcd}$ are needed.

Dot product Given $[\![\cdot]\!]$-shares of vectors $\boldsymbol{x}, \boldsymbol{y}$ where each element of the vector is $[\![\cdot]\!]$-shared, protocol Π_{DotP}, enables generation of $[\![\cdot]\!]$-shares of $\mathsf{z} = \boldsymbol{x} \odot \boldsymbol{y}$, where $\odot$ denotes the dot product operation. For this, observe that β_z can be written as

$$\beta_\mathsf{z} = \mathsf{z} + \alpha_\mathsf{z} = \boldsymbol{x} \odot \boldsymbol{y} + \alpha_\mathsf{z} = \sum_{i=1}^{n} \mathsf{x}_i\mathsf{y}_i + \alpha_\mathsf{z} = \sum_{i=1}^{\mathsf{n}} (\beta_{\mathsf{x}_i\mathsf{y}_i} - \beta_{\mathsf{x}_i}\alpha_{\mathsf{y}_i} - \beta_{\mathsf{y}_i}\alpha_{\mathsf{x}_i} + \alpha_{\mathsf{x}_i\mathsf{y}_i}) + \alpha_\mathsf{z} \tag{5.4}$$

Thus, the goal of preprocessing phase is to generate $[\cdot]$-shares of $\sigma = \sum_{i=1}^{n} \alpha_{\mathsf{x}_i\mathsf{y}_i}$, which is a dot product of $\{\alpha_{\mathsf{x}_i}\}_{i=1}^{n}, \{\alpha_{\mathsf{y}_i}\}_{i=1}^{n}$. Given $[\sigma]$, parties proceed with a similar online phase as that in multiplication to compute β_z (Equation (5.4)), where the terms are locally added before being sent, making the online communication independent of n [193, 136]. Similar to [136], to make the preprocessing communication for generating $[\sigma]$ independent of n, parties execute a semi-honest dot-product protocol [62] whose communication cost is independent of n. This is followed by a verification phase, similar to the one in [31], where parties invoke Π_{Verify}[3] (see Fig. 5.14) on the dot product triple, $[\{\alpha_{\mathsf{x}_i}\}_{i=1}^{\mathsf{n}}], [\{\alpha_{\mathsf{y}_i}\}_{i=1}^{\mathsf{n}}], [\sigma]$, to verify correctness of $[\sigma]$. As opposed to verification of m multiplication triples, which requires a communication cost of $\mathcal{O}(\log(m))$ elements, the cost for verifying the correctness of m dot products with vectors of size n now becomes $\mathcal{O}(\log(m\mathsf{n}))$ elements. Due to its similarity to multiplication, we omit formal protocol for dot product.

Matrix multiplication and convolution Matrix multiplication can easily be reduced to a dot product where each element in the resultant matrix can be computed via a dot product. Convolutions can also be reduced to matrix multiplication following standard techniques [216].

[3]Note that computations in Π_{Verify} remain unchanged except that its input parameters now correspond to dot product triples.

Truncation Repeated multiplications in fixed point arithmetic (FPA) cause an overflow. This necessitates the need for truncation, which truncates the last d bits from the result of multiplication, to retain FPA semantics. We follow a similar approach as in [174, 173] for probabilistic truncation. Here, to truncate a value v, we rely on a $(\mathsf{r}, \mathsf{r}^d)$-pair, where $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ and r^d is the truncated value of r (i.e. $\mathsf{r}^d = \mathsf{r}/2^d$). The truncated value v^d of v, is computed as $\mathsf{v}^d = (\mathsf{v} - \mathsf{r})^d + \mathsf{r}^d$.

Given $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$ can be generated in the preprocessing phase, our multiplication protocol can be modified to incorporate truncation without incurring any overhead in the online phase as follows. Use r instead of α_z while computing β_z. Parties truncate β_z locally to generate ${\beta_\mathsf{z}}^d = (\mathsf{z} + \mathsf{r})^d$ and generate $[\![(\mathsf{z} + \mathsf{r})^d]\!]$ non-interactively (see §5.4), followed by computing $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z} + \mathsf{r})^d]\!] - [\![\mathsf{r}^d]\!]$. To generate $[\![\cdot]\!]$-shares of the truncation pair $(\mathsf{r}, \mathsf{r}^d)$, we extend ideas in [173] to our setting, and the resultant protocol is called Π_{TrPair}. For this, parties non-interactively generate $[\![\mathsf{r}]\!]^{\mathbf{B}}$ using their shared-key setup, and truncate the last d bits of each of its share to generate $[\![\mathsf{r}^d]\!]^{\mathbf{B}}$. To obtain $[\![\mathsf{r}]\!]$ from $[\![\mathsf{r}]\!]^{\mathbf{B}}$, parties proceed as follows. Analogous steps enable generation of $[\![\mathsf{r}^d]\!]$ from $[\![\mathsf{r}^d]\!]^{\mathbf{B}}$. Set $\beta_\mathsf{r} = 0$ in $[\![\mathsf{r}]\!]$. Let the other shares of $[\![\mathsf{r}]\!]$ be denoted as r_{ij} for $1 \leq i < j \leq 5$. Without loss of generality, parties non-interactively sample all r_{ij} but r_{12}, as per the $[\![\cdot]\!]$-sharing. Enabling P_3, P_4, P_5 obtain $\mathsf{r}_{12} = \mathsf{r} - \sum_{ij \neq 12} \mathsf{r}_{ij}$ will complete generation of $[\![\mathsf{r}]\!]$. For this, observe that we can write $\mathsf{r} = \mathsf{v}_1 + \mathsf{v}_2 + \mathsf{v}_3 + \mathsf{v}_4$ where $\mathsf{v}_1 = \mathsf{r}_{34} + \mathsf{r}_{35} + \mathsf{r}_{45}$ is held by P_1, P_2, $\mathsf{v}_2 = \mathsf{r}_{45} + \mathsf{r}_{25}$ is held by P_1, P_3, $\mathsf{v}_3 = \mathsf{r}_{14} + \mathsf{r}_{15}$ is held by P_2, P_3 and $\mathsf{v}_4 = \mathsf{r}_{12} + \mathsf{r}_{13} + \mathsf{r}_{23}$ is held by P_4, P_5. Thus, revealing $\mathsf{v}_4 = \mathsf{r} - \mathsf{v}_1 - \mathsf{v}_2 - \mathsf{v}_3$ to P_4, P_5 enables them to compute $\mathsf{r}_{12} = \mathsf{v}_4 - \mathsf{r}_{13} - \mathsf{r}_{23}$ which they can send to P_3 by invoking Π_{Jmp}, thereby generating $[\![\mathsf{r}]\!]$. For this, given $[\![\mathsf{r}]\!]^{\mathbf{B}}$, parties compute $[\![\mathsf{v}_4]\!]^{\mathbf{B}} = [\![\mathsf{r}]\!]^{\mathbf{B}} + \sum_{i=1}^{3} [\![-\mathsf{v}_i]\!]^{\mathbf{B}}$ by evaluating Boolean addition circuit. Elaborately, P_1, P_2 generate $[\![-\mathsf{v}_1]\!]^{\mathbf{B}}$, P_1, P_3 generate $[\![-\mathsf{v}_2]\!]^{\mathbf{B}}$, and P_2, P_3 generate $[\![-\mathsf{v}_3]\!]^{\mathbf{B}}$ by invoking the joint sharing protocol, Π_{JSh2} (§5.4). Note that this joint sharing generates $[\![-\mathsf{v}_i[k]]\!]^{\mathbf{B}}$ for each bit $-\mathsf{v}_i[k]$ of $-\mathsf{v}_i$ for $i \in \{1, 2, 3\}, k \in \{0, \ldots, \ell - 1\}$. Parties proceed to compute the sum $[\![\mathsf{v}_4[k]]\!]^{\mathbf{B}} = [\![\mathsf{r}[k]]\!]^{\mathbf{B}} + \sum_{i=1}^{3} [\![-\mathsf{v}_i[k]]\!]^{\mathbf{B}}$ for each bit using a full adder (FA) circuit, as described in [173]. It follows from [173] that $x = x_1 + x_2 + x_3$ can be expressed as $x = 2c + s$ where $\mathrm{FA}(x_1[k], x_2[k], x_3[k]) \to (c[k], s[k])$ for $k \in \{0, \ldots, \ell - 1\}$. Here, s and c denote the sum and carry bits, respectively. Thus, parties compute $[\![\mathsf{v}_4[k]]\!]^{\mathbf{B}}$ for $k \in \{0, \ldots, \ell - 1\}$, simultaneously, by executing the FA's as given below.

- $\mathrm{FA}(\mathsf{r}[k], -\mathsf{v}_1[k], -\mathsf{v}_2[k]) \to (c_1[k], s_1[k])$
- $\mathrm{FA}(-\mathsf{v}_3[k], c_1[k-1], s_1[k]) \to (c_2[k], s_2[k])$
- $\mathrm{PPA}(2c_2, s_2) \to \mathsf{v}_4$

After the FA is executed, $[\![\mathsf{v}_4]\!]^{\mathbf{B}}$ is computed using the 2-input Parallel Prefix Adder (PPA) circuit [173] on inputs $2[\![c_2]\!]^{\mathbf{B}}$, $[\![s_2]\!]^{\mathbf{B}}$. The computations above are carried out on the $[\![\cdot]\!]^{\mathbf{B}}$-

shares, and $2c_1[k] = c_1[k-1]$ and $c[-1] = 0$. Having obtained $[\![\mathsf{v}_4]\!]^{\mathbf{B}}$, parties reconstruct v_4 towards P_4, P_5, who compute $\mathsf{r}_{12} = \mathsf{v}_4 - \mathsf{r}_{13} - \mathsf{r}_{23}$, and invoke Π_{Jmp} to send it to P_3. This completes the generation of $[\![\mathsf{r}]\!]$.

Bit to arithmetic Protocol Π_{Bit2A} allows computation of arithmetic shares, $[\![\mathsf{b}^{\mathsf{R}}]\!]$ of a bit $\mathsf{b} \in \mathbb{Z}_2$ from its Boolean shares, $[\![\mathsf{b}]\!]^{\mathbf{B}}$, where b^{R} denotes arithmetic equivalent of b over $\mathbb{Z}_{2^\ell}$. Observe that, following [138], $\mathsf{b}^{\mathsf{R}} = (\beta_{\mathsf{b}} \oplus \alpha_{\mathsf{b}})^{\mathsf{R}} = {\beta_{\mathsf{b}}}^{\mathsf{R}} + {\alpha_{\mathsf{b}}}^{\mathsf{R}} - 2{\beta_{\mathsf{b}}}^{\mathsf{R}}{\alpha_{\mathsf{b}}}^{\mathsf{R}}$. Given $\left[{\alpha_{\mathsf{b}}}^{\mathsf{R}}\right]$ and $[\mathsf{r}]$ for $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ can be generated in the preprocessing phase, parties can compute $\left[\mathsf{b}^{\mathsf{R}} + \mathsf{r}\right]$ in the online phase and reconstruct it towards all. Possession of $\mathsf{b}^{\mathsf{R}} + \mathsf{r}$ by all enables non-interactive generation of its $[\![\cdot]\!]$-shares (§5.4), from which $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\mathsf{b}^{\mathsf{R}} + \mathsf{r}]\!] - [\![\mathsf{r}]\!]$ can be computed. To generate $\left[{\alpha_{\mathsf{b}}}^{\mathsf{R}}\right]$, parties first generate $[\![{\alpha_{\mathsf{b}}}^{\mathsf{R}}]\!]$, and convert it to $[\cdot]$-shares via $\Pi_{[\![\cdot]\!]\to[\cdot]}$ (§5.3.3). To generate $[\![{\alpha_{\mathsf{b}}}^{\mathsf{R}}]\!]$, observe that the $[\cdot]^{\mathbf{B}}$-shares of α_{b} can be written as $\alpha_{\mathsf{b}} = \nu_1 \oplus \nu_2 \oplus \nu_3 \oplus \nu_4$ where $\nu_1 = \alpha_{\mathsf{b}_{34}} \oplus \alpha_{\mathsf{b}_{35}} \oplus \alpha_{\mathsf{b}_{45}}$, $\nu_2 = \alpha_{\mathsf{b}_{24}} \oplus \alpha_{\mathsf{b}_{25}}$, $\nu_3 = \alpha_{\mathsf{b}_{14}} \oplus \alpha_{\mathsf{b}_{15}}$ and $\nu_4 = \alpha_{\mathsf{b}_{12}} \oplus \alpha_{\mathsf{b}_{13}} \oplus \alpha_{\mathsf{b}_{23}}$. As seen in truncation pair generation, P_1, P_2 hold ν_1, P_1, P_3 hold ν_2, P_2, P_3 hold ν_3 and P_4, P_5 hold ν_4. Given $[\![\cdot]\!]^{\mathbf{B}}$-shares of each of $\nu_1, \nu_2, \nu_3, \nu_4$ can be generated via Π_{JSh2}, parties generate $[\![\cdot]\!]$-shares of $\mathsf{p} = \nu_1 \oplus \nu_2$ and $\mathsf{q} = \nu_3 \oplus \nu_4$ using the arithmetic equivalent of XOR and use these values to generate $[\![\alpha_{\mathsf{b}}{}^{\mathsf{R}}]\!] = [\![(\mathsf{p} \oplus \mathsf{q})^{\mathsf{R}}]\!]$. The protocol appears in Fig. 5.18.

Protocol $\Pi_{\mathsf{Bit2A}}(\mathcal{P}, [\![\mathsf{b}]\!]^{\mathbf{B}})$

Preprocessing:

– P_1, P_2 jointly share $\nu_1 = (\alpha_{\mathsf{b}_{34}} \oplus \alpha_{\mathsf{b}_{35}} \oplus \alpha_{\mathsf{b}_{45}})$, P_1, P_3 jointly share $\nu_2 = (\alpha_{\mathsf{b}_{24}} \oplus \alpha_{\mathsf{b}_{25}})$, P_2, P_3 jointly share $\nu_3 = (\alpha_{\mathsf{b}_{14}} \oplus \alpha_{\mathsf{b}_{15}})$ and P_4, P_5 jointly share $\nu_4 = (\alpha_{\mathsf{b}_{12}} \oplus \alpha_{\mathsf{b}_{13}} \oplus \alpha_{\mathsf{b}_{23}})$ to generate $[\![\nu_1^{\mathsf{R}}]\!], [\![\nu_2^{\mathsf{R}}]\!], [\![\nu_3^{\mathsf{R}}]\!], [\![\nu_4^{\mathsf{R}}]\!]$, respectively.

– Parties execute Π_{Mul} on $([\![\nu_1^{\mathsf{R}}]\!], [\![\nu_2^{\mathsf{R}}]\!])$, $([\![\nu_3^{\mathsf{R}}]\!], [\![\nu_4^{\mathsf{R}}]\!])$ to generate $[\![\nu_1^{\mathsf{R}} \cdot \nu_2^{\mathsf{R}}]\!]$ and $[\![\nu_3^{\mathsf{R}} \cdot \nu_4^{\mathsf{R}}]\!]$, respectively.

– Parties non-interactively compute $[\![\mathsf{p}^{\mathsf{R}}]\!] = [\![(\nu_1 \oplus \nu_2)^{\mathsf{R}}]\!] = [\![\nu_1^{\mathsf{R}}]\!] + [\![\nu_2^{\mathsf{R}}]\!] - 2 \cdot [\![\nu_1^{\mathsf{R}} \cdot \nu_2^{\mathsf{R}}]\!]$ and $[\![\mathsf{q}^{\mathsf{R}}]\!] = [\![(\nu_3 \oplus \nu_4)^{\mathsf{R}}]\!] = [\![\nu_3^{\mathsf{R}}]\!] + [\![\nu_4^{\mathsf{R}}]\!] - 2 \cdot [\![\nu_3^{\mathsf{R}} \cdot \nu_4^{\mathsf{R}}]\!]$.

– Parties execute Π_{Mul} on $[\![\mathsf{p}^{\mathsf{R}}]\!], [\![\mathsf{q}^{\mathsf{R}}]\!]$ to generate $[\![\mathsf{p}^{\mathsf{R}} \cdot \mathsf{q}^{\mathsf{R}}]\!]$, and compute $[\![{\alpha_{\mathsf{b}}}^{\mathsf{R}}]\!] = [\![(\mathsf{p} \oplus \mathsf{q})^{\mathsf{R}}]\!] = [\![\mathsf{p}^{\mathsf{R}}]\!] + [\![\mathsf{q}^{\mathsf{R}}]\!] - 2 \cdot [\![\mathsf{p}^{\mathsf{R}} \cdot \mathsf{q}^{\mathsf{R}}]\!]$.

– Parties non-interactively generate $[\![\mathsf{r}]\!]$ for $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, and invoke $\Pi_{[\![\cdot]\!]\to[\cdot]}$ to generate $\left[{\alpha_{\mathsf{b}}}^{\mathsf{R}}\right]$, $[\mathsf{r}]$.

Online:

– Compute $\left[\mathsf{b}^{\mathsf{R}} + \mathsf{r}\right] = {\beta_{\mathsf{b}}}^{\mathsf{R}} + \left[{\alpha_{\mathsf{b}}}^{\mathsf{R}}\right] - 2{\beta_{\mathsf{b}}}^{\mathsf{R}}\left[{\alpha_{\mathsf{b}}}^{\mathsf{R}}\right] + [\mathsf{r}]$, and reconstruct $\mathsf{b}^{\mathsf{R}} + \mathsf{r}$ towards all, similar to multiplication.

– Non-interactively generate $[\![\mathsf{b}^{\mathsf{R}} + \mathsf{r}]\!]$ (§5.4), followed by $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\mathsf{b}^{\mathsf{R}} + \mathsf{r}]\!] - [\![\mathsf{r}]\!]$.

Figure 5.18: Bit to arithmetic conversion

Note that the preprocessing phase can be optimized further. Instead of invoking the entire Π_{Mul} in the preprocessing phase which requires communicating 14ℓ elements, we can generate the required multiplicative terms by invoking $\mathcal{F}_{\mathsf{MulPre}}$ (whose current realization via the modified variant of [31] as described in §5.6.3, requires 6ℓ elements). For this, $[\![\cdot]\!]$-shares of $\nu_1, \nu_2, \nu_3, \nu_4$ are converted to $[\cdot]$-shares by invoking $\Pi_{[\![\cdot]\!]\to[\cdot]}$, followed by invoking $\mathcal{F}_{\mathsf{MulPre}}$ on the respective terms. The result of multiplication, generated as $[\cdot]$-shares, can be converted to $[\![\cdot]\!]$-shares by invoking $\Pi_{[\cdot]\to[\![\cdot]\!]}$.

Bit extraction Bit extraction (Π_{Bitext}) enables generation of $[\![\cdot]\!]^{\mathbf{B}}$-shares of the most significant bit (msb) of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ given $[\![\mathsf{v}]\!]$. Support for multi-input multiplication enables usage of the optimized bit extraction circuit proposed in [194], which takes two values as inputs and outputs the msb of the sum of these values. Given $[\![\mathsf{v}]\!]$, we generate the Boolean shares of the two inputs to the bit extraction circuit as follows. Observe that v can be written $\mathsf{v} = \beta_{\mathsf{v}} + (-\alpha_{\mathsf{v}})$. Thus, β_{v} and $-\alpha_{\mathsf{v}}$ serve as the two inputs. $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$ can be generated non-interactively in the online phase since all parties hold β_{v} (see §5.4). To generate $[\![-\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$ from $[\alpha_{\mathsf{v}}]$, parties proceed as follows in the preprocessing phase. Parties first generate $-\alpha_{\mathsf{v}}$ by locally negating all their shares of α_{v}. For ease of presentation, let $\alpha = -\alpha_{\mathsf{v}}$ and $[\alpha] = [-\alpha_{\mathsf{v}}] = (\alpha_{ij})_{1\le i<j\le 5}$. Recall that $\alpha = \nu_1 + \nu_2 + \nu_3 + \nu_4$ where $\nu_1 = \alpha_{34} + \alpha_{35} + \alpha_{45}$, $\nu_2 = \alpha_{24} + \alpha_{25}$, $\nu_3 = \alpha_{14} + \alpha_{15}$ and $\nu_4 = \alpha_{12} + \alpha_{13} + \alpha_{23}$, and each term is held by a pair of parties. Similar to Π_{TrPair}, after the different pairs of parties generate $[\![\nu_1]\!]^{\mathbf{B}}, [\![\nu_2]\!]^{\mathbf{B}}, [\![\nu_3]\!]^{\mathbf{B}}, [\![\nu_4]\!]^{\mathbf{B}}$, evaluating two sequential full adders followed by a PPA circuit generates $[\![\alpha]\!]^{\mathbf{B}}$. Having obtained $[\![\alpha]\!]^{\mathbf{B}}$ and $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$, parties execute the optimized bit extraction circuit to extract the $\mathsf{msb}(\mathsf{v})$.

Arithmetic to Boolean Protocol Π_{A2B} generates $[\![\cdot]\!]^{\mathbf{B}}$-shares for each bit of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, denoted as $[\![\mathsf{v}]\!]^{\mathbf{B}}$, from $[\![\mathsf{v}]\!]$. For this, observe that $\mathsf{v} = \beta_{\mathsf{v}} + (-\alpha_{\mathsf{v}})$. Thus, evaluating the optimized PPA circuit [194] on $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}, [\![-\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$ generates $[\![\mathsf{v}]\!]^{\mathbf{B}}$. For this, $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$ can be generated non-interactively since all parties hold β_{v} (see §5.4). To generate $[\![-\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$ from $[\alpha_{\mathsf{v}}]$, parties follow the steps as described in Π_{Bitext}.

Bit injection Given $[\![\mathsf{b}]\!]^{\mathbf{B}}$, $[\![\mathsf{v}]\!]$ where $\mathsf{b} \in \mathbb{Z}_2$, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, bit injection ($\Pi_{\mathsf{BitInj}}$) generates $[\![\mathsf{b}^{\mathsf{R}} \cdot \mathsf{v}]\!]$. For this, parties run Π_{Bit2A} to generate $[\![\mathsf{b}^{\mathsf{R}}]\!]$, followed by Π_{Mul} to generate $[\![\mathsf{b}^{\mathsf{R}} \cdot \mathsf{v}]\!]$.

Oblivious select Protocol Π_{Sel} takes as input $[\![\mathsf{x}_1]\!]$, $[\![\mathsf{x}_2]\!]$, $[\![\mathsf{b}]\!]^{\mathbf{B}}$, where $\mathsf{x}_1, \mathsf{x}_2 \in \mathbb{Z}_{2^\ell}$ and $\mathsf{b} \in \mathbb{Z}_2$, and outputs re-randomized $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{x}_{\mathsf{b}}$. Since $\mathsf{z} = \mathsf{x}_{\mathsf{b}} = \mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0) + \mathsf{x}_0$, computing $[\![\mathsf{z}]\!]$ requires one invocation of Π_{BitInj} and addition operations.

Equality check On input $[\![\mathsf{x}]\!], [\![\mathsf{y}]\!]$, equality check protocol (Π_{Eq}) outputs $[\![\mathsf{b}]\!]^{\mathbf{B}}$ where $\mathsf{b} = 1$, if $\mathsf{x} = \mathsf{y}$, and $\mathsf{b} = 0$, otherwise. Similar to [194], the approach is to compute $\mathsf{v} = \mathsf{x} - \mathsf{y}$ and check if all bits of v are 0. Concretely, parties first obtain $[\![\mathsf{v}]\!]^{\mathbf{B}}$ by invoking Π_{A2B} on $[\![\mathsf{v}]\!]$, compute $[\![\bar{\mathsf{v}}]\!]^{\mathbf{B}}$ ($\bar{\mathsf{v}}$ denotes bit complement of v) non-interactively, followed by invoking the 4-input (Boolean) multiplication, recursively, to generate $[\![\mathsf{b}]\!]^{\mathbf{B}}$.

Comparison On input $[\![\mathsf{x}]\!], [\![\mathsf{y}]\!]$, Π_{Comp} outputs $[\![\mathsf{b}]\!]^{\mathbf{B}}$ where $\mathsf{b} = 1$, if $\mathsf{x} < \mathsf{y}$, and $\mathsf{b} = 0$, otherwise. This reduces to checking msb of $\mathsf{v} = \mathsf{x} - \mathsf{y}$, and hence, Π_{Bitext} can be used.

Maxpool/minpool Maxpool allows computing the maximum element from a set of m elements. We follow a similar approach as in [138], where the elements are recursively compared in a pair-wise manner to obtain the maximum element. Minpool can also be computed analogously.

ReLU The ReLU function computes the maximum between 0 and a value v, and can be computed as $\mathsf{ReLU}(\mathsf{v}) = \bar{\mathsf{b}} \cdot \mathsf{v}$, where $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and $\mathsf{b} = 0$, otherwise. Here, $\bar{\mathsf{b}}$ denotes the complement of bit b. Given $[\![\mathsf{v}]\!]$, b can be computed via Π_{Bitext}, followed by non-interactively computing $\bar{\mathsf{b}}$, followed by Π_{BitInj} to compute $[\![\bar{\mathsf{b}}^{\mathsf{R}} \cdot \mathsf{v}]\!]$.

Complexity of building blocks Table 5.3 lists the complexities of the designed building blocks.

5.9 Benefit of having fewer parties online

As described in §5.6, rather than having a multiplication protocol with all parties online, considerable effort was spent in reducing the number of online parties to only 3. We now showcase the concrete improvements brought in by this approach. The results corroborate that the reduction in online parties is indeed beneficial.

Benchmark environment and parameters We report results in LAN (1 Gbps bandwidth) with 2.3 GHz Quad-Core Intel Core i7 machines having 16GB RAM. The average round trip time (rtt) for communicating 1KB of data between a pair of machines is 0.29 milliseconds (ms). The protocols build on the ENCRYPTO library [59] in C++17 over a 64-bit ring. We use multi-threading, wherever possible, to facilitate efficient computation and communication among the parties. Since there is no defined way to capture an adversary's misbehaviour,

Building block	Online		Preprocessing
	Rounds	Comm. (in bits)	Comm. (in bits)
Multiplication	1	8ℓ	6ℓ
3-input Multiplication	1	8ℓ	24ℓ
4-input Multiplication	1	8ℓ	66ℓ
Dot product	1	8ℓ	6ℓ
Matrix Multiplication	1	$8\mathsf{pq}\ell$	$6\mathsf{pq}\ell$
Multiplication with Truncation	1	8ℓ	$27\ell + 6\ell\log_2\ell$
Bit to arithmetic	1	8ℓ	22ℓ
Bit extraction	$\log_4\ell$	u_2'	$16\ell + 6\ell\log_2\ell + \mathsf{u}_1'$
Arithmetic to Boolean	$\log_4\ell$	u_2	$16\ell + 6\ell\log_2\ell + \mathsf{u}_1$
Bit Injection	2	16ℓ	28ℓ
Oblivious Select	2	16ℓ	28ℓ
Equality	$\log_4\ell$	$\mathsf{u}_2 + 168$	$16\ell + 6\ell\log_2\ell + \mathsf{u}_1 + 1386$
Comparison	$\log_4\ell$	u_2'	$16\ell + 6\ell\log_2\ell + \mathsf{u}_1'$
Maxpool/minpool	$\log_2 m(\log_4\ell + 2)$	$(m-1)(\mathsf{u}_2' + 16\ell)$	$(m-1)(44\ell + 6\ell\log_2\ell + \mathsf{u}_1')$
ReLU	$\log_4\ell + 2$	$\mathsf{u}_2' + 16\ell$	$44\ell + 6\ell\log_2\ell + \mathsf{u}_1'$

ℓ: size of ring in bits, instantiated with $\ell = 64$; $\mathsf{p} \times \mathsf{q}$ denotes the dimension of the resultant matrix after matrix multiplication; $\mathsf{u}_1' = 6\mathsf{n}_2 + 24\mathsf{n}_3 + 66\mathsf{n}_4$, $\mathsf{u}_2' = 8\,(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 41, \mathsf{n}_3 = 27, \mathsf{n}_4 = 47$ denote the number of AND gates in the optimized bit extraction circuit of [194] with $2, 3, 4$ inputs, respectively; $\mathsf{u}_1 = 6\mathsf{n}_2 + 24\mathsf{n}_3 + 66\mathsf{n}_4$, $\mathsf{u}_2 = 8\,(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [194] with $2, 3, 4$ inputs, respectively; m denotes number of elements to be compared via maxpool.

Table 5.3: Building blocks with their complexity.

following standard practice [173, 193, 38, 136], we benchmark honest executions of the protocols, including the verification required to attain GOD. Hence the reported run time does not account for the 3PC execution. Note, however, that 5PC execution itself accounts for the worst-case computation because it has a higher number of parties, including one malicious corruption, as opposed to 3PC. We use the time taken for the protocol to complete and communication between parties as the two parameters for benchmarks. We report these values separately for the online and preprocessing phases. Further, we also report online throughput (TP), which is the number of circuit executions that can be processed in a second.

Comparison We compare our optimized multiplication protocol, which requires only 3 out of the 5 parties for most of the online phase, with the non-optimized variant, which requires all parties to remain online. We also compare our protocol with the traditional $(5, 2)$ protocol obtained from [31], which also requires all parties to be online. To showcase the improvement

achieved in the optimized variant, we benchmark synthetic circuits of varying depths (10, 100, 1000, 10000) with 100 multiplication gates at each layer. For all the variants, Table 5.4 reports the time, throughput and monetary cost of the system. While throughput simultaneously captures the improvements in communication and round complexity, we additionally report monetary costs to showcase the effect of the number of parties on the operational cost of the system. We report these values only for the online phase. We estimate the monetary cost following standard Google Cloud pricing [205].

Circuit depth	Protocol type	Online time (s)	TP ($\times 10^2$)	Monetary cost ($\times 10^{-3}$ USD)
10	Optimized	0.005	121.189	0.002
	Non-optimized	0.011	59.435	0.006
	[31]	0.008	78.259	0.002
100	Optimized	0.034	19.037	0.018
	Non-optimized	0.107	6.006	0.057
	[31]	0.0543	11.783	0.031
1000	Optimized	0.329	1.948	0.178
	Non-optimized	1.059	0.604	0.571
	[31]	0.530	1.206	0.253
10000	Optimized	3.152	0.203	1.758
	Non-optimized	10.638	0.060	5.711
	[31]	5.154	0.124	2.452

Table 5.4: Comparison for synthetic circuits.

The round complexity of the non-optimized variant is roughly 3× that of the optimized variant, assuming that the time taken for Jmp-vrfy gets amortized. This is evident from the reported online time, for circuit depth 100 and beyond. This is, however, not the case for the circuit of depth 10. This is because the time taken for the Jmp-vrfy in the optimized variant (2 rounds) is comparable to that of circuit evaluation (10 rounds for Jmp-send) and hence does not get amortized. The improved online time is reflected as improvements in throughput as well, where the gain is up to 3×. Finally, the reduction in the number of online parties is clearly evident in monetary cost, since it captures the price paid to host the required number of parties (inclusive of its computation and communication). The optimized variant witnesses up to 69% savings in monetary cost compared to the non-optimized variant.

With respect to [31], our optimized variant has an improvement of up to 1.6× in run time and throughput. While the monetary cost reported in Table 5.4 is for the online phase, to draw a fair comparison between our (optimized) protocol and [31], we also account for the monetary cost of preprocessing phase. In doing so, we observe that even for a circuit of depth 10000, the

overall monetary cost of our protocol is 2.57×10^{-3} USD, which is only slightly higher than that of [31].

5.10 Dark pools algorithms

We consider two popular matching algorithms used in dark pools—continuous double auction (CDA) algorithm and volume-based matching algorithm. While the former processes orders in a continuous manner, the latter does so in scheduled intervals, and both algorithms rely on different parameters for matching orders. Both these matching algorithms have been considered in prior works, albeit in the traditional MPC setting [40, 60]. Although the functionality of these algorithms remains the same as described in [40], we take advantage of possible parallelization and tweak the algorithms to improve their round complexity. This, in turn, improves the run time of the protocols and the number of orders that can be processed in unit time (throughput). We next detail each of these algorithms and their overall performance.

5.10.1 Continuous double auction

The CDA algorithm maintains a sorted list of buy orders ($\mathcal{B}$) and sell orders ($\mathcal{S}$) that are yet to be matched. A buy order comprises the client's identity, $name^b$, the units to be bought, b, and the buying price, also known as *bid*, q. Analogously, a sell order comprises the client's identity $name^s$, the units to be sold s, and the selling price, also known as *offer* p. All the unmatched buy orders in the list $\mathcal{B}$ (where $|\mathcal{B}| = M$) are sorted in descending order of their bid. Similarly, sell orders in list $\mathcal{S}$ (where $|\mathcal{S}| = N$) are sorted in ascending order of offer. The CDA algorithm maintains this as an invariant.

The CDA algorithm for processing a new order has two phases–(i) matching, and (ii) insertion. In the matching phase, the incoming order is matched with orders of the opposite type. Elaborately, a buy order is said to match a sell order if the following criteria are met–(i) *Price criteria*: the bid of the buy order must be greater than or equal to the offer of the sell order and, (ii) *Volume criteria*: the units of one order must be able to satisfy the units of the other. Thus, when a new buy order arrives, it is matched with the first order in $\mathcal{S}$ based on the matching criteria. The buy order may continue to be matched with other sell orders in $\mathcal{S}$, until either of the criteria for matching fails. Hence matches need not be one-to-one. An incoming sell order can also be processed analogously. The matching phase concludes with the incoming order being in one of the following two states. The order may be *satisfied* if all of its units are exhausted by getting matched to opposite orders, or, it may be *partially satisfied* if

some of its units are still unmatched. If the incoming order is partially satisfied, the algorithm enters the insertion phase that involves inserting this order into the corresponding list $\mathcal{B}$ or $\mathcal{S}$ while respecting the sorted order maintained within it. We refer to the algorithm in [40] for further details.

A secure variant of the CDA algorithm was given in [40], where all orders remain hidden until they are satisfied. However, the order type (buy or sell) and hence the size of $\mathcal{S}$ and $\mathcal{B}$ are not regarded as sensitive information. We describe an improved secure protocol for CDA algorithm to process an incoming buy order. An incoming sell order can be processed analogously.

In [40], the protocol identifies matching sell orders in $\mathcal{S}$ sequentially and terminates when the incoming order can no longer be matched. Instead, we perform additional bookkeeping to identify all the matching sell orders in a single shot. This was not possible in [40] because the number of unmatched units remaining was tracked sequentially. However, we compute the cumulative sum w_i of the units of the first i sell orders in $\mathcal{S}$, which facilitates single-shot identification of matching sell orders. While the satisfaction of the price criteria for all sell orders in $\mathcal{S}$ can be determined in parallel, w allows determining satisfaction of the volume criteria also, for all sell orders in parallel. Thus, one does not require to wait for the i^{th} order to be matched before processing the $i+1^{\text{th}}$ order. Hence, all those sell orders where both conditions are met can be executed and revealed in public. Note that the last sell order to be matched could either be fully satisfied or partially satisfied and hence needs extra care. The protocol for the above matching phase is given in Fig. 5.19, where the changes made over the existing protocol are highlighted.

Protocol $\Pi_{\mathsf{PSL}}\left(\mathcal{P}, ([\![name_0^b]\!], [\![b_0]\!], [\![q_0]\!]), \mathcal{S}\right)$

- Set $[\![w_0]\!] = [\![0]\!]$
- For each $i = 1$ to N do in parallel: $[\![w_i]\!] = \sum_{j=1}^{i} [\![s_j]\!]$
- For each $i = 1$ to N do in parallel:
  - $[\![z_i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, [\![w_{i-1}]\!], [\![b_0]\!])$, $[\![z_i']\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, [\![p_i]\!], [\![q_0]\!] + 1)$
- For $i = 1$ to N do in parallel: $[\![f_i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Mul}}(\mathcal{P}, [\![z_i]\!]^{\mathbf{B}}, [\![z_i']\!]^{\mathbf{B}})$
- Reconstruct f_i's and set $k = i$ such that $f_i = 1$ and $f_{i+1} = 0$ for $i \in \{1, \ldots, N\}$. Else set $k = 0$.
- For each $i = 1$ to $k-1$ do in parallel: Reconstruct $([\![name_i^s]\!], [\![s_i]\!], [\![p_i]\!])$
- $[\![s_k']\!] = \Pi_{\mathsf{Sel}}([\![b_0]\!] - [\![w_{k-1}]\!], [\![s_k]\!], [\![z_{k+1}]\!]^{\mathbf{B}})$
- Reconstruct $([\![name_k^s]\!], [\![s_k']\!], [\![p_k]\!])$, set $[\![s_k]\!] = [\![s_k]\!] - [\![s_k']\!]$
- Delete first $k-1$ elements from $\mathcal{S}$.

Figure 5.19: CDA matching phase: processing sell list.

The insertion phase follows the matching phase, where the incoming buy order is *obliviously* inserted into $\mathcal{B}$ in the correct slot that respects the ordering maintained as an invariant. Since the steps of the protocol for the insertion phase as well as the overall CDA algorithm, remain the same as in [40], we do not elaborate on them. However, we continue to execute independent instructions in parallel within these protocols, too, and render the overall execution as efficient as possible. The protocols for the insertion phase and overall CDA are given in Fig. 5.20 and Fig. 5.21. In protocol, Π_{Insert} (Fig. 5.20), for the insertion phase of CDA, we note that each of the f_i's for $i \in \{1, 2, \dots\}$ can be computed in parallel. Subsequently, so can f'_i's followed by f''_i's. Note that the instructions in Π_{CDA} are all sequential.

Protocol $\Pi_{\mathsf{Insert}}\left(\mathcal{P}, (\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket), \mathcal{B}\right)$

- Insert $(\llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket)$ to the end of $\mathcal{B}$
- Compute $\llbracket f_0 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, \llbracket q_0 \rrbracket, \llbracket q_1 \rrbracket))$, followed by $\llbracket f_0 \rrbracket = \Pi_{\mathsf{Bit2A}}(\llbracket f_0 \rrbracket^{\mathbf{B}})$
- For $i = 1$ to $M + 1$ do
  - $\llbracket f_i \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, \llbracket q_0 \rrbracket, \llbracket q_i \rrbracket + 1))$
  - $\llbracket f_i \rrbracket = \Pi_{\mathsf{Bit2A}}(\llbracket f_i \rrbracket^{\mathbf{B}})$, $\llbracket f'_i \rrbracket = (1 - \llbracket f_i \rrbracket) \cdot \llbracket f_{i-1} \rrbracket)$, $\llbracket f''_i \rrbracket = (1 - \llbracket f_i \rrbracket) \cdot (1 - \llbracket f'_i \rrbracket)$
- For $i = 1$ to $M + 1$ do in parallel
  - $\llbracket name_i'^b \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket name_i^b \rrbracket + \llbracket f'_i \rrbracket \cdot \llbracket name_0^b \rrbracket + \llbracket f''_i \rrbracket \cdot \llbracket name_{i-1}^b \rrbracket$
  - $\llbracket b'_i \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket b_i \rrbracket + \llbracket f'_i \rrbracket \cdot \llbracket b_0 \rrbracket + \llbracket f''_i \rrbracket \cdot \llbracket b_{i-1} \rrbracket$, $\llbracket q'_i \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket q_i \rrbracket + \llbracket f'_i \rrbracket \cdot \llbracket q_0 \rrbracket + \llbracket f''_i \rrbracket \cdot \llbracket q_{i-1} \rrbracket$

Figure 5.20: Obliviously inserting into buy list.

Protocol $\Pi_{\mathsf{CDA}}\left(\mathcal{P}, (\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket), \mathcal{B}, \mathcal{S}\right)$

- Invoke Π_{PSL} on $(\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket)$
- Compute $\llbracket e \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{Eq}}(\mathcal{P}, \llbracket b_0 \rrbracket, 0)$ and $\llbracket q_0 \rrbracket = \Pi_{\mathsf{Sel}}(\llbracket q_0 \rrbracket, \llbracket 0 \rrbracket, \llbracket e \rrbracket^{\mathbf{B}})$
- Invoke Π_{Insert} to insert $(\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket)$ into buy list

Figure 5.21: Overall CDA.

5.10.2 Volume matching

Unlike the CDA algorithm, where orders are processed in a continuous manner, volume-based matching processes all the requests at fixed intervals. The algorithm matches orders based only on the volume. Hence, the i^{th} client only submits the number of units it wishes to buy b_i or sell s_i, and the matching is done on a first-come-first-serve basis. Similar to CDA, the buy orders and sell orders are maintained in a separate list (queue), ordered by their arrival. Since the

algorithm only accounts for volume, one is guaranteed that either all the sell orders or all buy orders are satisfied. That is, the type of orders whose total volume is lesser will be satisfied completely. After processing the orders, the algorithm outputs the sequence of updated buy/sell orders such that the value now at b'_i or s'_i denotes the number of units traded out of the original b_i or s_i request. Although the algorithm is the same as in [40], we provide a parallel variant of the same in Fig. 5.22 and highlight the changes made over the existing protocol. Unlike in [40], the algorithm can be improved to process each sell/buy order in parallel by some additional bookkeeping, as done in §5.10.1.

Protocol $\Pi_{\mathsf{VM}}\left(\mathcal{P}, \{[\![s_i]\!]\}_{i=1}^{N}, \{[\![b_j]\!]\}_{j=1}^{M}\right)$

1. Compute $[\![S]\!] = \sum_{i=1}^{N} [\![s_i]\!]$ and $[\![B]\!] = \sum_{j=1}^{M} [\![b_j]\!]$
2. Compute $[\![f]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, [\![B]\!], [\![S]\!])$
3. Set $[\![T]\!] = \Pi_{\mathsf{Sel}}([\![S]\!], [\![B]\!], [\![f]\!]^{\mathbf{B}})$, $[\![s_0]\!] = 0$ and $[\![b_0]\!] = 0$
4. For i from 1 to N do in parallel: $[\![L_i^s]\!] = [\![T]\!] - \sum_{j=0}^{i-1} [\![s_j]\!]$
5. For i from 1 to M do in parallel: $[\![L_i^b]\!] = [\![T]\!] - \sum_{j=0}^{i-1} [\![b_j]\!]$
6. For i from 1 to N do in parallel:
   - $[\![z_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, [\![L_i^s]\!], [\![1]\!]))$ and $[\![z_2]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, ([\![L_i^s]\!], [\![s_i]\!])$
   - $[\![z_1]\!] = \Pi_{\mathsf{Bit2A}}(\mathcal{P}, [\![z_1]\!]^{\mathbf{B}})$ and $[\![z_2]\!] = \Pi_{\mathsf{Bit2A}}(\mathcal{P}, [\![z_2]\!]^{\mathbf{B}})$
   - $[\![s_i]\!] = (([\![L_i^s]\!] - [\![s_i]\!]) \cdot [\![z_2]\!] + [\![s_i]\!]) \cdot (1 - [\![z_1]\!])$
7. For j from 1 to M do in parallel:
   - $[\![z_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, [\![L_j^b]\!], [\![1]\!]))$ and $[\![z_2]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}(\mathcal{P}, ([\![L_j^b]\!], [\![b_j]\!])$
   - $[\![z_1]\!] = \Pi_{\mathsf{Bit2A}}(\mathcal{P}, [\![z_1]\!]^{\mathbf{B}})$ and $[\![z_2]\!] = \Pi_{\mathsf{Bit2A}}(\mathcal{P}, [\![z_2]\!]^{\mathbf{B}})$
   - $[\![b_i]\!] = (([\![L_j^b]\!] - [\![b_i]\!]) \cdot [\![z_2]\!] + [\![b_j]\!]) \cdot (1 - [\![z_1]\!])$
8. Reconstruct $[\![s_i]\!]$ and $[\![b_j]\!]$ for all i and j

Figure 5.22: Volume matching.

5.10.3 Benchmarks

We benchmark the performance of the proposed protocols in the same environment as described in §5.9. Since the complexity of dark pool algorithms depends on the size of buy list (N) and sell list (M), following [40], we analyze these algorithms by varying N and M between 10 and 500. Moreover, since the complexity of the CDA algorithm additionally depends on the number of executed sell orders (s), we set this to be 10% of the maximum of N and M[4]. For CDA, these

[4]Dark pools are not obligated to report the detailed information regarding volumes and types of transactions. Hence, we can only speculate the parameters such as s, N, M. Further, accounting for the recent trend of smaller

results are reported in Table 5.5. As expected and evident from Fig. 5.23a, the run time of the algorithms increases with increasing N and M. However, this increase is more pronounced in the algorithm of [40] due to its sequential nature and heavy dependence on s. To capture this effect more clearly, we perform experiments with fixed $N = M = 100$ and vary s between 1 to 50 and report these results in Table 5.6.

N	M	Ref	Preprocessing		Online		
			Time (ms)	Com (KB)	Time (ms)	Com (KB)	TP (orders/s)
10	10	Ours	1.67	37.36	13.76	26.61	72.65
		[40]	1.62	24.15	15.70	15.41	63.71
20	10	Ours	1.73	52.28	14.41	36.52	69.38
		[40]	1.70	41.97	23.88	27.95	41.87
20	20	Ours	1.82	70.19	14.63	47.58	68.37
		[40]	1.70	41.97	22.69	27.95	44.06
40	20	Ours	1.94	100.02	14.60	67.39	68.52
		[40]	1.87	77.61	37.19	53.56	26.89
50	50	Ours	2.28	168.68	14.34	110.47	69.74
		[40]	1.98	95.44	42.64	66.62	23.45
100	50	Ours	2.73	243.27	15.10	159.99	66.23
		[40]	2.43	184.56	75.54	134.58	13.24
100	100	Ours	3.25	333.19	15.80	215.40	63.28
		[40]	2.66	184.57	75.61	134.58	13.23
200	100	Ours	4.09	482.37	16.73	314.45	59.78
		[40]	3.53	363.10	143.25	283.62	6.98
200	200	Ours	5.74	662.14	16.89	425.26	59.22
		[40]	4.26	363.14	141.81	283.62	7.05
400	200	Ours	7.73	960.83	17.58	623.36	56.90
		[40]	7.04	720.10	281.36	634.16	3.55
500	500	Ours	18.95	1648.69	17.67	1054.63	56.59
		[40]	10.87	898.78	354.43	835.64	2.82

Table 5.5: Comparison for CDA for varying N, M, and s = 1/10(max(N, M)).

As explained earlier and as is evident from Table 5.6, observe that the run time of CDA linearly depends on s for the algorithm of [40]. On the contrary, the parallelizations in our algorithm help in making the run time independent of s, and thereby bring up to 20× saving in run time. The poor run time of [40] in comparison to ours can also be attributed to the large number of reconstructions in the former's CDA algorithm that necessitate performing

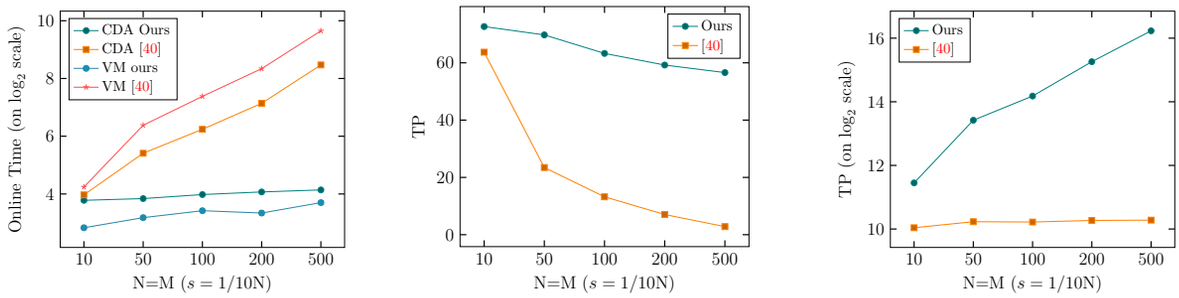
traders entering into dark pools, we consider the possibility of a large volume order matched against several small volume orders and set s to be 10%. This is in contrast to the unrealistic case of $s \in \{0, 1, 2, 3\}$ as in [40].

s	Ref	Preprocessing		Online		
		Time (ms)	Com (KB)	Time (ms)	Com (KB)	TP (orders/s)
1	Ours	3.41	333.19	17.13	190.09	58.37
	[40]	2.42	158.41	16.58	79.24	60.30
2	Ours	3.25	333.19	15.98	192.90	62.57
	[40]	2.56	161.32	24.50	84.68	40.81
4	Ours	3.28	333.19	15.38	198.53	65.00
	[40]	2.55	167.13	37.22	96.11	26.87
5	Ours	3.37	333.19	15.40	201.34	64.95
	[40]	2.48	170.04	42.73	102.08	23.40
10	Ours	3.26	333.19	15.46	215.40	64.67
	[40]	2.59	184.57	75.20	134.58	13.30
40	Ours	3.33	333.19	17.16	299.78	58.26
	[40]	3.06	271.77	281.21	421.39	3.56
50	Ours	3.18	333.19	15.77	327.90	63.40
	[40]	3.13	301.12	350.70	551.95	2.85

Table 5.6: Comparison for CDA for varying s and N=M=100.

verification each time a value is reconstructed (in our $(1, 1)$-FaF setting). The improvement of our algorithm is also reflected in throughput (TP), where our algorithm's TP remains almost constant, whereas the algorithm of [40] sees a steady fall. Here, TP is computed as $1/t_o$ where t_o is the online run time of the protocol.

The results for volume matching appear in Table 5.7. As expected, the throughput (TP) of volume matching is better than CDA. Further, due to the parallelizations introduced by our work, our algorithm's runtime increases very slowly compared to that of [40] with increasing N, M. This is visually represented in Fig. 5.23a, which compares the online runtime of volume matching and CDA algorithm. Since TP for volume matching is computed as $N + M/t_o$, where t_o denotes the online run time of the protocol, the slow increase in our run time helps in obtaining higher TP as N, M increase. This is not the case for [40], whose TP remains almost constant. The gain in TP for us thus turns out to be up to $62\times$ over the work of [40]. A visual comparison of TP for CDA and volume matching appears in Fig. 5.23b and Fig. 5.23c.



(a) CDA & Volume matching (b) CDA (c) Volume matching

Figure 5.23: Online time (a) and TP (orders/sec) comparison (b, c) of our algorithm with [40]

N	M	Ref	Preprocessing		Online		
			Time (ms)	Com (KB)	Time (ms)	Com (KB)	TP ($\times 10^3$ orders/s)
10	10	Ours	1.72	47.82	7.13	32.70	2.81
		[40]	1.70	45.94	18.93	9.31	1.06
20	10	Ours	1.81	71.06	7.83	48.45	3.83
		[40]	1.89	90.54	37.61	17.96	0.80
20	20	Ours	1.92	94.30	7.86	64.20	5.09
		[40]	1.89	90.54	34.83	17.96	1.15
40	20	Ours	2.11	140.78	7.79	95.69	7.70
		[40]	2.28	179.74	66.50	35.26	0.90
50	50	Ours	2.65	233.78	9.10	158.68	10.99
		[40]	2.56	224.37	83.28	43.91	1.20
100	50	Ours	3.11	350.27	9.29	237.42	16.14
		[40]	3.50	447.69	163.73	87.17	0.92
100	100	Ours	4.03	466.55	10.77	316.15	18.57
		[40]	3.74	447.73	167.17	87.17	1.20
200	100	Ours	5.04	699.44	10.02	473.62	29.95
		[40]	6.64	875.48	326.77	173.67	0.92
200	200	Ours	7.89	932.34	10.18	631.09	39.31
		[40]	7.19	894.37	323.33	173.67	1.24
400	200	Ours	10.73	1397.75	12.20	946.03	49.18
		[40]	12.30	1787.73	640.70	346.66	0.94
500	500	Ours	26.98	2329.07	12.99	1575.92	76.98
		[40]	23.34	2234.94	803.91	433.17	1.24

Table 5.7: Comparison for volume matching for varying N, M.

5.11 Privacy-preserving machine learning

To showcase that our FaF-secure protocols have wide applicability, we also benchmark the performance of popular neural networks in our setting. We consider a variety of network archi-

tectures, the accuracy of which follows from [174, 173, 220]. We begin with a fully connected 3-layer network (NN-1) that considers around 118K model parameters. We also consider a convolutional neural network (NN-2) comprising 2 hidden layers, with 100 and 10 nodes, respectively. Lastly, we consider the two popular deep neural networks of LeNet [147] and VGG16 [213]. LeNet comprises 2 convolutional and fully connected layers, followed by max pool for convolutional layers, with approximately 431K parameters. On the other hand, VGG16 has 16 layers and contains fully-connected, convolutional, ReLU activation and max pool layers with around 138 million parameters. We rely on the standard MNIST [146] dataset to perform secure inference using NN-1 and LeNet, while the CIFAR-10 [141] dataset for NN-2 and VGG16 networks. Following prior works, here we operate on $\ell = 64$ bit rings where 13 bits are reserved for the fractional part of the number, the msb indicates the sign bit, and the rest of the bits represent the integer part of the number. We only estimate the performance of these NN algorithms here since their accuracy follows from prior works [174, 220]. The benchmarks for the different NNs appear in Table 5.8. As expected, the run time and communication of our protocols increase as the depth of the NNs increases from NN-1 to VGG16.

NN type	Preprocessing		Online		
	Time (s)	Com (MB)	Time (s)	Com (MB)	TP (queries/s)
NN-1	0.011	0.417	0.008	0.071	1010.86
NN-2	0.037	1.708	0.010	0.290	814.99
LeNet	0.560	35.898	0.053	6.298	152.21
VGG16	9.676	549.664	0.473	94.951	16.89

Table 5.8: NN inference.

5.12 Security proofs

The simulation-based security proofs for our protocols are presented in this section. The simulations for 5PC are provided in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Jmp}})$-hybrid model. The ideal functionality, $\mathcal{F}_{\mathsf{Jmp}}$ appears in Fig. 5.24. The two simulators considered are $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$, which denote the ideal-world malicious adversary and the ideal-world semi-honest adversary, respectively. We let $\mathcal{S}_{\mathcal{A}}^{P_i}$ denote the malicious simulator when party P_i is maliciously corrupt and $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ denote the simulator for the semi-honest corruption of party P_j. We omit the superscript when it is understood from the context.

We use the following strategy for simulating the computation of a function f. 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 $\mathcal{S}_{\mathcal{A}}$ obtains the input of $\mathcal{A}$, using the known keys, and sets the inputs of the honest parties to be 0. Note that $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ already knows the inputs of $\mathcal{A}_{\mathcal{H}}$. Since $\mathcal{S}_{\mathcal{A}}$ knows all the inputs, it can honestly carry out the computation and obtain all the intermediate values as required for simulating the view of $\mathcal{A}$. Further, on invoking the ideal functionality $\mathcal{F}_{\mathsf{5PC-FaF}}$ with $\mathcal{A}$'s input (and $\mathcal{A}_{\mathcal{H}}$'s input), $\mathcal{S}_{\mathcal{A}}$ can obtain the output of the function. $\mathcal{S}_{\mathcal{A}}$ proceeds to simulate the various sub-protocols in topological order using the aforementioned values (inputs of $\mathcal{A}$ ($\mathcal{A}_{\mathcal{H}}$), intermediate values and circuit output). A similar approach is taken by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ while ensuring that the messages sent to $\mathcal{A}_{\mathcal{H}}$ are consistent with that in the view received from $\mathcal{S}_{\mathcal{A}}$.

The simulation steps are provided separately for the sub-protocols to ensure modularity. Carrying out these simulation steps in the respective order results in simulating the computation of the desired function f. While emulating $\mathcal{F}_{\mathsf{Jmp}}$, if a CP is identified, the simulator stops the simulation at that step and continues with the simulation of 3PC using the respective semi-honest 3PC simulator.

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

$\mathcal{F}_{\mathsf{Jmp}}$ interacts with parties in $\mathcal{P}$ and adversary $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

- $\mathcal{F}_{\mathsf{Jmp}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from P_s for $s \in \{i, j\}$, while it receives $(\mathsf{Select}, \mathsf{CP})$ from $\mathcal{S}_{\mathcal{A}}$. Here, CP denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wishes to choose as the conflict pair. Let $P^{\star} \in \mathcal{P}$ denote the party corrupted by $\mathcal{S}_{\mathcal{A}}$.
- If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{CP} = \perp$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \perp, \mathsf{msg}_k = \mathsf{v}_i$.
- Else, if $P^{\star} \in \mathsf{CP}$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{CP}$.
- Else, set $\mathsf{CP} = \{P^{\star}, P\}$ where $P \in \mathsf{CP}$. Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{CP}$
- Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

$\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.24: Ideal functionality for Jmp.

5.12.1 Simulations for 5PC protocols

In this section, we describe the simulation steps for input sharing, multiplication and reconstruction, followed by the complete 5PC.

5.12.1.1 Sharing

The ideal functionality for Π_{Sh} (Fig. 5.4) appears in Fig. 5.25.

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

$\mathcal{F}_{\mathsf{Sh}}$ interacts with parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$.

– Receive $(\mathsf{Input}, \mathsf{v})$ from dealer $P_d \in \mathcal{P}$. Let $P^\star$ be the party corrupted by $\mathcal{S}_{\mathcal{A}}$.

– Receive $\mathsf{continue}$ or $\mathtt{abort}$ with $(\mathsf{Select}, \mathsf{C})$ from $\mathcal{S}_{\mathcal{A}}$. Here, C denotes pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as conflict pair.

– If received $\mathsf{continue}$, randomly pick $\alpha_{\mathsf{v}_{\mathsf{ij}}} \in \mathbb{Z}_{2^\ell}$, for $1 \leq i < j \leq 5$ and compute $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{1 \leq i < j \leq 5} \alpha_{\mathsf{v}_{\mathsf{ij}}}$. Set $\mathsf{msg}_s = (\beta_{\mathsf{v}}, \{\alpha_{\mathsf{v}_{\mathsf{ij}}}\}_{i \neq s, j \neq s})$, for each $P_s \in \mathcal{P}$.

– Else if received $\mathtt{abort}$, then:

– If $P^\star \in \mathsf{C}$, then set $\mathsf{CP} = \mathsf{C}$ and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

– Else set CP to include $P^\star$ and one other party from $\mathcal{P}$, and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

– $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.25: Ideal functionality for Π_{Sh}.

The simulator for the sharing protocol appears in Fig. 5.26.

Lemma 5.1 (Security) *Protocol Π_{Sh} (Fig. 5.4) realizes $\mathcal{F}_{\mathsf{Sh}}$ (Fig. 5.25) with computational security in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Jmp}})$-hybrid model against FaF adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$ controlling P_i, P_j respectively.*

Proof: Claim 1: the view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view.

This is argued as follows. When P_i is the dealer, $\mathcal{A}$'s view consists of the random shares of α_{v} generated using the random keys provided by $\mathcal{S}_{\mathcal{A}}^{P_i}$ while emulating $\mathcal{F}_{\mathsf{Setup}}$. This is indistinguishable from $\mathcal{A}$'s view in the real-world. When P_i is a non-dealer, $\mathcal{A}$'s view consists of a subset of the random shares of α_{v} generated using the random keys provided by $\mathcal{S}_{\mathcal{A}}^{P_i}$ while emulating $\mathcal{F}_{\mathsf{Setup}}$. Additionally, it also sees $\beta_{\mathsf{v}} = 0 + \alpha_{\mathsf{v}}$. Since, the missing shares of α_{v} at $\mathcal{A}$ are chosen randomly by $\mathcal{S}_{\mathcal{A}}^{P_i}$, β_{v} remains random, and hence the views are indistinguishable.

Claim 2: the view generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s real-world view, where $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ knows the input and output of $\mathcal{A}_{\mathcal{H}}$, and view sent by $\mathcal{S}_{\mathcal{A}}^{P_i}$.

This is argued as follows. If P_j is the dealer, the argument follows similar to before, and $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$'s view is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s view. If P_j is a non-dealer, then $\mathcal{A}_{\mathcal{H}}$'s view consists

of β_{v}, the six random shares of α_{v}, and among the four missing shares of α_{v}, it also sees three shares which are received as part of the view sent by $\mathcal{A}$ to $\mathcal{A}_{\mathcal{H}}$. Since $\mathcal{A}_{\mathcal{H}}$ still misses the share $\alpha_{\mathsf{v}_{\mathsf{ij}}}$, the β_{v} sent by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ remains random, and hence the views are indistinguishable. □

Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$

Malicious Simulation:

Preprocessing:

– $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Setup}}$ and gives the respective keys to $\mathcal{A}$. The shares of α_{v} that are held by $\mathcal{A}$ are sampled non-interactively using the shared keys. Other values ($\alpha_{\mathsf{v}_{\mathsf{ij}}}$ for $1 \leq i < j \leq 5$ and $\alpha_{\mathsf{v}_{\mathsf{ji}}}$ for $1 \leq j < i \leq 5$), not known to P_i, are sampled randomly.

Online:

– If P_i is the dealer, $\mathcal{S}_{\mathcal{A}}$ receives β_{v} from $\mathcal{A}$. Given the knowledge of all shares of α_{v}, $\mathcal{S}_{\mathcal{A}}$ obtains $\mathcal{A}$'s input as $\mathsf{v} = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}}$. Following this, $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}$ as one of the senders, to deliver β_{v} to all parties. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{Sh}}$ with $(\mathsf{Input}, \mathsf{v})$, and $\mathsf{continue}/\mathtt{abort}$ and $(\mathsf{Select}, \mathsf{CP})$.

– Else, $\mathcal{S}_{\mathcal{A}}$ honestly generates β_{v} by setting the input, v, of honest dealer as $\mathsf{v} = 0$. $\mathcal{S}_{\mathcal{A}}$ either sends β_{v} to $\mathcal{A}$ and/or emulates $\mathcal{F}_{\mathsf{Jmp}}$ to deliver β_{v} to all, with $\mathcal{A}$ either as the sender or receiver, depending on the identity of P_i. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{Sh}}$ with $\mathsf{continue}$ or $\mathtt{abort}$, and $(\mathsf{Select}, \mathsf{CP})$.

Semi-Honest Simulation:

Preprocessing:

– $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ receives the shared keys generated during $\mathcal{F}_{\mathsf{Setup}}$ from $\mathcal{S}_{\mathcal{A}}$, and the corresponding shares of α_{v}. The shares of α_{v} that are held by $\mathcal{A}_{\mathcal{H}}$, other than the ones held by $\mathcal{A}$, are sampled non-interactively using the shared keys. Shares not known to P_j are sampled randomly.

Online:

– If P_i is the dealer, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends the β_{v} received from $\mathcal{S}_{\mathcal{A}}$ to $\mathcal{A}_{\mathcal{H}}$ and/or emulates $\mathcal{F}_{\mathsf{Jmp}}$. Else, it performs these steps with a β_{v} generated by setting $\mathsf{v} = 0$.

Figure 5.26: Simulator for Π_{Sh} for sharing v.

5.12.1.2 Joint sharing

The simulator for the joint sharing protocol where two parties jointly share a value v in the preprocessing phase appears in Fig. 5.27. The simulations for joint sharing when the value to be shared is available in the online phase is similar.

Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$

Malicious Simulation:

– If P_i is one among the two dealers, $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}$ as one of the senders to send one share of α_{v} to one other party.

– Else if P_i is the recipient of the share of α_{v}, then $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}$ as the receiver.

– Else, there is nothing to simulate.

Semi-Honest Simulation:

– If P_j is one of the dealers, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send the share of α_{v} to one other honest party.

– Else, if P_j is the recipient of the share of α_{v}, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as the receiver.

– Else if P_j is neither the dealer nor the receiver, there is nothing to simulate.

Figure 5.27: Simulator for Π_{JSh} for sharing v.

Observe that view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view. This is because values received by $\mathcal{A}$ are random which is as per the real-world protocol. Similarly, view of $\mathcal{A}_{\mathcal{H}}$ generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from real-world view.

5.12.1.3 Reconstruction

The ideal functionality for Π_{Rec} (Fig. 5.7) appears in Fig. 5.28.

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

$\mathcal{F}_{\mathsf{Rec}}$ interacts with parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$.

– Receive $(\mathsf{Input}, [\![\mathsf{v}]\!]_s, P_i)$ from each $P_s \in \mathcal{P}$.

– Set $\mathsf{msg}_i = \beta_{\mathsf{v}} - \sum_{1 \le i < j \le 5} \alpha_{\mathsf{v}_{\mathsf{ij}}}$ and $\mathsf{msg}_s = \perp$ for $P_s \in \mathcal{P} \setminus \{P_i\}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

– $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.28: Ideal functionality for Π_{Rec}.

The simulator for the reconstruction protocol appears in Fig. 5.29.

Lemma 5.2 (Security) *Protocol Π_{Rec} (Fig. 5.7) realizes $\mathcal{F}_{\mathsf{Rec}}$ (Fig. 5.28) with computational security in the $\mathcal{F}_{\mathsf{Setup}}$-hybrid model against FaF adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$ controlling P_i, P_j respectively.*

Proof: The view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view. This is argued as follows. $\mathcal{A}$'s view consists of random $\alpha_{\mathsf{v}_{\mathsf{pq}}}$ for $1 \leq p < q \leq 5, p \neq i, q \neq i$ such that one share, say, $\alpha_{\mathsf{v}_{\mathsf{ij}}}$ (unknown to $\mathcal{A}$) is adjusted as $\alpha_{\mathsf{v}_{\mathsf{ij}}} = \beta_{\mathsf{v}} - \mathsf{v} - \sum_{1 \leq p < q \leq 5, p \neq i, q \neq j} \alpha_{\mathsf{v}_{\mathsf{pq}}}$ to ensure reconstruction of correct output. Since these missing shares are chosen randomly by $\mathcal{S}_{\mathcal{A}}^{P_i}$, the β_{v} remains random and, the views are indistinguishable. Similarly, the view generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s real-world view, since $\mathcal{A}_{\mathcal{H}}$ still misses one random share $\alpha_{\mathsf{v}_{\mathsf{ij}}}$, which keeps β_{v} random. □

Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$

Malicious Simulation:

– To simulate reconstruction towards $\mathcal{A}$:

- Invoke $\mathcal{F}_{\mathsf{Rec}}$ with $(\mathsf{Input}, [\![\mathsf{v}]\!]_i)$.

- $\mathcal{S}_{\mathcal{A}}$ sets a missing share of $\alpha_{\mathsf{v}_{\mathsf{ij}}}$ of v, not held by P_i (and P_j) as $\alpha_{\mathsf{v}_{\mathsf{ij}}} = \beta_{\mathsf{v}} - \mathsf{v} - \sum_{1 \leq p < q \leq 5, p \neq i, q \neq j} \alpha_{\mathsf{v}_{\mathsf{pq}}}$, where $\alpha_{\mathsf{v}_{\mathsf{pq}}}$ were sampled using the shared keys, and v is the output obtained by $\mathcal{S}_{\mathcal{A}}$ from the ideal functionality.

- $\mathcal{S}_{\mathcal{A}}$ sends $\alpha_{\mathsf{v}_{\mathsf{ij}}}$ and its hash to $\mathcal{A}$ on behalf of the honest parties that hold $\alpha_{\mathsf{v}_{\mathsf{ij}}}$. $\mathcal{S}_{\mathcal{A}}$ sends the other shares of α_{v} which include $\alpha_{\mathsf{vik}}, \alpha_{\mathsf{vil}}, \alpha_{\mathsf{vim}}$ (and were sampled randomly), together with its hash to $\mathcal{A}$ on behalf of honest parties that hold these shares.

Semi-Honest Simulation:

– $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ receives the view from $\mathcal{S}_{\mathcal{A}}$. To simulate reconstruction towards $\mathcal{A}_{\mathcal{H}}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends the missing shares and their hashes to $\mathcal{A}_{\mathcal{H}}$ on behalf of the honest parties by using these values as present in the view received from $\mathcal{S}_{\mathcal{A}}$.

Figure 5.29: Simulator for Π_{Rec} of output $[\![\mathsf{v}]\!]$.

5.12.1.4 Multiplication

The ideal functionality for Π_{Mul} (Fig. 5.8) appears in Fig. 5.30. Due to the asymmetry in our multiplication protocol, we consider the following two cases for simulation– (i) when the maliciously corrupt P_i is one among P_1, P_2, P_3, and (ii) when the maliciously corrupt P_i is one among P_4, P_5. The simulator for case(i) appears in Fig. 5.31.

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

$\mathcal{F}_{\mathsf{Mul}}$ interacts with parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$.

– Receive $(\mathsf{Input}, [\![\mathsf{a}]\!]_s, [\![\mathsf{b}]\!]_s, [\alpha_{\mathsf{z}}]_s)$ from $P_s \in \mathcal{P}$. Let $P^\star$ be the malicious party controlled by $\mathcal{S}_{\mathcal{A}}$.

– Receive $\mathsf{continue}$ or $\mathtt{abort}$ with $(\mathsf{Select}, \mathsf{C})$ from $\mathcal{S}_{\mathcal{A}}$. Here, C denotes pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as conflict pair.

– If received $\mathsf{continue}$, compute $[\![\mathsf{z}]\!]$ where $\mathsf{z} = \mathsf{ab} + \alpha_{\mathsf{z}}$. Set $\mathsf{msg}_s = [\![\mathsf{z}]\!]_s$, for each $P_s \in \mathcal{P}$.

– Else if received $\mathtt{abort}$, then:

– If $P^\star \in \mathsf{C}$, then set $\mathsf{CP} = \mathsf{C}$ and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

– Else set CP to include $P^\star$ and one other party from $\mathcal{P}$, and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

– $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.30: Ideal functionality for Π_{Mul}.

Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$

Malicious Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.

Online:

– $\mathcal{S}_{\mathcal{A}}$ honestly generates shares of β_{z} on behalf of honest parties.

– $\mathcal{S}_{\mathcal{A}}$ simulates *send* of Jmp with $\mathcal{A}$ as one of the senders to send the missing share of β_{z} to the other two online parties (P_1, P_2, P_3). $\mathcal{S}_{\mathcal{A}}$ simulates *send* of Jmp with $\mathcal{A}$ as the receiver to send the missing shares of β_{z} to $\mathcal{A}$ on behalf of the honest parties.

Verification:

– $\mathcal{S}_{\mathcal{A}}$ honestly generates hash on all β_{z}s involved in verification on behalf of the honest online parties, and sends the hash to $\mathcal{A}$.

– If $\mathcal{A}$ sends an inconsistency bit $\mathsf{b} = 0$, $\mathcal{S}_{\mathcal{A}}$ simulates *send* and *verify* of Jmp with $\mathcal{A}$ as one of the senders to send β_{z} to the offline parties (P_4, P_5), if $P_i \in \{P_1, P_2\}$. This is followed by simulation of *verify* of Jmp towards $\mathcal{A}$.

– Else, if $\mathcal{A}$ sends an inconsistency bit $\mathsf{b} = 1$, $\mathcal{S}_{\mathcal{A}}$ simulates the binary search where hashes are sent until $\mathcal{A}$ broadcasts an inconsistency bit with $\mathsf{b} = 0$ and levels L_p, L_{p+1} are identified. $\mathcal{S}_{\mathcal{A}}$ simulates *send* and *verify* of Jmp with $\mathcal{A}$ as one of the senders if $P_i \in \{P_1, P_2\}$ to send β_{z} up to level L_p. This is followed by simulation of *verify* of Jmp towards $\mathcal{A}$ for β_{z}s up to level L_{p+1}. If the simulation of *verify* of latter Jmp did not output a CP, $\mathcal{S}_{\mathcal{A}}$ sends the identity of P_j to $\mathcal{A}$.

– Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{Mul}}$ with $(\mathsf{Input}, [\![\mathsf{a}]\!]_i, [\![\mathsf{b}]\!]_i, [\alpha_{\mathsf{z}}]_i)$, and

$\mathsf{continue/abort}$ and $(\mathsf{Select}, \mathsf{CP})$.

Semi-Honest Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.

Online: If P_j is one of the online parties, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates *send* of Jmp with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send the missing share of β_{z} to the remainder honest online party. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates *send* of Jmp with $\mathcal{A}_{\mathcal{H}}$ as the receiver to send the missing share of β_{z} to $\mathcal{A}_{\mathcal{H}}$ on behalf of the honest party.

Verification: If P_j is one of the online parties, then

– $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ honestly generates hash on all β_{z}s involved in verification on behalf of the honest online parties, and sends the hash to $\mathcal{A}_{\mathcal{H}}$.

– Depending on the bit obtained in the view from $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ either proceeds with simulating Jmp with $\mathcal{A}_{\mathcal{H}}$ as one of the senders if $P_j \in \{P_1, P_2\}$ for sending β_{z} towards offline parties, or it simulates the hash-based consistency check. For the latter, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ recursively performs the hash exchange until levels L_p, L_{p+1} as present in the view of $\mathcal{S}_{\mathcal{A}}$ are identified. Following this, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates *send* and *verify* of Jmp with $\mathcal{A}_{\mathcal{H}}$ as one of the senders if P_j is one among P_1 or P_2 for sending β_{z} up to level L_p to offline parties. Then, simulation of *verify* of Jmp towards $\mathcal{A}_{\mathcal{H}}$ for β_{z}s up to level L_{p+1} is performed.

If P_j is one of the offline parties, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates the similar steps as above which are carried out after the hash-based consistency check.

Figure 5.31: Simulator for Π_{Mul} when $P_i \in \{P_1, P_2, P_3\}$.

The simulator for case(ii) appears in Fig. 5.32.

Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$

Malicious Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.

Online: There is nothing to simulate.

Verification:

– $\mathcal{S}_{\mathcal{A}}$ honestly generates β_{z} on behalf of honest parties.

– $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}$ as the receiver to send β_{z} to $\mathcal{A}$ on behalf of the honest parties.

– Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{Mul}}$ with $(\mathsf{Input}, [\![\mathsf{a}]\!]_i, [\![\mathsf{b}]\!]_i, [\alpha_{\mathsf{z}}]_i)$, and $\mathsf{continue/abort}$ and $(\mathsf{Select}, \mathsf{CP})$.

Semi-Honest Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.

Online:

– If P_j is one of the online parties:

- $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send the missing share of β_{z} (generated honestly) to the other two online parties. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as the receiver to send the missing shares of β_{z} to $\mathcal{A}_{\mathcal{H}}$ on behalf of the honest parties.

– If P_j is one of the offline parties, there is nothing to simulate.

Verification:

– If P_j is one of the online parties, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends the hash of all β_{z} in this segment to $\mathcal{A}_{\mathcal{H}}$ and emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send β_{z} to the honest offline party.

– If P_j is one of the offline parties, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as receiver to send β_{z} (reused from the view received from $\mathcal{S}_{\mathcal{A}}$) to $\mathcal{A}_{\mathcal{H}}$ on behalf of honest parties.

Figure 5.32: Simulator for Π_{Mul} when $P_i \in \{P_4, P_5\}$.

Lemma 5.3 (Security) *Protocol Π_{Mul} (Fig. 5.8) realizes $\mathcal{F}_{\mathsf{Mul}}$ (Fig. 5.30) with computational security in the $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{Jmp}}, \mathcal{F}_{\mathsf{MulPre}})$-hybrid model against FaF adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$ controlling P_i, P_j respectively.*

Proof: We argue indistinguishability in the following two cases.

Case 1: When the maliciously corrupt P_i is one among P_1, P_2, P_3. Observe that the view generated in this case by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view. This is because $\mathcal{A}$ receives random shares of β_{z} which are generated honestly by the simulator. Since $\mathcal{A}$ still misses one share of the mask α_{z}, the β_{z} received via $\mathcal{F}_{\mathsf{Jmp}}$ remains random. Hence, the views are indistinguishable. A similar argument applies to $\mathcal{A}_{\mathcal{H}}$'s view being indistinguishable.

Case 2: When the maliciously corrupt P_i is one among P_4, P_5. Similar to case 1, the real-world view of $\mathcal{A}$ is indistinguishable from the view generated by $\mathcal{S}_{\mathcal{A}}$ since $\mathcal{A}$ misses one share of the α_{z} which keeps β_{z} random. A similar argument, as before, holds for the indistinguishability of the view of $\mathcal{A}_{\mathcal{H}}$. □

Preprocessing We next prove that Π_{MulPre} securely computes $\mathcal{F}_{\mathsf{MulPre}}$ in the (1, 1)-FaF model in 5PC setting.

Lemma 5.4 *Protocol $\Pi_{\mathsf{CheatIdentify}}$ (Fig. 5.12) securely computes $\mathcal{F}_{\mathsf{CheatIdentify}}$ (Fig. 5.11) over field $\mathbb{F}$ in the $(1,1)$-FaF model with error $\leq \frac{2\log L+4}{\mathbb{F}-5}$ in the 5PC setting .*

Proof: Let $\mathcal{S}_{\mathcal{A}}$ be the ideal world malicious simulator, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world semi-honest simulator, $\mathcal{A}$ be the real-world malicious adversary and $\mathcal{A}_{\mathcal{H}}$ be the semi-honest real-world adversary. Consider the following cases.

Case 1: P_i **is corrupt.** In this case $\mathcal{S}_{\mathcal{A}}$ receives P_i's inputs and honest parties $[\cdot]$-shares of c. This implies that $\mathcal{S}_{\mathcal{A}}$ can perfectly simulate the opening of $[b]$ and $q(\mathsf{r})$ since it has the honest parties' $[\cdot]$-shares of c, and receives honest parties' $[\cdot]$-shares of points on $q(\cdot)$ from $\mathcal{A}$ during the simulation. We next show how to simulate the opening of $\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r})$. For this, since $\mathcal{S}_{\mathcal{A}}$ knows the inputs of P_i, it knows the actual values of $\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r})$. Thus, $\mathcal{S}_{\mathcal{A}}$ is only required to choose random values for shares of the honest parties while ensuring that together with P_i's shares, it opens to the correct values.

To see that the view of $\mathcal{A}$ is the same here as in the real execution, observe that for each $e \in \{1, 2\}$,

$$\boldsymbol{f}_e(\mathsf{r}) = \boldsymbol{\lambda}_0(\mathsf{r}) \diamond \boldsymbol{f}_e(0) + \boldsymbol{\lambda}_1(\mathsf{r}) \diamond \boldsymbol{f}_e(1) + \boldsymbol{\lambda}_2(\mathsf{r}) \diamond \boldsymbol{f}_e(2) \tag{5.5}$$

where $\boldsymbol{\lambda}_0(\mathsf{r}), \boldsymbol{\lambda}_1(\mathsf{r}), \boldsymbol{\lambda}_2(\mathsf{r})$ are the Lagrange coefficients. Since shares of $\boldsymbol{f}_e(0)$ held by honest parties are random under the constraint that together with P_i's shares they open to $\boldsymbol{f}_e(0)$, so are the shares of $\boldsymbol{f}_e(\mathsf{r})$. Thus, the distribution is the same in both executions. If some honest party outputs reject, then $\mathcal{A}$ broadcasts an index j, which $\mathcal{S}_{\mathcal{A}}$ forwards to $\mathcal{F}_{\mathsf{CheatIdentify}}$. If $\mathsf{out} = \mathsf{reject}$, but honest parties output accept, then $\mathcal{S}_{\mathcal{A}}$ outputs fail and halts. Observe that when $\mathcal{S}_{\mathcal{A}}$ does not output fail, the simulation is perfect. The main difference is when $\mathcal{S}_{\mathcal{A}}$ outputs fail. This event occurs when Equation (5.1) does not hold, yet honest parties output accept. This occurs with probability $\leq \frac{2\log L+4}{\mathbb{F}-5}$, which is the error probability of the simulation. Finally, $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Subcase: P_j is semi-honest. P_j's view consists of (i) shares of $q(1), q(2), q(3)$ received in each of the $\log \bar{L} - 1$ iterations, (ii) shares of $q(0), q(1), \ldots, q(4)$ received in the last iteration, and, (iii) the shares received for reconstructing $b, q(\mathsf{r}), \boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r})$. While (i), (ii) are received as part of the view of $\mathcal{S}_{\mathcal{A}}$, (iii) can be simulated by sending random shares under the constraint that the reconstructed values are consistent with the ones in the view received from $\mathcal{S}_{\mathcal{A}}$. Thus the simulation is perfect.

Case 2: P_i **is honest and** P_j **is corrupt.** In this case, $\mathcal{S}_{\mathcal{A}}$ receives accept from $\mathcal{F}_{\mathsf{CheatIdentify}}$. This implies that although $\mathcal{S}_{\mathcal{A}}$ does not know the input, it knows that b should be 0 in each iteration and $q(\mathsf{r})$ should equal $g(\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}))$ in the last iteration, unless P_j misbehaves. Since $\mathcal{S}_{\mathcal{A}}$ knows P_j's shares of the inputs, it can simulate the openings correctly. Elaborately, for each sharing of $q(1), q(2)$ and $q(3)$ (and $\boldsymbol{f}_1(0), \boldsymbol{f}_2(0), q(0), \ldots, q(4)$ in the last step) in the simulation, $\mathcal{S}_{\mathcal{A}}$ sends random shares on behalf of P_i to $\mathcal{A}$. Since $\mathcal{S}_{\mathcal{A}}$ knows P_j's shares of $c, q(1), q(2)$, it can computes its shares of $b_l = c - q(1) - q(2)$. It then chooses the honest parties shares under the constraint that $b = \sum_{l=1}^{\log L} \gamma_l b_l$ will reconstruct to 0. Following this, $\mathcal{S}_{\mathcal{A}}$ uses P_j's shares of $\boldsymbol{f}_1(e), \ldots, \boldsymbol{f}_L(e)$ for $e \in \{1, 2\}$, and $q(1), q(2), q(3)$ to compute P_j's shares of

$\boldsymbol{f}_1(\mathsf{r}), \ldots, \boldsymbol{f}_L(\mathsf{r})$ and $q(\mathsf{r})$. Then, it can simulate the next iteration as before. Finally, $\mathcal{S}_{\mathcal{A}}$ uses P_j's shares of $\boldsymbol{f}_1(0), \boldsymbol{f}_1(1), \boldsymbol{f}_1(2), \boldsymbol{f}_2(0), \boldsymbol{f}_2(1), \boldsymbol{f}_2(2)$ and $q(0), \ldots, q(4)$ to compute P_j's shares of $\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}), q(\mathsf{r})$. $\mathcal{S}_{\mathcal{A}}$ simulates the opening of $b, \boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}), q(\mathsf{r})$ as follows.

– To simulate the opening of b, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for the honest parties under the constraint that all the shares together will reconstruct to 0.

– To simulate the opening of $\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r})$, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for the honest parties.

– To simulate the opening of $q(\mathsf{r})$, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for the honest parties under the constraint that the reconstructed $q(\mathsf{r})$ will satisfy the equation: $q(\mathsf{r}) = g(\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}))$.

If $\mathcal{A}$ sends consistent shares, $\mathcal{S}_{\mathcal{A}}$ sends $\mathsf{out} = \mathsf{accept}$ to $\mathcal{F}_{\mathsf{CheatIdentify}}$. Else, since $\mathcal{S}_{\mathcal{A}}$ knows P_i's shares, it can compute the message that should have been sent by $\mathcal{A}$, and identifies the cheater on behalf of P_i. $\mathcal{S}_{\mathcal{A}}$ sends reject with index j to $\mathcal{F}_{\mathsf{CheatIdentify}}$ in this case.

We claim that $\mathcal{A}$'s view in the real and ideal execution is identically distributed. $\mathcal{A}$'s view consists of (i) shares sent by P_i for points on q, (ii) shares for points $\boldsymbol{f}_1(0), \boldsymbol{f}_2(0)$, (iii) the opened b, and (iv) the opened $\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}), q(\mathsf{r})$. Shares in (i) and (ii) are uniformly distributed, with respect to (iii), $\mathcal{A}$ sees random shares which open to 0 in both worlds. Finally, the claim in (iv) follows from Equation (5.5), similar to that in case 1, where $\boldsymbol{f}_e(\mathsf{r})$ for $e \in \{1, 2\}$ is randomly distributed in the ideal as well as the real world. Given that $\boldsymbol{f}_e(\mathsf{r})$ for $e \in \{1, 2\}$ is random, we obtain $q(\mathsf{r})$ being random as long as $q(\mathsf{r}) = g(\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}))$ holds.

Subcase: P_i is semi-honest. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ has all inputs of P_i. Thus, the simulation can be carried out honestly, taking into consideration the view received from $\mathcal{S}_{\mathcal{A}}$. Thus, the simulation is perfect.

Subcase: P_k is semi-honest. This is similar to case 2. Since P_i is honest, b should be 0 in each iteration and $q(\mathsf{r})$ should equal $g(\boldsymbol{f}_1(\mathsf{r}), \boldsymbol{f}_2(\mathsf{r}))$ in the last iteration. Since $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ knows P_k's shares of the inputs, it can simulate the openings correctly. Thus, the simulation is perfect.

□

Lemma 5.5 *Protocol Π_{Verify} (Fig. 5.14) securely computes $\mathcal{F}_{\mathsf{Verify}}$ (Fig. 5.13) over field $\mathbb{F}$ in the $(1,1)$-FaF model with error $\log m \cdot \frac{1}{\mathbb{F}}$ in the $(\mathcal{F}_{\mathsf{MiniMPC}}, \mathcal{F}_{\mathsf{CheatIdentify}})$-hybrid model in the 5PC setting.*

Proof: Let $\mathcal{S}_{\mathcal{A}}$ be the ideal world malicious simulator, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world semi-honest simulator and let $\mathcal{A}$ be the real world malicious adversary and $\mathcal{A}_{\mathcal{H}}$ be the real world semi-honest adversary. $\mathcal{S}_{\mathcal{A}}$ is invoked by $\mathcal{F}_{\mathsf{Verify}}$ which sends it the corrupted party's shares of $(\mathsf{x}_k, \mathsf{y}_k, \mathsf{z}_k)_{k=1}^m$ and $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$ and $\mathsf{d}_k = \mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k$ for $k \in \{1, 2, \ldots, m\}$. Further, $\mathcal{F}_{\mathsf{Verify}}$ sends $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ the shares for $P_{\mathcal{H}}$, which is the semi-honest party.

Random $\theta_1, \ldots, \theta_m \in \mathbb{F}$ are generated. $\mathcal{S}_{\mathcal{A}}$ plays the role of $\mathcal{F}_{\mathsf{CheatIdentify}}$ and $\mathcal{F}_{\mathsf{MiniMPC}}$. Similar to the proof of Theorem 5.4, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for corrupted party for each ψ^j, where P_j is honest and hands these to $\mathcal{A}$. Then, $\mathcal{S}_{\mathcal{A}}$ receives the honest parties' shares for ψ^i, where P_i is the maliciously corrupt party. If the shares dealt by $\mathcal{A}$ are inconsistent, then the consistency check takes care of this. The presence of honest majority enables $\mathcal{S}_{\mathcal{A}}$ to use the honest parties' shares to compute ψ^i for the corrupt P_i and its shares. Thus, for each $i \in \{1, 2, \ldots, 5\}$, $\mathcal{S}_{\mathcal{A}}$ can simulate $\mathcal{F}_{\mathsf{CheatIdentify}}$, handing accept or reject to $\mathcal{A}$, accordingly. If the output is reject for any $i \in \{1, 2, \ldots, 5\}$, then $\mathcal{A}$ sends index of a party P_j to $\mathcal{S}_{\mathcal{A}}$, which together with P_i forms a disputed pair of parties. Then, $\mathcal{S}_{\mathcal{A}}$ sends $\mathsf{reject}, (i, j)$ to $\mathcal{F}_{\mathsf{Verify}}$, outputs whatever $\mathcal{A}$ outputs and halts.

If the simulation has not ended with a reject, then it means that all ψ^i's are correct. Thus, $\mathcal{S}_{\mathcal{A}}$ can compute $\beta = \sum_{k=1}^{m} \theta_k \cdot (\mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k) = \sum_{k=1}^{m} \theta_k \cdot \mathsf{d}_k$ and choose random shares for the honest parties, given the value of β and the corrupted party's shares (known to $\mathcal{S}_{\mathcal{A}}$). Using these shares, $\mathcal{S}_{\mathcal{A}}$ simulates the reconstruction procedure. Consider the following cases.

– If $\mathcal{A}$ sent incorrect shares, causing the opening of β to fail, then $\mathcal{S}_{\mathcal{A}}$ takes the first pair of parties P_i, P_j for which pair-wise inconsistency occurred, and sends $\mathsf{reject}, (i, j)$ to $\mathcal{F}_{\mathsf{Verify}}$, outputs whatever $\mathcal{A}$ outputs and halts.

– If $\beta = 0$: if $\mathsf{out} = \mathsf{reject}$ (honest parties output accept in this case), $\mathcal{S}_{\mathcal{A}}$ outputs fail and halts; if $\mathsf{out} = \mathsf{accept}$, $\mathcal{S}_{\mathcal{A}}$ sends accept to $\mathcal{F}_{\mathsf{Verify}}$, outputs whatever $\mathcal{A}$ outputs and halts.

– If $\beta \neq 0$, simulation proceeds to the binary search, where $\mathcal{S}_{\mathcal{A}}$ simulates each steps as described so far. If a pair of disputed parties is located, then it is sent to $\mathcal{F}_{\mathsf{Verify}}$. If honest parties output accept, then $\mathcal{S}_{\mathcal{A}}$ outputs fail (here it must hold that $\mathsf{out} = \mathsf{reject}$, since otherwise the simulation would not have reached the binary search phase). If parties found an incorrect triple $\mathsf{x}_{\bar{k}}, \mathsf{y}_{\bar{k}}, \mathsf{z}_{\bar{k}}$ such that $\mathsf{z}_{\bar{k}} \neq \mathsf{x}_{\bar{k}} \cdot \mathsf{y}_{\bar{k}}$ without identifying a disputed pair, then $\mathcal{S}_{\mathcal{A}}$ asks $\mathcal{F}_{\mathsf{Verify}}$ to find such a pair by sending it $\bar{k}$. Upon receiving (i, j) from $\mathcal{F}_{\mathsf{Verify}}$, $\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{F}_{\mathsf{MiniMPC}}$, handing (i, j) to $\mathcal{A}$. Finally, $\mathcal{S}_{\mathcal{A}}$ outputs whatever $\mathcal{A}$ outputs. Note that an event where the $\bar{k}^{\text{th}}$ triple is correct is not possible, because in this case β must be equal to 0.

$\mathcal{A}$'s view consists of (i) random shares of β^j for each honest party P_j, (ii) message sent by $\mathcal{F}_{\mathsf{CheatIdentify}}$, (iii) the revealed β, and (iv) message from $\mathcal{F}_{\mathsf{MiniMPC}}$. The argument for identical distribution of $\mathcal{A}$'s view in (i), (ii), (iii) follows from the proof of Theorem 5.4. For (iv), since $\mathcal{S}_{\mathcal{A}}$ receives a pair of parties with conflicting views in the computation of the $\bar{k}^{\text{th}}$ triple from $\mathcal{F}_{\mathsf{Verify}}$, it can simulate the role of $\mathcal{F}_{\mathsf{MiniMPC}}$ perfectly. Thus, the only difference between the simulation and real-execution is the event where $\mathcal{S}_{\mathcal{A}}$ outputs fail. This happens when $\exists k \in \{1, 2, \ldots, m\} : \mathsf{d}_k \neq 0$ (which is why $\mathsf{out} = \mathsf{reject}$) but the parties eventually output accept. This occurs when $\beta = 0$

in one of binary search steps. Since there are $\log m$ steps and $\Pr[\beta = 0] = \frac{1}{\mathbb{F}}$ in each step, we have that $\Pr[\mathsf{fail}] \leq \frac{\log m}{\mathbb{F}}$, which is the error in the simulation.

Following this, $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ to simulate the view for $\mathcal{A}_{\mathcal{H}}$. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ chooses random shares for corrupted party for each ψ^j, where P_j is honest and hands these to $\mathcal{A}_{\mathcal{H}}$. Then, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ receives the honest parties' shares for $\psi^{\mathcal{H}}$, where $P_{\mathcal{H}}$ is the semi-honest party. The presence of honest majority enables $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ to use the honest parties' shares to compute $\psi^{\mathcal{H}}$ for $P_{\mathcal{H}}$ and its shares. Thus, for each $i \in \{1, 2, \ldots, 5\}$, $\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{F}_{\mathsf{CheatIdentify}}$, handing accept or reject to $\mathcal{A}_{\mathcal{H}}$ according to $\mathcal{S}_{\mathcal{A}}$'s view. If the output is reject for any $i \in \{1, 2, \ldots, 5\}$, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends $\mathsf{reject}, (i, j)$ to $\mathcal{A}_{\mathcal{H}}$, as present in $\mathcal{S}_{\mathcal{A}}$'s view. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates the reconstruction procedure for β using shares received from $\mathcal{A}_{\mathcal{H}}$. Now, depending on the view received from $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends (i, j) or accept to $\mathcal{A}_{\mathcal{H}}$. The argument for indisntinguishability of the views of $\mathcal{A}_{\mathcal{H}}$ in real and ideal world follows similar to the argument for $\mathcal{A}$. □

Lemma 5.6 *Protocol* Π_{MulPre} *(Fig. 5.16) securely computes* $\mathcal{F}_{\mathsf{MulPre}}$ *(Fig. 5.15) over the field* $\mathbb{F}$ *or ring* $\mathbb{Z}_{2^\ell}$ *in the* $(1, 1)$*-FaF model in the* $\mathcal{F}_{\mathsf{Verify}}$*-hybrid model in the 5PC setting.*

Proof: Consider the case of a corrupt P_1. $\mathcal{S}_{\mathcal{A}}$ generates $[\cdot]$-shares for $\{\mathsf{x}_k, \mathsf{y}_k, \mathsf{r}_k\}_{k=1}^m$, and learns these values on clear. Step 3 of the protocol is simulated by sending random values to $\mathcal{A}$. $\mathcal{S}_{\mathcal{A}}$ also computes the secret $\mathsf{x}_k \cdot \mathsf{y}_k$ for $k \in \{1, 2, \ldots, m\}$. If inconsistent shares are received in step 4 from $\mathcal{A}$, then $\mathcal{S}_{\mathcal{A}}$ detects the inconsistency, and the simulation outputs a pair of conflicting parties. Else, if the shares are consistent but the correct output is not received, $\mathcal{S}_{\mathcal{A}}$ computes the difference between these values and simulates $\mathcal{F}_{\mathsf{Verify}}$. If cheating took place, then it sens reject and $\mathsf{d}_k \neq 0$ to $\mathcal{A}$. Then, it waits to receive from $\mathcal{A}$ either a pair of conflicting parties or a request to $\mathcal{F}_{\mathsf{Verify}}$ to find such a pair. In the latter case, $\mathcal{S}_{\mathcal{A}}$ finds such a pair of conflicting parties by computing the messages that should have been sent by the corrupted party and compares it with what was received from $\mathcal{A}$. Then, $\mathcal{S}_{\mathcal{A}}$ sends the obtained pair to $\mathcal{A}$. If no cheating took place, then $\mathcal{S}_{\mathcal{A}}$ sends accept to $\mathcal{A}$. Following this, $\mathcal{A}$ can decide to reject, in which case a pair of conflicting parties is sent as output. Observe that since $\mathcal{A}$'s view consists of random shares in both the worlds, the views are identical. Then, $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$. Simulation by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ for a semi-honest party follows trivially as there are no messages to simulate other than those from P_1 which are already received as part of $\mathcal{A}$'s view.

Cases where other parties are corrupt can be simulated trivially. Simulation for semi-honest P_1 also follows. □

5.12.1.5 The complete 5PC

The ideal functionality for computing a function f via $(1,1)$-FaF secure 5PC appears in Fig. 5.33.

Overview of the simulation steps Observe that the complete 5PC protocol begins with the input sharing phase, followed by an evaluation phase where addition and multiplication gates are evaluated and concludes with a reconstruction phase. For each of these phases, we use the simulation steps described above depending on the identity of the maliciously corrupt P_i and a semi-honest P_j. The simulation proceeds as follows. The simulator is able to extract malicious $\mathcal{A}$'s input while performing the simulation steps for input sharing, and knows $\mathcal{A}_{\mathcal{H}}$'s input. Thus, it can invoke the ideal functionality, $\mathcal{F}_{\mathsf{5PC-FaF}}$ (Fig. 5.33), to obtain the output of the function being simulated. Simulation is not required for addition gates as it is a local operation. For multiplication gates, the simulation steps as described for multiplication are invoked. Observe that in all steps, the view of $\mathcal{A}$, as generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$, is indistinguishable from its real-world view. Similar is the case for $\mathcal{A}_{\mathcal{H}}$. If at any step, $\mathcal{F}_{\mathsf{Jmp}}$ outputs a CP, 5PC simulation stops and the rest of the steps are simulated using the semi-honest 3PC simulator. Steps for share conversion have to be simulated towards $\mathcal{A}_{\mathcal{H}}$, where the simulator carries out steps as per the honest protocol execution, reusing the shares held by $\mathcal{A}$, wherever necessary. Finally, for reconstructing the output, the simulator uses the output received from $\mathcal{F}_{\mathsf{5PC-FaF}}$ to adjust the value of the missing share that has to be sent to $\mathcal{A}$ and $\mathcal{A}_{\mathcal{H}}$. Indistinguishability of the views follows from the indistinguishability of the views for each of the phases. Thus, the view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view, and the view generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s real-world view.

Functionality $\mathcal{F}_{\mathsf{5PC-FaF}}$

$\mathcal{F}_{\mathsf{5PC-FaF}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$. Let x_s, y_s be the input and output corresponding to a party P_s respectively, i.e. $(y_1, y_2, y_3, y_4, y_5) = f(x_1, x_2, x_3, x_4, x_5)$.

- $\mathcal{F}_{\mathsf{5PC-FaF}}$ receives (Input, x_s) from $P_s \in \mathcal{P}$ and computes $(y_1, y_2, y_3, y_4, y_5) = f(x_1, x_2, x_3, x_4, x_5)$.

Output: Send (Output, y_s) to $P_s \in \mathcal{P}$.

$\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 5.33: Ideal functionality for evaluating f in 5PC $(1,1)$-FaF Model.

Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$

Malicious Simulation:

– $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Setup}}$ to generate common PRF keys.

– $\mathcal{S}_{\mathcal{A}}$ invokes the simulator for input sharing and extracts $\mathcal{A}$'s input. $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\mathsf{5PC-FaF}}$ on $\mathcal{A}$'s input to obtain the function output v.

– For addition operations, there is nothing to simulate. For multiplications, $\mathcal{S}_{\mathcal{A}}$ invokes the simulator for multiplication.

– $\mathcal{S}_{\mathcal{A}}$ invokes the reconstruction simulator to reconstruct output v.

– $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Semi-Honest Simulation:

– $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ invokes the simulator for input sharing.

– For addition operations, there is nothing to simulate. For multiplications, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ invokes the simulator for multiplication.

– $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ invokes the reconstruction simulator to reconstruct output v.

Figure 5.34: Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$ for $\mathsf{5PC-FaF}$.

Theorem 5.1 *Assuming collision resistant hash functions exist, protocol* $\mathsf{5PC-FaF}$ *(Fig. 5.17) realizes* $\mathcal{F}_{\mathsf{5PC-FaF}}$ *(Fig. 5.33) with computational security in the* $\mathcal{F}_{\mathsf{Setup}}$*-hybrid model with* $(1,1)$*-FaF security.*

Proof: The view of the adversaries generated by the simulators is indistinguishable from their real-world views. The indistinguishability of the views from input sharing and multiplication follows from Lemma 5.1 and Lemma 5.3, respectively. With respect to reconstruction, on obtaining the output from $\mathcal{F}_{\mathsf{5PC-FaF}}$, the simulators either simulate the reconstruction steps (see Lemma 5.2 for indistinguishability argument), or execute the simulator for semi-honest 3PC. In both cases, the simulated view is indistinguishable from the real-world view. □

5.12.2 Simulations for building blocks

In this section, we describe the simulation steps for the building blocks described in §5.8. We begin with the simulation steps for multi-input multiplication, dot product, bit to arithmetic, bit injection, bit extraction and arithmetic to Boolean.

Since the multi-input multiplication and dot product protocol are very similar to the multiplication protocol, we omit simulation steps for the same. Further, observe that the protocol for bit to arithmetic essentially invokes the joint sharing and multiplication protocols. Hence,

simulation steps for bit to arithmetic involves executing the simulation steps for joint sharing and multiplication in the order in which they appear in the protocol. Indistinguishability follows from the indistinguishability of the simulation steps in the underlying protocols. Similar to bit to arithmetic, bit injection involves an invocation of bit to arithmetic followed by a multiplication. Hence, the simulation steps follow from the simulation of the underlying protocols. Finally, bit extraction, truncation as well as arithmetic to Boolean rely on the invocation of joint sharing followed by evaluating the bit extraction or the PPA circuit. Both the circuit evaluations rely on invoking the multiplication protocol. Hence, similar to the previous protocols, simulation steps for bit extraction, truncation and arithmetic to Boolean can be obtained by following the steps for simulating joint sharing and multiplication, in the order in which they appear in the resultant protocol.

Similarly, it is easy to observe that the protocols for oblivious select, equality check, comparison, maxpool and ReLU build on top of the prior building blocks. Hence, their simulation follows from the simulation of the underlying protocols.

5.12.3 Security against a $(1, 1)$-mixed adversary

A closely related notion to FaF is that of mixed adversarial model [19, 51, 75, 82, 87, 108, 107], where a single (centralized) adversary is allowed to corrupt t parties maliciously and a disjoint subset of $h^\star$ parties semi-honestly. A protocol secure against such an adversary is said to be $(t, h^\star)$-mixed secure. It may seem that the mixed notion subsumes the FaF notion, but [5] shows otherwise. However, we show that our designed protocols are also secure in the $(1, 1)$ mixed adversarial model. The intuition for our protocols being secure in the mixed adversarial model as well is as follows. Observe that since the mixed model comprises a centralized adversary, as opposed to the decentralized one in the FaF model, the view of the semi-honest parties is available to the adversary while deciding the attack strategy for the malicious parties. The design of our protocols is such that it inherently is capable of withstanding such attacks due to the threshold of our secret-sharing scheme being set as $t + h^\star$, thus lending our protocols secure against the centralized $(1, 1)$-mixed adversary as well. We next provide the simulation proof for the same. Since the proofs follow easily from the simulation proofs for FaF security, in our case, we restrict to discussing the mixed-secure simulation for the sharing protocol.

The ideal functionality for the sharing protocol secure against a mixed adversary appears in Fig. 5.35.

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

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

– Receive $(\mathsf{Input}, \mathsf{v})$ from dealer $P_d \in \mathcal{P}$. Let $P^\star$ be the malicious party corrupted by $\mathcal{S}_{\mathsf{mixed}}$.

– Receive $\mathsf{continue}$ or $\mathtt{abort}$ with $(\mathsf{Select}, \mathsf{C})$ from $\mathcal{S}_{\mathsf{mixed}}$. Here, C denotes pair of parties that $\mathcal{S}_{\mathsf{mixed}}$ wants to choose as conflict pair.

– If received $\mathsf{continue}$, randomly pick $\alpha_{\mathsf{v}_{\mathsf{ij}}} \in \mathbb{Z}_{2^\ell}$, for $1 \leq i < j \leq 5$ and compute $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{1 \leq i < j \leq 5} \alpha_{\mathsf{v}_{\mathsf{ij}}}$. Set $\mathsf{msg}_s = (\beta_{\mathsf{v}}, \{\alpha_{\mathsf{v}_{\mathsf{ij}}}\}_{i \neq s, j \neq s})$, for each $P_s \in \mathcal{P}$.

– Else if received $\mathtt{abort}$, then:

– If $P^\star \in \mathsf{C}$, then set $\mathsf{CP} = \mathsf{C}$ and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

– Else set CP to include $P^\star$ and one other party from $\mathcal{P}$, and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

Figure 5.35: Mixed-secure ideal functionality for input sharing.

The simulator for the sharing protocol secure against a mixed adversary appears in Fig. 5.36.

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

Let P_l be the malicious party and P_m be the semi-honest party controlled by adversary $\mathcal{A}$.

Preprocessing

– $\mathcal{S}_{\mathsf{mixed}}$ emulates $\mathcal{F}_{\mathsf{Setup}}$ and gives the respective keys to $\mathcal{A}$. The shares of α_{v} that are held by $\mathcal{A}$ are sampled non-interactively using the shared keys. Other values ($\alpha_{\mathsf{v}_{\mathsf{ij}}}$ for $1 \leq i < j \leq 5$ and $\alpha_{\mathsf{v}_{\mathsf{ji}}}$ for $1 \leq j < i \leq 5$), not known to $\mathcal{A}$, are sampled randomly.

Online

– If P_l or P_m is the dealer, $\mathcal{S}_{\mathsf{mixed}}$ receives β_{v} from $\mathcal{A}$. Given the knowledge of all shares of α_{v}, $\mathcal{S}_{\mathsf{mixed}}$ obtains $\mathcal{A}$'s input as $\mathsf{v} = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}}$. Following this, $\mathcal{S}_{\mathsf{mixed}}$ emulates $\mathcal{F}_{\mathsf{Jmp}}$ with $\mathcal{A}$ as one of the senders, to deliver β_{v} to all parties. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathsf{mixed}}$ sets CP and invokes $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$ with $(\mathsf{Input}, \mathsf{v})$, and $\mathsf{continue}/\mathtt{abort}$ and $(\mathsf{Select}, \mathsf{CP})$.

– Else, $\mathcal{S}_{\mathsf{mixed}}$ honestly generates β_{v} by setting the input, v, of honest dealer as $\mathsf{v} = 0$. $\mathcal{S}_{\mathsf{mixed}}$ either sends β_{v} to $\mathcal{A}$ and/or emulates $\mathcal{F}_{\mathsf{Jmp}}$ to deliver β_{v} to all, with $\mathcal{A}$ either as the sender or receiver, depending on the identity of P_i. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathsf{mixed}}$ sets CP and invokes $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$ with $\mathsf{continue}$ or $\mathtt{abort}$, and $(\mathsf{Select}, \mathsf{CP})$.

Figure 5.36: Simulator corresponding to $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$.

Observe that the view generated by the simulator is indistinguishable from the real-world view, and the argument follows similar to as given in Lemma 5.1.

Chapter 6

Secure Allegation Escrow System

This chapter discusses the application of allegation escrow systems. The results in this chapter have led to a publication at The Web Conference 2023 [139].

6.1 Overview

Recall that an allegation escrow is a system that allows victims of crimes to file a confidential report about the crime, so that necessary actions can be taken to provide justice to the victim (§1.1.2.2). However, users hesitate to participate in these systems due to the fear of retribution. Thus, to increase trust in the system, cryptographic solutions are being designed to realize web-based secure allegation escrow (SAE) systems. We identify shortcomings in the prior allegation escrow systems. This also includes identifying attacks on the state-of-the-art system of [14], that can compromise a victim's privacy. To address these privacy breaches, we design a secure allegation escrow system called Shield, while retaining the salient features from prior works. The features provided by Shield, in comparison to the prior works, appear in Table 6.1. As evident from the table, [14] focuses on providing an $\mathcal{O}(1)$ complexity solution, which comes at the expense of privacy. On the contrary, we prioritize user *privacy* over system *efficiency* since privacy is essential for an SAE system. Hence, our protocol aims at achieving as efficient a solution as possible while guaranteeing *no* privacy breach. A new replacement to the secure matching protocol that identifies a revealable set of matching allegations and the inclusion of a new duplicity check protocol that prevents users from filing duplicate allegations lies at the heart of Shield. The challenge in designing the matching protocol lies in handling a user-defined and private reveal threshold. The subtlety in designing the duplicity check is in ensuring that a genuine alleger is allowed to complain against multiple perpetrators if required. This should

be done while simultaneously guaranteeing that a corrupt alleger cannot forge the identity of another honest alleger to file duplicates (unless these two allegers collude). For completeness, we additionally provide features such as allegation modification and deletion, which were absent in the state of the art. Note that these can optionally be included in the system, depending on the deployment scenario. We refer the reader to §1.1.2.2 for a quick overview and desirable properties of an SAE system.

Protocol	*Flexible* reveal threshold	*Private* reveal threshold	Duplicity complexity	Matching complexity
[198]	✘	✘	NA [a]	$\mathcal{O}(\mathsf{N})$ [b]
[144]	✘	✘	$\mathcal{O}(\mathsf{N})$	$\mathcal{O}(\mathsf{N} \cdot q)$
[14]	✔	✘	✘	$\mathcal{O}(1)$
Ours	✔	✔	$\mathcal{O}(\mathsf{N})$	$\mathcal{O}(\mathsf{N} \cdot \mathsf{maxths})$

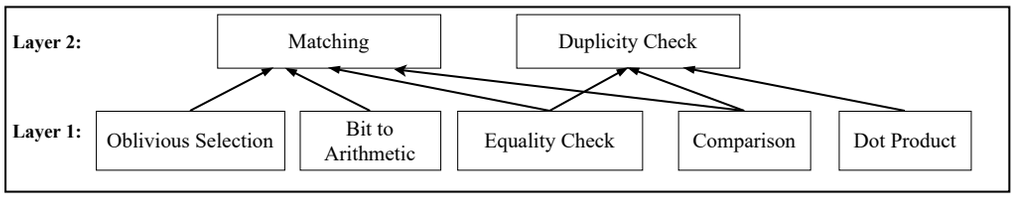
q: fixed reveal threshold, maxths: upper bound on flexible reveal threshold, N: number of allegations in the system.
[a]Duplicity check is not applicable here. Filing a duplicate allegation, to prematurely reveal a genuine one, requires a threshold of at least 2 as opposed to 1 in Callisto. [b]Due to missing details in Callisto, the complexity reported assumes requirement of a linear scan to identify matching allegations.

Table 6.1: Comparison of SAE protocols.

We resort to a modular approach to design the protocols, as shown in Fig. 6.1, by identifying the MPC building blocks that would be required and their interdependence. These building blocks have been extensively used in realizing privacy-preserving machine learning (PPML) [49, 193, 50, 38, 136, 173, 174]. Importantly, we allow Shield to make black-box use of the MPC building blocks. This not only allows Shield to inherit the latter's security guarantees and efficiency, but also opens up the possibility of utilizing the future advancements of MPC in a seamless way. We focus on benchmarking the complexity of allegation processing in our system and report the overhead involved in the enhancement. We realize Shield using the FaF-secure MPC designed in Chapter 5. Finally, we elaborately discuss the design choices that such a system should incorporate when attempting to achieve an ideal solution.

6.2 Related work

Callisto [198] provides a first-step solution to guarantee user privacy when filing allegations. However, it can be greatly improved to better cater to user privacy. Callisto does not provide



Primitives categorized into layers where higher layer primitives build over lower layer ones. These implicitly build over Layer 0 - input sharing, reconstruction, addition, multiplication - provided by underlying MPC.

Figure 6.1: Hierarchy of primitives.

a formal threat model and selectively accounts for misbehaviour from a malicious user. It also requires placing trust in the various entities involved. For instance, the DB server in Callisto, which stores perpetrator-id (pid) and the encrypted allegation text, is not modelled as a trusted entity, even when compromising the same allows an adversary to learn the number of allegations against a specific pid (*probing attack*). Additionally, during allegation modification, the DB server can learn whether pid is modified or if the allegation text was modified. Thus, the DB server cannot be modelled as an untrusted entity. Moreover, Callisto only has an invitation-based registration of users, and a policy-based solution to ensure explicit tracking of users (via mapping the real-world identity to the unique ID provided during registration) is not done. Further, unlike stated in Callisto, trivially extending support to the mentioned features results in privacy issues: (i) support for higher reveal thresholds leads to premature allegation revealing due to duplicates and (ii) support for allegation matching when users are given the flexibility to file using different identifying attributes (name, email-id, phone number, etc.) of a perpetrator leads to unidentified matching allegations due to mismatch in perpetrator attributes. We elaborate on the issues in each case next. Since Callisto has a reveal threshold of 1 (see §1.1.2.2 for a discussion on reveal threshold), it does not have to explicitly check for duplicate allegations being filed. However, the check for duplicity is indispensable for higher thresholds. Since Callisto does not hold its users accountable for the filed allegations, the current framework is not equipped to check for duplicates. Similarly, we see possible issues in matching allegations using multiple identifiers for a given perpetrator. Callisto suggests viewing the **pid** as a vector issued by the key server, with each component of the vector corresponding to an identifier of the perpetrator, as submitted by the victim. The DB server would then identify a match if the vectors have a common component. However, we would like to note that such a solution can, in fact, hinder the correctness of the system. The flexibility of having victims query on different identifiers for the same victim could result in two allegations being filed with non-overlapping identifiers. This would result in the DB server failing to match the allegations, despite both being against the same perpetrator.

In an attempt to overcome the issues present in Callisto, the work in [144] relies on the cryptographic technique of MPC. It provides a distributed escrow system referred to as WhoToo for reporting sexual misconduct. This distributed variant of the system ensures that the entities in the system are no longer required to be treated as a single point of trust. The authors in [144] identify the following issues with Callisto—(i) Callisto fails to bind the alleger identity to the allegation (i.e., alleger is not *accountable* for its allegation), which facilitates attacks that prematurely reveal allegations to the LOC, (ii) Callisto is susceptible to probing attack as stated earlier, and (iii) Callisto leaks the user's identity to the key server, each time the former authenticates itself to the latter, in the process of querying for a pid. The authors in [144] propose a solution that specifically addresses these three attacks.

The recent work in [14] not only addresses the issues pointed in [144] but also enhances WhoToo by identifying and addressing the limitations present therein. First, the authors in [14] make a strong argument in favour of empowering the users with the flexibility of having a user-defined reveal threshold instead of enforcing a globally predetermined reveal threshold, as in WhoToo. Second, they focus on providing a computationally more efficient and hence scalable solution. Unlike the solution in WhoToo that requires $\mathcal{O}(q\mathsf{N})$ computations to process a newly filed allegation, the solution presented in [14] requires $\mathcal{O}(1)$ computations. Here, q is the globally-fixed public threshold and N denotes the total number of allegations in the system. Additionally, the authors in [14] showcase how the system can be generalized to handle allegations against different types of crimes, rather than limit to sexual misconduct alone. Thus, [14] presents the state-of-the-art solution to realizing a secure allegation escrow system. Similar to WhoToo, [14] distributes the trust among multiple parties via MPC. However, in an attempt to achieve $\mathcal{O}(1)$ computational efficiency, it loses out on guaranteeing user privacy. Attacks that breach user privacy are given in §6.2.1.

Finally, to reduce the trust placed on escrows, the work of [101] uses secure enclaves (built on Intel SGX) as an additional defence mechanism. However, due to the backlash faced by SGX-based solutions, we do not delve into it [224, 148, 33].

6.2.1 Attacks on [14]

Before we discuss the attacks that can be launched on [14], we give a quick overview of their bucketing algorithm that is used to identify matching allegations.

Bucketing algorithm of [14] The bucketing algorithm is used to identify a set of revealable allegations, if any, in the system when a new allegation is filed. For this, the allegations present

in the system are grouped together as *collections* and are stored in a data structure that consists of numbered *buckets*. The invariant maintained by the data structure is such that an allegation a, waiting for j more matching allegations before it can be revealed, is present in bucket j. Thus, when a new allegation a with threshold t is filed, it is included in bucket t[1]. Additionally, the SAE protocol ensures that the escrows can compare and match allegations only within the same bucket. This is achieved by the escrows associating a unique private key sk_J with respect to each bucket j where the key is known to them in $[\![\cdot]\!]$-shared form. The escrows jointly compute the verifiable pseudorandom function (PRF) value for each allegation in bucket j (using a distributed protocol). Specifically, the escrows invoke a PRF with shares of the bucket key $[\![\mathsf{sk}_\mathsf{J}]\!]$ on the shares of the input $[\![\mathsf{H}(\mathsf{pid})]\!]$ ($\mathsf{H}(\cdot)$ is a collision-resistant hash function) for each allegation. The PRF output is revealed on clear to all escrows. This allows the escrows to locally determine the matching allegations within the bucket (two allegations match if they have the same PRF value implying that they have the same meta-data or perpetrator-id). Such matching allegations are grouped together as a collection. As a collection grows in size, it is copied onto lower buckets to ensure the above-mentioned invariant is maintained. During this process, if two different (non-intersecting) collections C_1, C_2 with matching meta-data happen to overlap (i.e., they span across a common bucket), then the collections are coalesced into a single collection C. The collection C consists of the union of all the allegations in C_1, C_2 and is said to span the union of the buckets spanned by C_1 and C_2. Thus, a collection may span across many (consecutive) buckets. It is interesting to note that all the allegations in a collection have matching meta-data, but all allegations with matching meta-data need not belong to the same collection (as they may belong to different collections spanning non-overlapping buckets)! In the life cycle of an allegation (i.e. from its filing to its revelation), it starts off as a singleton collection; a collection grows in size when more matching allegations are found within the same bucket; the collection is propagated to lower buckets when its size increases (i.e. to maintain the invariant); the collection is revealed when it reaches *bucket-0*, indicative of the fact that the allegations in the collection are waiting for no (0) more allegations before it can be revealed. The exact bucketing rules are given below.

Bucketing Rules: Apply the following rules repeatedly (in any order) till no further rules apply. Rules 2,3, and 4 only apply to collections that haven't been revealed.

1. When an allegation with threshold t is filed, it forms a singleton collection and is added to bucket t.
2. If $Min(A)$ is the smallest bucket occupied by a collection A and every allegation in A has

[1]Note that the interpretation of threshold in [14] is slightly different and hence we modify the SAE protocol description to keep the interpretation of t consistent with ours.

a threshold $< Min(A) + |A| - 1$, A is copied to bucket $Min(A) - 1$. Note that A still occupies the buckets it used to occupy. Copying adds the collection to a new bucket.

3. When two collections overlap and occupy the same bucket and their allegations are found to match, they coalesce into one collection.
4. When a collection reaches bucket-0, all of its allegations are revealed.
5. If a collection A is revealed, we make sure it occupies buckets $1, \ldots, |A|$, even as A grows. This enables future matching allegations to be revealed.

We next elaborate on the attacks that can be launched on this system.

Attack by filing fake allegations. To determine matching allegations, the protocol in [14] only compares allegations that are waiting for the *same* number of additional allegations required to satisfy their public reveal threshold. Whenever a comparison results in a match, such allegations are grouped (collections) and processed as a single unit from then on. Although the allegation remains hidden, each escrow learns the lifecycle of an allegation, which includes the time of allegation filing, whether a comparison results in a match, grouping of matched allegations, and the number of additional allegations that each filed allegation awaits. Consider now a scenario where a perpetrator colludes with an escrow and has access to the view of the escrow. Based on the information of when the perpetrator launched the assault and the timing of a filed allegation, it may be suspicious that this allegation is indeed against it. To confirm the suspicion, the perpetrator can file a fake allegation against itself by setting the same (publicly known) threshold as that of the suspected one. The colluding escrow can thus learn if the suspected allegation and the fake allegation are a match. [14] argues that such adversarial behaviour would result in leaving a non-repudiable paper trail (i.e. each alleger is held *accountable* for its allegation), and hence an adversary would not take such risks. However, the counterargument is that the adversary will be at risk only when its fake allegation is revealed, which may not always be the case. Given access to information such as the number of users victimized, the timing of the assault, etc., and the view of the escrow, an adversary (perpetrator) can take a well-educated risk and file such fake allegations with the confidence that it will not be revealed. For instance, let an allegation be filed with threshold $\mathtt{t} > 2$ after a perpetrator launched an assault. A suspicious perpetrator can launch the above attack with the confidence that it will not be revealed if it had harmed only a single victim, implying the absence of other matching allegations. This makes it disadvantageous for a user to set high thresholds, and thus, the whole purpose of having a system with a flexible reveal threshold is lost. To avoid such attacks, we design protocols that keep the reveal threshold private and leak no intermediate information to the escrows. We also bound the highest threshold that can be set to increase

the probability of an allegation being matched and revealed. This also tackles the issue of an adversary trying to overload the system by flooding it with fake allegations with a very high threshold that will remain unrevealed if the threshold is unbounded.

Attack by filing duplicate allegations. Recall that a duplicate allegation is one that is filed by the same alleger against the same perpetrator more than once. Since the system allows each user to file multiple allegations, a corrupt user can file duplicate allegations against a targeted perpetrator. Thus, duplicate allegations together with genuine allegations may form a revealable set, leading to the possibility of prematurely revealing genuine allegations against the same perpetrator (see example in the §1.1.2.2). Clearly, the privacy of an honest user is breached since its allegation may be revealed even when its threshold criteria are not met by other genuine allegations. One may argue that the above attack may be deterred because the system maintains a non-repudiable paper trail, and the filer of duplicates will eventually be penalized. The counterargument, however, is that despite the corrupt user being punished, the damage to a genuine victim is irrevocable. The presence of such an attack lowers the trust of genuine victims in the system and may discourage them from using it. Hence, we design a duplicity check protocol to prevent this.

6.3 Design of Shield

6.3.1 System model

We design Shield to comprise 5 escrows (e.g., computationally powerful hired servers) $\mathcal{P} = \{P_1, P_2, \ldots, P_5\}$ that are connected via pairwise private and authentic channels in a synchronous network. These escrows enact the role of parties in the underlying MPC protocol. Protocols are designed in the FaF model with a static, malicious probabilistic polynomial time (PPT) adversary that can corrupt up to one party and a *different* semi-honest adversary that can corrupt at most one other party. We assume that an arbitrary number of system users may collude with the maliciously corrupt escrows.

6.3.2 Shield functionality

In this section, we design an ideal functionality $\mathcal{F}_{\mathsf{Shield}}$ for our Shield system (Fig. 6.2) that follows on similar lines to [14]. For ease of readability, we describe the functionality for a *robust* system here. At a high-level, $\mathcal{F}_{\mathsf{Shield}}$ aims to achieve *alleger's anonymity and allegation secrecy*. That is,

an alleger's identity and its allegation should remain hidden from the escrows until the allegation is revealed as a part of a revealable collection. $\mathcal{F}_{\mathsf{Shield}}$ consists of six phases—(i) initialization, (ii) user registration, (iii) allegation filing, (iv) duplicity check, (v) allegation matching and (vi) allegation revealing. Throughout the phases, it maintains a few data structures—R: the registered users of the system; A: all the allegations filed in the system, $\mathcal{S}$: a collection of revealable allegations; and P: details of the revealed perpetrators. The i^{th} entry in A is denoted as $\mathsf{a_i}$, which mainly has three attributes pid (perpetrator's id), t (reveal threshold) and Text (crime details). Similarly, the i^{th} entry of P is denoted as p_i, and it has two attributes pid (perpetrator's id) and ac (count of allegations revealed against this perpetrator).

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

$\mathcal{F}_{\mathsf{Shield}}$ interacts with escrows ($\mathcal{P}$), user, ideal world malicious adversary $\mathcal{S}_{\mathcal{A}}$ and ideal world semi-honest adversary $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ and works as follows:

- **Initialization** Initialize empty lists R, A, $\mathcal{S}$ and P.
- **Registration** On receiving a message ("Register", c, ID) from a user with identifier ID, send the message ("Registered", c, ID) to all escrows if the certificate c verifies. Include ID in list R and set $\mathrm{ID.count} = \mathsf{maxalg}$.
- **Allegation Filing** On receiving a message ("Allege", ID, $\mathsf{a_{new}}$) from a user ID and allegation $\mathsf{a_{new}}$, send the message ("Failed attempt") to escrows if $\mathrm{ID} \notin \mathsf{R}$ or if $\mathrm{ID.count} = 0$. Else send message ("Allege"), reduce ID.count by 1, and enter the next phase.
- **Duplicity Check** Check if ($\mathsf{a_{new}.pid}$, ID) is part of some $\mathsf{a_i}$ in A. If found, *ignore* and send the message ("Duplicate") to escrows. Else, include ($\mathsf{a_{new}}$,ID) in A, send the message ("File allegation") to all escrows, and enter the matching phase.
- **Matching** If $\mathsf{a_{new}.pid} = \mathsf{p}_i\mathsf{.pid}$, then set $\mathsf{a_{new}.t} = \mathsf{a_{new}.t} - \mathsf{p}_i\mathsf{.ac}$. Determine if there exists a revealable subset $\mathcal{S}$ in A. If found, do as follows and continue to the next phase
  - Delete each $\mathsf{a_i} \in \mathcal{S}$ from A and send ("Found", I) to escrows where I denotes indices of allegations in $\mathcal{S}$ with respect to A.
  - If $\mathsf{a_i.pid} = \mathsf{a_{new}.pid}$ and $\mathsf{a_i} \notin \mathcal{S}$, set $\mathsf{a_i.t} = \mathsf{a_i.t} - |\mathcal{S}|$.
  - If $\mathsf{p}_i\mathsf{.pid} = \mathsf{a_{new}.pid}$, set $\mathsf{p}_i\mathsf{.ac} = \mathsf{p}_i\mathsf{.ac} + |\mathcal{S}|$. Else include a new entry $(\mathsf{a_{new}.pid}, |\mathcal{S}|)$ in P and send message ("New P entry") to the escrows.
- **Allegation Revealing** Reveal allegations $\mathcal{S}$ to the escrows. Reset list $\mathcal{S}$ to be empty.

$\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 6.2: Ideal functionality for Shield.

In the initialization phase, $\mathcal{F}_{\mathsf{Shield}}$ initializes these data structures to empty lists. During

the registration phase, a user sends a request to get registered, and the functionality adds the entry to R and notifies the escrows. Here, $\mathcal{F}_{\mathsf{Shield}}$ must ensure that every allegation should be associated with an authentic, real-world identity, which ensures *accountability*. To tie up with a real-world identity, a user must submit a certificate c, obtained earlier from a certification authority (CA), along with its request, which $\mathcal{F}_{\mathsf{Shield}}$ verifies before registering the user. This helps in tracing back an allegation to its alleger in the case of misbehaviour, which discourages *fake* allegations. Additionally, it can be used to discourage *duplicate* allegations. For every registered user, $\mathcal{F}_{\mathsf{Shield}}$ sets the maximum number of allowed allegations to maxalg. This count is decreased every time the user files an allegation in the allegation filing phase.

Next, during the allegation filing phase, a user submits a new allegation $\mathsf{a}_{\mathsf{new}}$. If the user is not registered or the user's quota of the maximum allowed allegation count is 0, $\mathcal{F}_{\mathsf{Shield}}$ ignores this allegation. Else, it decreases the corresponding user's allowed allegation count by 1. It notifies the escrows of a failed/successful incoming allegation (but nothing beyond) and moves on to the next phase, where duplicity of $\mathsf{a}_{\mathsf{new}}$ is checked with respect to the allegations in A. If found to be a non-duplicate, $\mathsf{a}_{\mathsf{new}}$ is added in A (and accordingly, a message is sent to the escrows indicating duplicate/non-duplicate). Next, the matching phase is started. Here, $\mathcal{F}_{\mathsf{Shield}}$ first checks if pid of $\mathsf{a}_{\mathsf{new}}$ matches with any entry p_i of P. If this is true, then $\mathsf{a}_{\mathsf{new}}$'s reveal threshold is reduced by the number of allegations against p_i. This ensures that the alleger of $\mathsf{a}_{\mathsf{new}}$ only has to find this reduced number of supporters from the unrevealed ones in A. Next, $\mathcal{F}_{\mathsf{Shield}}$ finds if there is a revealable subset $\mathcal{S}$ in A. If so, it performs a series of adjustments in the maintained lists before publishing $\mathcal{S}$ in the next phase—(a) the allegations in $\mathcal{S}$ to be revealed are erased from A, (b) it reduces the threshold of each matched, yet not to be revealed, allegation a_i (pid of a_i equals to that of $\mathsf{a}_{\mathsf{new}}$) by $|\mathcal{S}|$, (c) it increases the allegation count of the perpetrator of $\mathsf{a}_{\mathsf{new}}$ in list P or includes an entry for $\mathsf{a}_{\mathsf{new}}$'s pid in P if it was not present earlier in the list. In the reveal phase, $\mathcal{F}_{\mathsf{Shield}}$ reveals $\mathcal{S}$.

6.3.3 Shield overview

6.3.3.1 Primitives used to realize Shield

Here, we elaborate on the primitives relied upon by Shield.

Verifiable pseudorandom function (VRF) [73] Informally, a VRF is a pseudorandom function $F_{\mathsf{sk_v}}(\cdot)$ along with a proof generation function $\pi_{\mathsf{sk_v}}(\cdot)$ such that a PPT adversary cannot distinguish $F_{\mathsf{sk_v}}(x)$ from an output of a random function without the access to the VRF secret key $\mathsf{sk_v}$ or $\pi_{\mathsf{sk_v}}(x)$. The correct computation of $F_{\mathsf{sk_v}}(x)$ can be verified given a matching public

key $\mathsf{pk_v}$ and proof $\pi_{\mathsf{sk_v}}(x)$. Let Π_{vrf} be a protocol that inputs $[\![\mathsf{sk_v}]\!]$ and $[\![\mathsf{x}]\!]$, and outputs $[\![F_{\mathsf{sk_v}}(x)]\!]$ and $[\![\pi_{\mathsf{sk_v}}(x)]\!]$.

Message authentication code (MAC) Informally, a MAC is a function which takes as input a secret key $\mathsf{sk_m}$ and a message x. The output, denoted by mac, can be used to authenticate x and confirm its origin from the entity holding $\mathsf{sk_m}$. Let Π_{mac} be the protocol that inputs $[\![\mathsf{sk_m}]\!]$, $[\![\mathsf{x}]\!]$, and outputs $[\![\mathsf{mac}]\!]$.

Following [14], we instantiate Π_{vrf}, Π_{mac} using the PRF construction of [73]. For the VRF, we restrict the output to only $\pi_{\mathsf{sk_v}}(x)$ since $F_{\mathsf{sk_v}}(x)$ can be generated using $\pi_{\mathsf{sk_v}}(x)$ and public key $\mathsf{pk_v}$, in our instantiation.

MPC building blocks The designed protocols rely on MPC building blocks described in Fig. 6.1. Their description and semantics of the inputs and outputs are provided in Table 6.2.

Building block	Notation	Description
Comparison	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} < \mathsf{y}$, else outputs $\mathsf{b} = 0$
Equality	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} = \mathsf{y}$, else outputs $\mathsf{b} = 0$
Oblivious Select	$[\![\mathsf{x_b}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{x}_0]\!], [\![\mathsf{x}_1]\!], [\![\mathsf{b}]\!]^{\mathbf{B}})$	Obliviously selects $\mathsf{x_b}$ among $\mathsf{x}_0, \mathsf{x}_1$
Bit2A	$[\![\mathsf{b}]\!] = \Pi_{\mathsf{Bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	Converts bit to its arithmetic equivalent
Dot product	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{DotP}}([\![\boldsymbol{x}]\!]^{\mathbf{B}}, [\![\boldsymbol{y}]\!]^{\mathbf{B}})$	Computes $\mathsf{z} = \bigoplus_{i=1}^{\mathsf{n}} x_i \wedge y_i$ †

† x_i: i^{th} element of vector $\boldsymbol{x}$; n: vector size; $\bigoplus$: XOR; $\wedge$: AND.

Table 6.2: Description of building blocks

6.3.3.2 The Shield system

Shield is designed to realize the $\mathcal{F}_{\mathsf{Shield}}$ ideal functionality described above. The designed system continues to have six phases, and the details of realizing each of these phases are described next. These follow similar to [14] and set the stage for our new duplicity check and matching protocol described next.

Escrow initialization This allows the escrows to establish the necessary setup required for the underlying MPC and for securely processing a filed allegation. Escrows establish authenticated communication links between themselves. They generate $[\![\cdot]\!]$-shares of the secret keys required for a VRF ($[\![\mathsf{sk_v}]\!]$) and a MAC ($[\![\mathsf{sk_m}]\!]$) primitive. The public key ($\mathsf{pk_v}$) of the VRF is,

however, known on clear to all the escrows. The use of these primitives during the later phases (registration, filing, and duplicity check) is discussed in place.

User registration Here, escrows perform initialization with respect to users in preparation for them to securely participate in the system. A high-level overview is discussed next.

(i) Escrows must be able to verify and register only valid users of the system. For this, a user contacts the CA, who verifies its real-world identity and supplies a certificate, c. Escrows register the user in the system after verifying c.

(ii) To empower only registered users to file an allegation, the system relies on digital signature schemes. To file an allegation later (with accountability), a user U needs to authenticate its allegation using a digital signature (DS) secret key sk^{U}, and the escrows verify an allegation's authenticity using the corresponding public verification key pk^{U}. Additionally, the escrows must be able to validate that the used pk^{U} belongs to a registered user. Hence, it is required that each registered user records its verification key with the escrows during registration itself. However, the knowledge of the verification key on clear allows the escrows to link an allegation to a user. To validate a user's allegation without linking to its identity, the escrows issue to the user a VRF proof on its verification key, without learning the latter. This is enabled by having the user secret-share its key, and the escrows (jointly) compute $[\![\pi_{\mathsf{sk_v}}]\!] = \Pi_{\mathsf{vrf}}\left([\![\mathsf{sk_v}]\!], [\![\mathsf{pk}^{\mathsf{U}}]\!]\right)$ and reconstruct $\pi_{\mathsf{sk_v}}$ towards the user. During registration, since the escrows hold pk^{U} and $\pi_{\mathsf{sk_v}}$ in shared format, they cannot associate these to the user, even though they know the user identity via CA certificate in the current phase. Hence, when a user presents pk^{U} and $\pi_{\mathsf{sk_v}}$ on clear during allegation filing, user identity remains hidden while the escrows can validate the user. This mechanism also prevents an outsider from faking its registration as it cannot generate valid proof even while colluding with the corrupt escrows since no subset of them knows $\mathsf{sk_v}$. Next, we must allow a user to file multiple allegations[2], if required, without the escrows learning that the two allegations have originated from the same user. To tackle this, user U generates maxalg pairs of keys $(\mathsf{pk}^{\mathsf{U}}_i, \mathsf{sk}^{\mathsf{U}}_i)$ corresponding to a DS scheme, where a unique pair is to be consumed for each allegation, and $[\![\cdot]\!]$-shares these towards the escrows. The escrows compute VRF proof $[\![\pi^i_{\mathsf{sk_v}}]\!] = \Pi_{\mathsf{vrf}}\left([\![\mathsf{sk_v}]\!], [\![\mathsf{pk}^{\mathsf{U}}_i]\!]\right)$ for $i \in \{1, \ldots, \mathsf{maxalg}\}$, which is reconstructed towards the user. Here, maxalg denotes the maximum number of allegations that a user is allowed to file.

(iii) The escrows must be able to identify if the same user is filing a *duplicate* allegation against the same perpetrator ID, pid. For this, the user and the escrows rely on a MAC to

[2]Multiple allegations should be against different perpetrators. If a user allegation is yet to be revealed, it must be disallowed to allege the same perpetrator (duplicate). However, a revealed alleger victimized by the same perpetrator at a later time must be allowed to file the allegation (non-duplicate).

generate a unique unforgeable user identity, uid to be given in shares along with each allegation. This is achieved by the user $[\![\cdot]\!]$-sharing a random $\mathsf{r} \in \mathbb{G}$ among the escrows. The escrows compute $[\![\mathsf{uid}]\!] = \Pi_{\mathsf{mac}}([\![\mathsf{sk_m}]\!], [\![\mathsf{r}]\!])$, and reconstruct it towards the user. Since the escrows hold $\mathsf{sk_m}, \mathsf{r}$ in shared format, they learn nothing. Further, a user is unable to generate a valid uid for itself or forge another user's uid due to (i) r being unique and secret to each user, and (ii) lack of knowledge about $\mathsf{sk_m}$. Details of how uid facilitates the detection of duplicates appear in §6.3.4. We note that this step is a new addition and is required to detect duplicates.

(iv) The escrows must also be able to learn the user's identity when its submitted allegation needs to be revealed. For this, the escrows rely on the MAC. The escrows compute $[\![\mathsf{mac}_i^{\mathsf{U}}]\!] = \Pi_{\mathsf{mac}}([\![\mathsf{sk_m}]\!], [\![\mathsf{pk}_i^{\mathsf{U}}]\!])$ and reconstruct it. This allows the escrows to store the association between a user and mac(s) generated on all its DS verification keys in a local map. Looking ahead, during the allegation revealing, the escrows recompute the MAC on the DS verification key used in an allegation. The recomputed MAC is matched against entries in the local map to trace the user of the allegation. Observe the duality in the need for anonymization in (ii) and traceability in (iv).

Allegation filing User U connects to all the escrows via an anonymous communication channel. It files i^{th} new allegation by submitting $\mathsf{pk}_i^{\mathsf{U}}, \pi_{\mathsf{sk_v}}(\mathsf{pk}_i^{\mathsf{U}})$ on clear, and shares of allegation denoted $[\![\mathsf{a_{new}}]\!] = ([\![\mathsf{uid}]\!], [\![\mathsf{r}]\!], [\![\mathsf{pid}]\!], [\![\mathsf{t}]\!], [\![\mathsf{Text}]\!])$, all signed using $\mathsf{sk}_i^{\mathsf{U}}$. That is, j^{th} escrow receives $\mathsf{pk}_i^{\mathsf{U}}, \pi_{\mathsf{sk_v}}(\mathsf{pk}_i^{\mathsf{U}})$ and signed jth share of $\mathsf{a_{new}}$. Escrows check that the submitted $\mathsf{pk}_i^{\mathsf{U}}$ was not used previously, followed by verifying the proof $\pi_{\mathsf{sk_v}}(\mathsf{pk}_i^{\mathsf{U}})$. Upon success, they proceed to verify the signature on $\mathsf{a_{new}}$ components using $\mathsf{pk}_i^{\mathsf{U}}$. If any verification fails, escrows ignore $\mathsf{a_{new}}$. Else, escrows proceed to the next phase.

Duplicity check The duplicity check protocol (§6.3.4) is run by the escrows to discard a new allegation if it is a duplicate.

Allegation matching This involves running our new matching protocol (§6.3.5) to identify a revealable set $\mathcal{S}$ of matching allegations.

Allegation revealing If a revealable set $\mathcal{S}$ is found during matching, this phase reveals $\mathcal{S}$ (together with the identity of the allegers) to the escrows[3]. Since the pk^{U} (submitted during

[3]Note that Shield supports an alternative model where the shared $\mathcal{S}$ is directly reconstructed to an external authority (designated to deliver justice) and thus hiding this crucial information from escrows, which may be simply hired for compute-service.

allegation filing) associated with allegations in $\mathcal{S}$ is known to the escrows, they recompute $[\![\mathsf{mac}^{\mathsf{U}}]\!] = \Pi_{\mathsf{mac}}([\![\mathsf{sk}_{\mathsf{m}}]\!], [\![\mathsf{pk}^{\mathsf{U}}]\!])$, and reconstruct it. Using the local map of valid mac(s) generated during registration, they can de-anonymize the user.

A schematic representation of Shield appears in Fig. 6.3. For ease of reading the upcoming sections, Table 6.3 enlists commonly-used notations in this chapter.

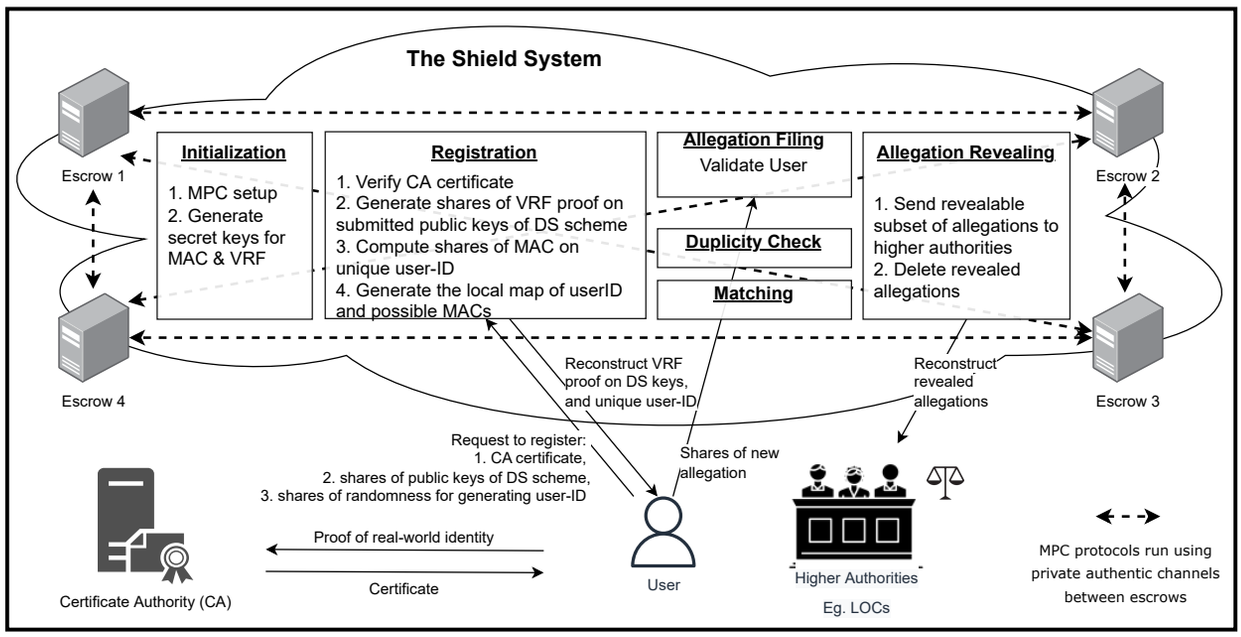


Figure 6.3: Schematic Diagram of Shield.

Notation		Description
N		Number of allegations in the system
maxalg		Maximum number of allegations that can be filed by a user
maxths		Upper bound on the reveal threshold
a		Allegation with attributes (uid, r, pid, t, Text)*
$\mathsf{a.x}$		Refers to attribute x of a
A		List of non-duplicate allegations filed in the system: $\{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$
P $\{\mathsf{p}_1, \mathsf{p}_2, ..\}$	=	List of revealed perpetrators, $\mathsf{p}_i = (\mathsf{pid}, \mathsf{ac})^\dagger$
$\mathcal{S}$		Set of revealable allegations
sk_{m}		Secret key for MAC
$\mathsf{pk}_{\mathsf{v}}, \mathsf{sk}_{\mathsf{v}}$		Public key and secret key for VRF
$(\mathsf{pk}_i^{\mathsf{U}}, \mathsf{sk}_i^{\mathsf{U}})$		i^{th} public key and secret key of user U for digital signature

* uid, r: unique id, pid: perpetrator id; t: reveal threshold, Text: crime description

† pid: perpetrator id; ac: count of allegations revealed against the perpetrator

Table 6.3: Notations used in this chapter.

6.3.4 Duplicity check protocol

To prevent the filing of duplicate allegations, two measures are taken. First, every user U is associated with a unique unforgeable user identity (uid) during the registration, which is verified when an allegation is filed. Second, the unique identity and the perpetrator identity of the new allegation is matched with that of the existing allegations. While the latter is the obvious test for duplicity, the former test ensures that a user cannot submit an allegation without registering and impersonating another user due to the unforgeability of the unique identity. Note that a user U cannot impersonate user U' unless it obtains $\mathsf{U}'.\mathsf{uid}$ by colluding with U'. Due to accountability property, such collusions are deterred. The protocol overview is given next.

To check if a valid uid has been submitted, escrows recompute $[\![\mathsf{uid}']\!] = \Pi_{\mathsf{mac}}([\![\mathsf{sk}_\mathsf{m}]\!], [\![\mathsf{a}_{\mathsf{new}}.\mathsf{r}]\!])$ using the secret-shared r submitted as part of the new allegation, $\mathsf{a}_{\mathsf{new}}$, and check equality of uid' and $\mathsf{a}_{\mathsf{new}}.\mathsf{uid}$. If the submitted uid is valid, escrows proceed to verify if there exists an allegation in the system $\mathsf{a}_\mathsf{i} \in \mathsf{A}$ such that $\mathsf{a}_\mathsf{i}.\mathsf{uid} = \mathsf{a}_{\mathsf{new}}.\mathsf{uid}$ and $\mathsf{a}_\mathsf{i}.\mathsf{pid} = \mathsf{a}_{\mathsf{new}}.\mathsf{pid}$. To determine this, for each allegation in the system, escrows invoke Π_{Eq} (Table 6.2) protocol to check for equality of pids and equality of uids. To determine if a_i is a duplicate, escrows check if both equalities hold. Formal details appear in Fig. 6.4.

Protocol $\Pi_{\mathsf{Dup}}(\{\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{N}\}, \mathsf{a}_{\mathsf{new}}, \mathsf{sk}_\mathsf{m})$

- $[\![\mathsf{uid}']\!] = \Pi_{\mathsf{mac}}([\![\mathsf{sk}_\mathsf{m}]\!], [\![\mathsf{a}_{\mathsf{new}}.\mathsf{r}]\!])$ and $[\![\mathsf{chk}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{uid}']\!], [\![\mathsf{a}_{\mathsf{new}}.\mathsf{uid}]\!])$
- Reconstruct chk. If $\mathsf{chk} = 1$, then for each allegation $\mathsf{a}_\mathsf{i} \in \mathsf{A}$,
  - $[\![\mathsf{chk}_1^i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_\mathsf{i}.\mathsf{pid}]\!], [\![\mathsf{a}_{\mathsf{new}}.\mathsf{pid}]\!])$
  - $[\![\mathsf{chk}_2^i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_\mathsf{i}.\mathsf{uid}]\!], [\![\mathsf{a}_{\mathsf{new}}.\mathsf{uid}]\!])$
- Reconstruct $\mathsf{chk}_3 = \Pi_{\mathsf{DotP}}(\{[\![\mathsf{chk}_1^i]\!]^{\mathbf{B}}, [\![\mathsf{chk}_2^i]\!]^{\mathbf{B}}\}_{i=1}^{\mathsf{N}})$.
- If $\mathsf{chk}_3 = 1$, ignore the processing of $\mathsf{a}_{\mathsf{new}}$ and discard it.

Figure 6.4: Duplicity check.

Finally, observe that escrows essentially perform a linear scan over the list of all allegations in the system. Hence duplicity check has the complexity of $\mathcal{O}(\mathsf{N})$.

6.3.5 Matching protocol

Let $\mathsf{a_{new}}$ be a newly filed allegation. The objective of the matching protocol is to determine the *largest* set $\mathcal{S}$, if any, of matching allegations against $\mathsf{a_{new}.pid}$, whose reveal criteria are met, i.e., reveal threshold of each allegation in $\mathcal{S}$ must be less than $|\mathcal{S}|$.

A cleartext algorithm for identifying $\mathcal{S}$ is as follows: (a) sort the matching allegations in A (associated with $\mathsf{a_{new}.pid}$) in decreasing order of threshold; let $\mathsf{A'} = \{\mathsf{a_1}', \mathsf{a_2}', \ldots, \mathsf{a_h}'\}$ be the sorted list of matched allegations, (b) let $\mathcal{S}_i = \{\mathsf{a_i}', \mathsf{a_{i+1}}', \ldots, \mathsf{a_h}'\}$, and check if $\mathsf{a_i}'.\mathsf{t} < |\mathcal{S}_i|$, starting from $i = 1$ to h, (c) the lowest value of i (i.e., largest $\mathcal{S}_i$) for which (b) is satisfied determines the largest revealable subset $\mathcal{S}_i$.

To preserve privacy, it is *not* enough to do the above computation on secret-shared data (that include $\mathsf{a_{new}}$ and A). The sequence of operations carried out during the computation must also not leak private information. In particular, in the cleartext algorithm described above, the length of the list $|\mathsf{A'}| = \mathsf{h}$ leaks information on the number of matching allegations against $\mathsf{a_{new}.pid}$ in the system. Further, during the construction of $\mathcal{S}_i$, the inclusion or exclusion of an allegation in $\mathcal{S}_i$ reveals information about the relative ordering of the allegations with respect to the threshold. Thus, the sequence of operations (selection, comparison, equality, etc.) on secret-shared data also needs to be carried out *obliviously*. A protocol is data-oblivious if the sequence of operations and memory accesses made during the protocol run are independent of the input. Thus, our objective is to design a data-oblivious matching protocol that operates on secret shared data. As an example, we note that this would mean we must operate on A rather than identifying $\mathsf{A'}$ during matching. Our matching protocol thus consists of the following four steps.

Step 1: Updating $\mathsf{a_{new}}$'s threshold based on perpetrators revealed so far When a new allegation, $\mathsf{a_{new}}$, is filed in the system, its threshold should be reduced by the number of allegations that have already been revealed against $\mathsf{a_{new}.pid}$. This ensures that the alleger of $\mathsf{a_{new}}$ only has to find this reduced number of supporters from the unrevealed ones in A. The first step checks if there exists an entry for perpetrator $\mathsf{a_{new}.pid}$ in P. If true, then the threshold $\mathsf{a_{new}.t}$ is reduced by $\mathsf{p}_i.\mathsf{ac}$, where $\mathsf{a_{new}.pid} = \mathsf{p}_i.\mathsf{pid}$. Note that these steps should be performed obliviously. For this, we first compute a secret-shared bit via Π_{Eq} that determines if $\mathsf{a_{new}.pid} = \mathsf{p}_i.\mathsf{pid}$, for each $\mathsf{p}_i \in \mathsf{P}$. Then, the oblivious select protocol Π_{Sel} (Table 6.2) is invoked to obliviously determine if the threshold remains unchanged ($\mathsf{a_{new}.t}$) or is updated ($\mathsf{a_{new}.t} - \mathsf{p}_i.\mathsf{ac}$).

Step 2: Searching for largest $\mathcal{S}$ Determining the *largest* set of revealable matching allegations translates to finding the highest threshold $\mathsf{t_{max}}$ (among the filed allegations) such

that the number of matching allegations $\mathsf{a_j}$ with $\mathsf{a_j.t} \leq \mathsf{t_{max}}$ is greater than $\mathsf{t_{max}}$. Let maxths denote the maximum reveal threshold allowed in Shield. To determine such a $\mathsf{t_{max}}$, we examine each possible threshold from maxths down to 1 iteratively. The order ensures we stop with the largest set $\mathcal{S}$. Starting with $i = \mathsf{maxths}$ iteration, we scan through the list of all allegations to determine the count ($= |\mathcal{S}|$) of matching allegations having threshold less than $i+1$ ($=\mathsf{t_{max}}+1$). The i^{th} loop terminates if and when the determined count value is greater than i and a flag is set. In each iteration, to avoid repeated comparison between allegations $\mathsf{a_i}$ and $\mathsf{a_{new}}$ to determine if there is a match, we store this information, generated during the first scan, in a Boolean array M of length $\mathsf{N}+1$ (including $\mathsf{a_{new}}$). The array entries are secret shared bits where the i^{th} entry, $[\![\mathsf{M}_i]\!]^{\mathbf{B}}$ is a 1 if $\mathsf{a_i.pid} = \mathsf{a_{new}.pid}$, and 0 otherwise. To prevent leaking information about whether an allegation satisfies the above threshold conditions or not, steps of updating count are performed obliviously via Π_{Sel}.

Step 3: Update A This step is executed only when flag is set in some iteration i in step 2, indicating the existence of $\mathcal{S}$, with count number of matching allegations and with the highest threshold equal to i. The goal here is to identify the allegations in this set, delete these from A, and update the threshold of non-revealable yet matching allegations to reflect the newly revealed number of allegations—all of these without leaking any additional information. To determine if an allegation belongs to $\mathcal{S}$, we scan through the list of all allegations $\mathsf{a_j} \in \mathsf{A}$, and identify if a matching $\mathsf{a_j}$ is revealable, i.e., $\mathsf{a_j.t} \leq i < \mathsf{count}$. This is done by reducing the thresholds of all matching allegations in A by count, followed by checking if the updated threshold is < 0 (which will be the case if it is a revealable allegation). This operation implicitly updates the threshold of non-revealable yet matching allegations. The operation of whether $\mathsf{a_j}$ is a match and, consequently, whether the threshold is updated or not, is made oblivious, as in step 1. That is, we compute a secret-shared bit that implies if $\mathsf{a_j.pid} = \mathsf{a_{new}.pid}$, and use it to update the threshold ($\mathsf{a_j.t} - \mathsf{count}$ or $\mathsf{a_j.t}$) via Π_{Sel}. Next, delete $\mathcal{S}$ from A.

Step 4: Updating P when $\mathcal{S}$ is found If $\mathsf{a_{new}.pid}$ does not exist in P, then a new entry in P is added with its allegation count $\mathsf{ac} = \mathsf{count}$. Else, the corresponding entry, say, p_i is updated to reflect the number of newly revealed allegations, i.e., $\mathsf{p}_i.\mathsf{ac} = \mathsf{p}_i.\mathsf{ac} + \mathsf{count}$. This step is performed via Π_{Sel}, as described earlier.

The formal matching protocol appears in Fig. 6.5. The matching protocol entails performing multiple scans over the list of all allegations. The number of times the scan is performed may vary. However, note that the protocol terminates after at most maxths iterations, where maxths is the upper bound on the reveal threshold. Hence, matching has a complexity of $\mathcal{O}(\mathsf{N} \cdot \mathsf{maxths})$.

Protocol $\Pi_{\mathsf{Mat}}(\{\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{N}\}, \mathsf{a}_{\mathsf{new}}, \mathsf{P})$

STEP 1:

- **for** *each* p_i *in* P **do**:
  - $[\![\mathsf{chk}_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_{\mathsf{new}}.\mathsf{pid}]\!], [\![\mathsf{p}_i.\mathsf{pid}]\!])$
  - $[\![\mathsf{a}_{\mathsf{new}}.\mathsf{t}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{a}_{\mathsf{new}}.\mathsf{t}]\!], [\![\mathsf{a}_{\mathsf{new}}.\mathsf{t} - \mathsf{p}_i.\mathsf{ac}]\!], [\![\mathsf{chk}_1]\!]^{\mathbf{B}})$
- Add $\mathsf{a}_{\mathsf{new}}$ as the $\mathsf{N}+1^{\text{th}}$ allegation in the system

STEP 2:

- **for** $i = 1$ *to* N **do**: $[\![\mathsf{M}_i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_{\mathsf{new}}.\mathsf{pid}]\!], [\![\mathsf{a}_\mathsf{i}.\mathsf{pid}]\!])$
- Set $[\![\mathsf{M}_{\mathsf{N}+1}]\!]^{\mathbf{B}} = [\![1]\!]^{\mathbf{B}}$
- **for** *each possible threshold* $i = \mathsf{maxths}$ *to* 1 **do**:
  - $[\![\mathsf{count}]\!] = [\![0]\!]$
  - **for** *each allegation* $\mathsf{a_j}$ *for* $j = 1$ *to* $\mathsf{N}+1$ **do**
    - $[\![\mathsf{chk}_2]\!]^{\mathbf{B}} = \Pi_{\mathsf{Comp}}([\![\mathsf{a_j}.\mathsf{t}]\!], i+1)$
    - $[\![\mathsf{count}]\!] = [\![\mathsf{count}]\!] + \Pi_{\mathsf{Bit2A}}\left([\![\mathsf{M}_j]\!]^{\mathbf{B}} \cdot [\![\mathsf{chk}_2]\!]^{\mathbf{B}}\right)$
  - Reconstruct $\mathsf{flag} = \Pi_{\mathsf{Comp}}([\![i]\!], [\![\mathsf{count}]\!])$. If flag is set to 1, **break** from the loop

STEP 3:

- If the flag is set to 0, Terminate the protocol. Else, **for** *each allegation* $\mathsf{a_j}$ *for* $j = 1$ *to* $\mathsf{N}+1$ **do**:
  - $[\![\mathsf{a_j}.\mathsf{t}]\!] = \Pi_{\mathsf{Sel}}\left([\![\mathsf{a_j}.\mathsf{t}]\!], [\![\mathsf{a_j}.\mathsf{t} - \mathsf{count}]\!], [\![\mathsf{M}_j]\!]^{\mathbf{B}}\right)$
  - Reconstruct $\mathsf{chk}_3 = \Pi_{\mathsf{Comp}}([\![\mathsf{a_j}.\mathsf{t}]\!], [\![0]\!])$
  - If $\mathsf{chk}_3 = 1$, include $\mathsf{a_j}$ in $\mathcal{S}$ and delete from A Else, **continue**.

STEP 4:

- Initialize $[\![\mathsf{flag}]\!]^{\mathbf{B}} = 0$
- **for** *each record* p_i *in* P **do**:
  - $[\![\mathsf{chk}_1]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_{\mathsf{new}}.\mathsf{pid}]\!], [\![\mathsf{p}_i.\mathsf{pid}]\!])$
  - $[\![\mathsf{flag}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Sel}}\left([\![\mathsf{flag}]\!]^{\mathbf{B}}, [\![1]\!]^{\mathbf{B}}, [\![\mathsf{chk}_1]\!]^{\mathbf{B}}\right)$
  - $[\![\mathsf{p}_i.\mathsf{ac}]\!] = \Pi_{\mathsf{Sel}}\left([\![\mathsf{p}_i.\mathsf{ac}]\!], [\![\mathsf{p}_i.\mathsf{ac}]\!] + [\![\mathsf{count}]\!], [\![\mathsf{chk}_1]\!]^{\mathbf{B}}\right)$
- Reconstruct flag. If not set, then create a new record p_{i+1} in list defined as $[\![\mathsf{p}_{i+1}.\mathsf{pid}]\!] = [\![\mathsf{a}_{\mathsf{new}}.\mathsf{pid}]\!]$ and $[\![\mathsf{p}_{i+1}.\mathsf{ac}]\!] = [\![\mathsf{count}]\!]$.

Figure 6.5: Allegation matching.

6.4 Additional features

Our system can easily be extended to support the modification and deletion of filed allegations. The protocols for the same follow on similar lines as that of the duplicity check protocol and entail performing a linear scan of the allegations in the system. The details are described next.

6.4.1 Allegation modification

Unlike in Callisto, where any aspect of the allegation can be modified, we only allow U to modify the reveal threshold t and/or allegation text Text. Allowing modifications to other components such as the user-ID uid, randomness r and perpetrator-ID pid of an allegation a is absurd and hampers the functionality of the system. This is because modifying uid, r is equivalent to a user claiming to be someone else. Similarly, modifying the pid is equivalent to the user alleging a different perpetrator when it has already alleged someone else. Hence, there are several checks that the escrows must perform before modifying an allegation to restrict the modification to the above components.

First, the escrows must verify that a registered user of the system is submitting the request. Second, the escrows must verify that the user submitting the (updated) allegation is not impersonating another user. The first check is addressed by the escrows and the user carrying out the same steps as in the allegation filing phase, where the user is now expected to instead submit the updated allegation by consuming a new DS key pair $(\mathsf{pk}^{\mathsf{U}}, \mathsf{sk}^{\mathsf{U}})$. For the second check, to ensure the user is attempting to modify its own allegation, the escrows verify the validity of the submitted uid by recomputing it as done in the duplicity check protocol. Once validated, the escrows obliviously identify and update the allegation $\mathsf{a} \in \mathsf{A}$ in question. Note that knowing which allegation was updated may leak sensitive data based on auxiliary information such as the time of the filed allegation and the time of the modification. Hence, similar to the duplicity check protocol, escrows scan through each allegation a_i in the system and check for equality of uid and pid of a_i and submitted a'. Whenever these components are a match, the escrows obliviously update the components t and Text via Π_{Sel}. The escrows additionally maintain a flag to detect if an allegation was modified successfully or not. This is done to ensure that a valid yet malicious user does not burden the system with fake requests to modify an allegation. If all the checks pass and yet the flag is not set, it indicates that no allegation was modified. Hence, the submitted pk^{U} can be used to trace back the user, as done in the allegation revealing phase, and penalize it accordingly. That is, the escrows compute the MAC on the submitted pk^{U} and de-anonymize the user through the local map of mac(s). The formal protocol for the same is provided in Fig. 6.6.

Protocol $\Pi_{\mathsf{Mod}}(\{\mathsf{a}_1, \ldots, \mathsf{a}_N\}, \mathsf{a}', \mathsf{sk}_m)$

– Initiate *allegation filing* phase with a' as the submitted allegation. If any verification fails, ignore processing of a', discard it and halt. Else set $[\![\mathsf{flag}]\!] = 0$ and continue.

– $[\![\mathsf{uid}']\!] = \Pi_{\mathsf{mac}}([\![\mathsf{sk}_m]\!], [\![\mathsf{a}'.\mathsf{r}]\!])$ and $[\![\mathsf{chk}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{uid}']\!], [\![\mathsf{a}'.\mathsf{uid}]\!])$

– Reconstruct chk. If $\mathsf{chk} = 1$, then for each allegation $\mathsf{a}_i \in \mathsf{A}$,

- $[\![\mathsf{chk}_1^i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_i.\mathsf{uid}]\!], [\![\mathsf{a}'.\mathsf{uid}]\!])$
- $[\![\mathsf{chk}_2^i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_i.\mathsf{pid}]\!], [\![\mathsf{a}'.\mathsf{pid}]\!])$
- $[\![\mathsf{chk}_3^i]\!]^{\mathbf{B}} = [\![\mathsf{chk}_1^i \cdot \mathsf{chk}_2^i]\!]^{\mathbf{B}}$
- $[\![\mathsf{a}_i.\mathsf{t}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{a}_i.\mathsf{t}]\!], [\![\mathsf{a}'.\mathsf{t}]\!], [\![\mathsf{chk}_3^i]\!]^{\mathbf{B}})$
- $[\![\mathsf{a}_i.\mathsf{Text}]\!] = \Pi_{\mathsf{Sel}}([\![\mathsf{a}_i.\mathsf{Text}]\!], [\![\mathsf{a}'.\mathsf{Text}]\!], [\![\mathsf{chk}_3^i]\!]^{\mathbf{B}})$

– Reconstruct $\mathsf{flag} = \Pi_{\mathsf{DotP}}(\{[\![\mathsf{chk}_1^i]\!]^{\mathbf{B}}, [\![\mathsf{chk}_2^i]\!]^{\mathbf{B}}\}_{i=1}^{\mathsf{N}})$. If $\mathsf{flag} = 1$, report success. Else trace malicious user via a map of mac(s) on verification keys.

Figure 6.6: Allegation modification.

Note that each successful request of allegation modification must be followed by the allegation matching phase since updating the threshold can trigger the possibility of having a revealable subset.

6.4.2 Allegation deletion

Not only must a user U be allowed to modify an allegation, but the system must also facilitate U to delete the same. The designed system facilitates deletion and works on similar lines of allegation modification. U places the request for deletion by resubmitting the allegation to be deleted, albeit under a new pair of DS key pair $(\mathsf{pk}^{\mathsf{U}}, \mathsf{sk}^{\mathsf{U}})$. The escrows perform the same verification as done in the allegation filing phase. If all the verification succeeds, the escrows recompute the uid' using r and match it against the submitted uid, as done in the allegation modification. If the check passes, the escrows identify the allegation to be deleted. Unlike in the case of the allegation modification protocol, where it was necessary to hide the allegation being modified, we note that for deleting an allegation, the escrows must learn which allegation it is. Hence, the identification of the allegation in question need not be performed obliviously. This also avoids explicitly maintaining a flag variable. Thus, escrows perform a linear scan over the list of all allegations to determine the a_i that has uid (using chk_1^i) and pid (using chk_2^i) equal to that of the submitted allegation a'. The escrows determine if both the equalities hold (using chk_3^i) and delete their shares of a_i if this is the case. If no such allegation a_i is found, then the escrows determine the malicious user by tracing the identity of the user using the submitted pk^{U}. The formal protocol for the same is given in Fig. 6.7.

Protocol $\Pi_{\mathsf{Del}}(\{\mathsf{a}_1, \ldots, \mathsf{a}_N\}, \mathsf{a}', \mathsf{sk}_m)$

– Initiate *allegation filing* phase with a' as the submitted allegation. If any verification fails, ignore processing of a', discard it and halt. Else continue.

– $[\![\mathsf{uid}']\!] = \Pi_{\mathsf{mac}}([\![\mathsf{sk}_m]\!], [\![\mathsf{a}'.\mathsf{r}]\!])$ and $[\![\mathsf{chk}]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{uid}']\!], [\![\mathsf{a}'.\mathsf{uid}]\!])$

– Reconstruct chk. If $\mathsf{chk} = 1$, then for each allegation $\mathsf{a}_i \in \mathsf{A}$,

- $[\![\mathsf{chk}_1^i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_i.\mathsf{uid}]\!], [\![\mathsf{a}'.\mathsf{uid}]\!])$
- $[\![\mathsf{chk}_2^i]\!]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([\![\mathsf{a}_i.\mathsf{pid}]\!], [\![\mathsf{a}'.\mathsf{pid}]\!])$
- $[\![\mathsf{chk}_3^i]\!]^{\mathbf{B}} = [\![\mathsf{chk}_1^i \cdot \mathsf{chk}_2^i]\!]^{\mathbf{B}}$
- Reconstruct chk_3^i. If $\mathsf{chk}_3^i = 1$ delete a_i, report success, halt.

– If no a_i was deleted, then trace the malicious user using the local map of mac(s) on verification keys.

Figure 6.7: Allegation deletion.

A possible optimization is to associate a public token with every allegation and enforce the user to also submit the corresponding token when requesting deletion. This allows the escrows to identify the requested allegation to be deleted without relying on MPC. However, the escrows are required to use MPC to verify that the requested allegation is, in fact, filed by the user. Hence, all checks performed previously remain the same, except they are performed specific to the identified allegation directly. This avoids the linear scan across A.

We finally would like to note that our system can be extended to handle different types of crimes. Towards this, the allegation can include an additional field indicating the type. Note that this would affect the duplicity check and matching protocols and require careful re-definitions of the same.

6.5 Discussions

Here we address some concerns that may arise with respect to the design of Shield.

Privacy vs. efficiency The inception of SAE systems was to protect user privacy and thereby encourage user participation. Hence, for a user-centric system such as SAE, privacy should be given utmost importance and must not be compromised at any cost. Further, a user who filed an allegation is inherently bound to wait until matching allegations are found and can be revealed. Given that this waiting period can vary from days to months [198, 14], improving the efficiency of the system (complexity of processing a filed allegation) has an insignificant effect on the wait time of the user. Thus, choosing efficiency over privacy is ill-advised for an

SAE system.

Pessimistic view of the world Our design decisions are made considering the worst-case scenario. Specifically, we assume malicious capabilities of users and escrows and their collusion. The protocols in Shield allow the escrows to detect misbehaviour by a malicious user. Relying on the accountability property offered by Shield, the escrows (or higher authorities) can trace back the user's real-world identity and punish the same. Thus, the accountability property plays an important role in deterring malicious users. Further, the choice of a variable threshold instead of a globally fixed one is made to cater to the victims' varying levels of vulnerability, thereby making the system more inclusive. The presence of a flexible reveal threshold further demands empowering a victim to change the reveal threshold of its allegation, which may be necessary to ensure the allegation is revealed sooner. This is facilitated by the allegation modification protocol. Note that our design choices in no way make us loose-out on any feature provided by prior systems, rather only improve upon them.

Choice of MPC Note that an allegation escrow system has highly sensitive information and is required to run perpetually. Realization of the system cannot afford to cease providing the service due to malicious activities. Thus, employing any MPC protocol that enables the adversary to abort would be a misfit. However, previous works rely on MPC with identifiable abort (i.e., the protocol aborts and reveals the identity of the corrupt party upon misbehaviour) but restart the computation with one less party when a corrupt party is identified. Although this is a possibility, it comes at the cost of re-sharing the state of the corrupt party (escrow) that is thrown out of the computation, which is expensive. Hence, it is imperative to design protocols that attain the strongest security of guaranteed output delivery. GOD prevents an adversary from wasting honest escrow's valuable compute resources by preventing repeated failures, which is otherwise possible in the weaker security notions that allow an adversary to abort. Thus, the presence of GOD uplifts the trust in the system and encourages user participation. It is well known that an honest majority among the computing parties is necessary to achieve GOD [55]. Moreover, due to the challenges in identifying a large number of compute parties for real-world deployments and efficiency reasons, MPC for a small number of parties is gaining huge interest [175, 11, 84, 12, 36, 183, 49, 193, 29, 193, 50, 38, 30]. Hence, to realize an SAE system, we focus on benchmarking honest majority MPC protocols with a small number of parties providing the strongest security of GOD. Specifically, we consider the 5PC (1,1)-FaF setting discussed in Chapter 5.

Other challenges Several details, such as the requirement of anonymous communication channels for alleger anonymity, trust in the user's client-side software, and other deployment considerations, which apply to our solution, are not emphasized. These follow from prior works, as described in the deployment considerations of [14]. Our goal was to identify privacy breaches in the existing works and formalize the security desirable in such a system. Thus, we only provide an algorithmic solution that achieves the desired security. Addressing the system-level challenges that may arise in the actual deployment of the solution and designing a user-friendly interface is the necessary next step and is left as future work.

6.6 Benchmarks

Shield, can be realized using any MPC protocol that provides the identified primitives in Table 6.2. Although we prioritize privacy over efficiency, we strive to achieve as efficient a solution as possible. We instantiate our (1, 1)-FaF secure 5PC protocols described in §5. The protocols are implemented in Python. Our code accounts for multithreading. We instantiate the communication layer between the parties using the PyTorch library. We use the Crypto library for AES and hashlib for generating SHA256 hash. Our code was developed for benchmarking and not optimized for industry-grade use. We note that a C++-based implementation can give better performance. Our protocols are benchmarked over LAN, instantiated using n1-standard-64 instances of Google Cloud with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors, and 240 GB of RAM. The machines have a bandwidth of 16Gbps. We use latency (time taken for the protocol to complete) and communication between escrows as the two parameters for benchmarks. We report values separately for the online phase and total (= preprocessing+online). Our protocols are benchmarked over the ring $\mathbb{Z}_{2^\ell}$ ($\ell = 64$), except for computing MAC and VRF, the security of which demands operating on a 1024-bit prime-order field.

Recall the six phases that comprise the Shield system. Observe that initialization is a one-time process, and hence does not contribute to the cost of keeping the system running. However, the escrows may be challenged to register multiple users at once. Hence, we report the cost of registering a single user in Table 6.4, where we fix the number of digital signature keys being registered by the user as 50 following [14].

Allegation filing involves escrows locally performing operations (without any interaction). Hence, we do not explicitly report the cost of this phase. Duplicity check and allegation matching make up the compute-intensive phases for running the system. Unlike registration, these phases can only process one allegation at a time. Hence, we report costs for processing one filed allegation in these phases. These costs were not reported in [14] since it had a constant-

#Escrows	Online		Total	
	Latency (s)	Com (MB)	Latency (s)	Com (MB)
5	6.87	164.81	12.11	368.46

Table 6.4: Communication and latency for registering a user.

time matching (and a duplicity check was missing). Hence, the costs reported here capture the overhead in comparison to [14], which is the price paid for obtaining full privacy.

The complexity of our duplicity check is dependent on the number of allegations in the system and hence is benchmarked for a varying number of allegations. For this, we consider a system where the number of filed allegations ranges from 100-100,000, which is sufficient to account for deployment in any institution (e.g., universities, private or government workplaces, etc.). We note that 10^5 filed (unrevealed) allegations account for a pessimistic view, so real-world deployment may be faster. The variations in latency and communication for online and preprocessing phases appear in Table 6.5. The reported online latency is roughly 22 minutes, even in the presence of 10^5 allegations, which showcases its practicality. As expected, communication scales linearly with the number of allegations.

#Escrows	$\lvert\mathsf{A}\rvert$	Online		Total	
		Latency (s)	Com (MB)	Latency (s)	Com (GB)
5	10^2	2.45	2.46	5.27	0.05
	10^3	8.45	14.63	20.86	0.36
	10^4	128.16	91.48	179.65	3.83
	10^5	1325.54	913.67	1559.91	38.37

Table 6.5: Communication and latency for duplicity check.

The above analysis also holds for matching since it too depends on the number of allegations in the system. Additionally, the matching protocol depends on the size of the revealed perpetrator list P, and the upper bound on the reveal threshold. To analyze the effect of this, we benchmark cases where $|\mathsf{P}|$ may be linear (1/10) or sub-linear ($\sqrt{\cdot}$) in the number of allegations and bound the reveal threshold by 10. These results are reported in Table 6.6. $|\mathsf{P}|$ is chosen to be linear and sub-linear in the number of allegations to account for a large band of variation in the possible number of revealed perpetrators. As evident from Table 6.6, in the presence of 10^5 allegations in the system, processing a new allegation requires < 21 minutes in the online phase. Thus, the online run time of the matching protocol showcases its practicality despite having a dependence on the number of allegations in the system. This can be attributed to the use of the preprocessing model. Moreover, for 10^5 allegations, changing $|\mathsf{P}|$ from 300 to 10^4

has an overhead of not more than 3 minutes in online runtime, and less than 3 MB overhead in online communication. This shows the minimal effect $|\mathsf{P}|$ has on the complexity of matching. Further, to showcase the effect of threshold bound on the complexity of matching, we vary the bound from 10 to 50 with fixed $|\mathsf{A}| = 10^5$, $|\mathsf{P}| = 10^4$ and report results in Table 6.7. An increase in bound from 10 to 50 results in an overhead of 2.14×, 4.65× in online latency and online communication, respectively. A pictorial comparison of matching and duplicity appears in Fig. 6.8.

#Escrows	$\|\mathsf{A}\|$	$\|\mathsf{P}\|$	Online		Total	
			Latency (s)	Com (MB)	Latency (s)	Com (MB)
5	10^2	10	1.13	0.31	3.48	2.25
	10^3	30	15.09	3.04	32.5	18.35
		10^2	25.24	3.12	47.89	20.06
	10^4	10^2	116.34	31.11	281.04	195.59
		10^3	179.17	30.8	342.51	196.35
	10^5	300	1109.22	312.48	2464.68	1947.45
		10^4	1256.66	310.46	2972.86	1956.80

Table 6.6: Overhead of matching for varying number of allegations and number of revealed perpetrators.

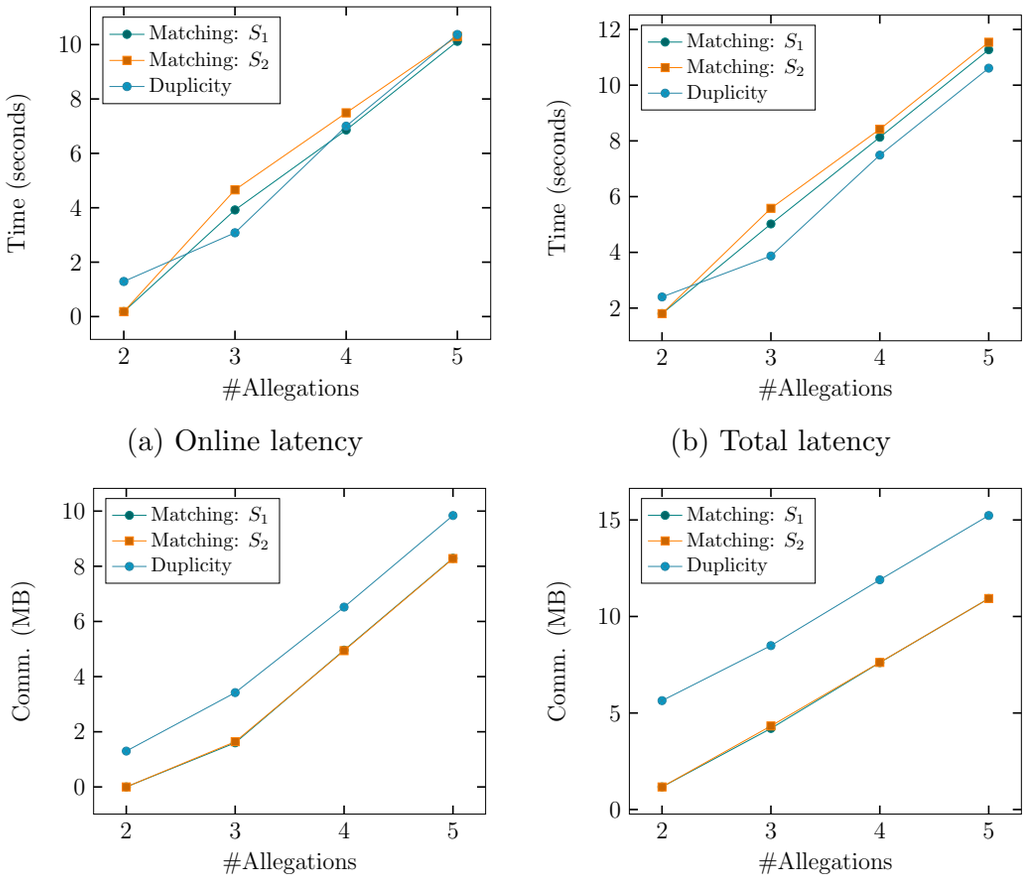
#Escrows	Max. threshold	Online		Total	
		Latency (s)	Com (MB)	Latency (s)	Com (MB)
5	10	1256.66	310.46	2972.86	1956.80
	20	1910.33	590.28	3797.48	1966.61
	30	2371.89	889.77	4329.94	2885.84
	40	2428.67	1164.79	5041.09	3804.45
	50	2656.61	1444.27	5777.38	4723.02

Table 6.7: Communication and latency for matching with varying upper bounds on threshold for $|\mathsf{A}| = 10^5$, $|\mathsf{P}| = 10^4$.

Since the operations in the allegation modification are similar to those in the duplicity check, its cost is the same as that reported for the duplicity check. The same holds true for deletion.

6.7 Security proofs

The simulation-based $(1, 1)$-FaF secure proofs for Shield are presented in this section. Let $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ denote the ideal-world malicious adversary and the ideal-world semi-honest adversary,



(a) Online latency

(b) Total latency

(c) Online comm.

(d) Total comm.

Figure 6.8: Variation in latency and communication for varying number of allegations for duplicity and matching protocol. For matching, S_1 indicates the setting where #perpetrators revealed are sublinear in #allegations, while S_2 indicates a setting where #perpetrators revealed is 10% of #allegations. Plots are log-log plots with x-axis logarithmic in base 10 and y-axis logarithmic in base 2.

respectively. Let $\mathcal{A}$ and $\mathcal{A}_{\mathcal{H}}$ denote the real-world malicious adversary and the real-world semi-honest adversary, respectively. We begin by discussing the security of Π_{Dup} and Π_{Mat}, followed by proving the security of Shield.

We give the ideal functionality for duplicity check and matching in Fig. 6.9 and Fig. 6.10, respectively. Owing to the modular architecture (see Fig. 6.1), it is easy to observe that the duplicity and matching protocols in layer 2 build on top of protocols in layer 1 and layer 0. Hence, their simulation follows from the simulation of underlying protocols.

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

$\mathcal{F}_{\mathsf{Dup}}$ interacts with the escrows in $\mathcal{P}$, the ideal world malicious adversary $\mathcal{S}_{\mathcal{A}}$ and the ideal world semi-honest adversary $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

– It receives as input the shares of the following from the parties: the newly filed allegation $\mathsf{a_{new}}$, allegations in the system $\{\mathsf{a_1}, \ldots, \mathsf{a_N}\}$, and the MAC key $\mathsf{sk_m}$.

– Reconstruct $\mathsf{a_{new}}$, $\{\mathsf{a_1}, \ldots, \mathsf{a_N}\}$ and $\mathsf{sk_m}$ using the shares of honest parties.

– Recompute the user ID $\mathsf{uid}' = \mathrm{MAC}(\mathsf{sk_m}, \mathsf{a_{new}.r})$ and check if $\mathsf{uid}' = \mathsf{a_{new}.uid}$.

– If it is not equal, send *ignore* message to escrows and terminate.

– Else check if there exists $\mathsf{a_i} \in \{\mathsf{a_1}, \ldots, \mathsf{a_N}\}$ such that $\mathsf{a_i.uid} = \mathsf{a_{new}.uid}$ and $\mathsf{a_i.pid} = \mathsf{a_{new}.pid}$.

– If $\mathsf{a_i}$ as described above exists, output 1 to all the escrows. Else, it outputs 0.

$\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 6.9: Ideal functionality for duplicity check.

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

$\mathcal{F}_{\mathsf{Mat}}$ interacts with the escrows in $\mathcal{P}$, the ideal world malicious adversary $\mathcal{S}_{\mathcal{A}}$ and the ideal world semi-honest adversary $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

– Receive the shares of the following from the parties: $\mathsf{a_{new}}$, $\{\mathsf{a_1}, \ldots, \mathsf{a_N}\}$, and all entries p_i in P.

– Reconstruct $\mathsf{a_{new}}$, $\{\mathsf{a_1}, \ldots, \mathsf{a_N}\}$ and P using received shares of honest parties.

– (**Step 1**) Check if there exists entry p_i in P such that $\mathsf{p}_i.\mathsf{pid}$ matches $\mathsf{a_{new}.pid}$. If found, reduce $\mathsf{a_{new}.t}$ by $\mathsf{p}_i.\mathsf{ac}$.

– Include $\mathsf{a_{new}}$ as the $\mathsf{N}+1^{th}$ allegation.

– (**Step 2**) Consider all subsets of matching allegations to $\mathsf{a_{new}}$ and check if any of them satisfy their reveal criteria. Let $\mathcal{S}$ denote such a set, if found.

– If no such set is found send *ignore* message to escrows and terminate.

– Else (**Step 3**) consider all those matching allegations in $\{\mathsf{a_1}, \ldots, \mathsf{a_N}\}$ but not in $\mathcal{S}$. For each such allegation $\mathsf{a_i}$, update its threshold to $\mathsf{a_i.t} - |\mathcal{S}|$.

– (**Step 4**) Check if there exists a record p_i such that $\mathsf{p}_i.\mathsf{pid} = \mathsf{a_{new}.pid}$. If such a entry is found, update $\mathsf{p}_i.\mathsf{ac}$ by adding $|\mathcal{S}|$ to it. Else, create a new entry in P as $(\mathsf{pid}, \mathsf{ac}) = (\mathsf{a_{new}.pid}, |\mathcal{S}|)$ and send message ("New P entry") to all escrows.

– Send the index i for all allegations $\mathsf{a_i}$ in $\{\mathsf{a_1}, \ldots, \mathsf{a_N}\}$ that are now included in $\mathcal{S}$ to the escrows. Delete these entries from the list of all allegations.

$\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 6.10: Ideal functionality for matching.

Lemma 6.1 (Security) *Protocol* Π_{Dup} *(Fig. 6.4) securely realizes* $\mathcal{F}_{\mathsf{Dup}}$ *(Fig. 6.9) in the computational setting against FaF adversaries* $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Proof: Simulation proceeds via the simulation steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. Note that since the simulator receives from the ideal functionality the values that are reconstructed during the protocol execution, the simulator can also successfully reconstruct this towards the adversary. □

Lemma 6.2 (Security) *Protocol* Π_{Mat} *(Fig. 6.5) securely realizes* $\mathcal{F}_{\mathsf{Mat}}$ *(Fig. 6.10) in the computational setting against FaF adversaries* $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Proof: Simulation proceeds via the simulation steps of the underlying protocols. Thus, the indistinguishability of the simulated and real-world view of the adversary follows from the indistinguishability of the simulation steps of the underlying protocols. Further, regarding the simulation of break, note that the simulator receives from the ideal functionality the values that are reconstructed during the protocol execution. This allows the simulator to reconstruct the values towards the adversary, thereby enabling the simulation of the break statements too. Further, note that the point at which protocol breaks discloses the number of allegations that can be revealed, if any. Else, the loop breaks after a public number of iterations (equals maxths). In either case, escrows always learn this information depending on if they reveal/not reveal a set of allegations. Hence, it is not a breach of privacy. □

We now prove the security of the Shield.

Theorem 6.1 *Let* VRF *be a secure distributed input VRF protocol, let* MAC *be a secure MAC, and let the employed signature scheme be strongly existentially unforgeable. Then* Shield *realizes* $\mathcal{F}_{\mathsf{Shield}}$ *(Fig. 6.2) with computational security in the* $(\mathcal{F}_{\mathsf{Dup}}, \mathcal{F}_{\mathsf{Mat}})$*-hybrid model with* $(1,1)$*-FaF security.*

Proof: We provide the description of a simulator $\mathcal{S}_{\mathcal{A}}$ designed to simulate the view of the malicious adversary $\mathcal{A}$ in the real-world and a simulator $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ to simulate the view of a semi-honest adversary $\mathcal{A}_{\mathcal{H}}$. We assume $\mathcal{A}$ can corrupt at most one escrow and any number of users. We begin with steps followed by $\mathcal{S}_{\mathcal{A}}$ and then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

The initialization phase involves interaction only among the escrows. $\mathcal{S}_{\mathcal{A}}$ emulates the distributed key generation[4] to generate the public key $\mathsf{pk_v}$ and secret key $\mathsf{sk_v}$ required to evaluate

[4] A distributed key generation functionality outputs $[\![\cdot]\!]$-shares of a secret key $\mathsf{sk_v}$, and the public key $\mathsf{pk_v}$, on clear, to the escrows. A secure protocol for this can be realized using the $(1,1)$-FaF secure 5PC.

VRF. It also generates the secret key $\mathsf{sk_m}$ required to evaluate MAC. It sends the corrupt escrow's shares of $\mathsf{sk_v}$, $\mathsf{sk_m}$ to $\mathcal{A}$ while the public key is sent in clear. Note that $\mathcal{S}_\mathcal{A}$ is aware of all the keys generated so far. The simulator also emulates the CA.

Since the registration and allegation filing phases require interaction between escrows and users, we consider the following two cases:

Case 1: Honest user and corrupt escrow

Upon receiving the message ("Register", c, ID) from $\mathcal{F}_{\mathsf{Shield}}$, $\mathcal{S}_\mathcal{A}$ generates maxalg new public keys $\mathsf{pk}_1^\mathsf{U}, \ldots, \mathsf{pk}_{\mathsf{maxalg}}^\mathsf{U}$ for a user U, a random r and provides the secret-shares of these, corresponding to the corrupt escrow, to $\mathcal{A}$. $\mathcal{S}_\mathcal{A}$ then participates in the distributed computation of the VRF on the received public keys and the computation of MAC on r to generate secret-shares of the VRF on the supplied public keys and shares of uid, respectively . This computation is simulated by running simulation steps with respect to Π_{vrf} and Π_{mac}, which outputs random shares of VRF on the public keys and random shares of uid. $\mathcal{S}_\mathcal{A}$ simulates generation and reconstruction of $\mathsf{mac}_i^\mathsf{U} = \mathsf{MAC}(\mathsf{sk_m}, \mathsf{pk}_i^\mathsf{U})$ for $i \in \{1, \ldots, \mathsf{maxalg}\}$ towards $\mathcal{A}$ ($\mathcal{S}_\mathcal{A}$ can simulate this since it has the key $\mathsf{sk_m}$ and pk_i^U). So far the view generated by $\mathcal{S}_\mathcal{A}$ is indistinguishable from $\mathcal{A}$'s real world view.

Upon receiving the message ("Allege") from $\mathcal{F}_{\mathsf{Shield}}$ regarding the attempt to file a new allegation, $\mathcal{S}_\mathcal{A}$ chooses a pk^U which has been authenticated in the registration steps and chooses random shares for each component in the allegation, signs these under pk^U and sends it to $\mathcal{A}$. On receiving the message about duplicity check from $\mathcal{F}_{\mathsf{Shield}}$, $\mathcal{S}_\mathcal{A}$ emulates $\mathcal{F}_{\mathsf{Dup}}$ accordingly and proceeds to the matching phase. Similarly, depending on the message received from $\mathcal{F}_{\mathsf{Shield}}$ for the matching phase, $\mathcal{S}_\mathcal{A}$ emulates $\mathcal{F}_{\mathsf{Mat}}$. Finally, to simulate the revealing of an allegation depending on the user identifier ID received from $\mathcal{F}_{\mathsf{Shield}}$, $\mathcal{S}_\mathcal{A}$ picks a pk that was filed with respect to ID. Since $\mathcal{S}_\mathcal{A}$ knows the key $\mathsf{sk_m}$, it simulates steps of MAC computation such that it leads to reconstructing $\mathsf{MAC}(\mathsf{sk_m}, \mathsf{pk})$ towards $\mathcal{A}$. The allegation components received from $\mathcal{F}_{\mathsf{Shield}}$ are reconstructed towards $\mathcal{A}$. Observe that in all the above steps, since $\mathcal{S}_\mathcal{A}$ follows the simulation steps of the underlying protocols, the indistinguishability of the simulation follows from the indistinguishability of the underlying simulations.

Case 2: Corrupt user and corrupt escrow

During registration, $\mathcal{A}$ sends the proof of ID, c obtained from the CA for a corrupt user U to $\mathcal{S}_\mathcal{A}$. It also sends the honest escrows' shares of the maxalg public keys $\mathsf{pk}_1^\mathsf{U}, \ldots, \mathsf{pk}_{\mathsf{maxalg}}^\mathsf{U}$ and a random r to $\mathcal{S}_\mathcal{A}$. If the proof of ID is invalid or the shares are inconsistent, $\mathcal{S}_\mathcal{A}$ sends $\perp$ to $\mathcal{A}$. Else, it sends ("Register", c, ID) to $\mathcal{F}_{\mathsf{Shield}}$ from the corrupted alleger's ID. Since $\mathcal{S}_\mathcal{A}$ knows the shares with respect to all honest escrows, it can reconstruct the underlying values (pk_i^U and r). $\mathcal{S}_\mathcal{A}$ then participates in the distributed computation of the VRF and MAC on the keys and on

r, submitted by $\mathcal{A}$, to generate shares of the VRF on the public keys and shares of uid. This computation is simulated by running simulation steps with respect to Π_{vrf} and Π_{mac}. Further, $\mathcal{S}_{\mathcal{A}}$ simulates generation and reconstruction of $\mathsf{mac}_i^{\mathsf{U}} = \mathsf{MAC}(\mathsf{sk}_{\mathsf{m}}, \mathsf{pk}_i^{\mathsf{U}})$, for $i \in \{1, \ldots, \mathsf{maxalg}\}$ towards $\mathcal{A}$.

While filing an allegation, $\mathcal{A}$ sends honest escrow's shares of the allegation components to $\mathcal{S}_{\mathcal{A}}$, who checks if the submitted pk^{U} is valid and if had not been used earlier. If the check fails, $\mathcal{S}_{\mathcal{A}}$ sends $\perp$ to $\mathcal{A}$. Else, $\mathcal{S}_{\mathcal{A}}$ determines the ID with which pk^{U} is registered (recall that $\mathcal{S}_{\mathcal{A}}$ is able to do it since it can reconstruct all pk^{U}'s submitted during registration) and connects to $\mathcal{F}_{\mathsf{Shield}}$ on ID's channel.

Following this, on receiving the message about duplicity check from $\mathcal{F}_{\mathsf{Shield}}$, $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Dup}}$ accordingly and proceeds to the matching phase. Similarly, depending on the message received from $\mathcal{F}_{\mathsf{Shield}}$ for the matching phase, $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{Mat}}$. On receiving a message from $\mathcal{F}_{\mathsf{Shield}}$ to reveal an allegation filed by a corrupt user U with identity ID, $\mathcal{S}_{\mathcal{A}}$ does the following. Since $\mathcal{S}_{\mathcal{A}}$ knows the key sk_{m}, it simulates steps of Π_{mac} such that it leads to reconstructing $\mathsf{MAC}(\mathsf{sk}_{\mathsf{m}}, \mathsf{pk}^{\mathsf{U}})$ towards $\mathcal{A}$, where pk^{U} is the key used by the corrupt user for filing the allegation. The allegation components received from $\mathcal{F}_{\mathsf{Shield}}$ are reconstructed towards $\mathcal{A}$. Observe that in all the above steps, since $\mathcal{S}_{\mathcal{A}}$ follows the simulation steps of the underlying protocols, the indistinguishability of the simulation follows from the indistinguishability of the underlying simulations.

Finally, $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$. Observe that $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ only has to simulate semi-honest escrow, which is similar to the steps carried out above, with the difference that, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ will invoke the semi-honest simulator of the underlying protocols and reuse the messages present in the view provided by $\mathcal{S}_{\mathcal{A}}$, wherever required. Further, only the case with the honest user and corrupt escrow has to be simulated. Due to the close resemblance, we omit the details.

□

Chapter 7

Secure Computation with Constant Number of Parties

In this chapter, we discuss the secure protocols that allow performing computation with multiple (constant) parties. The results in this chapter have led to a publication in the Journal of Cryptology 2023 [140].

7.1 Overview

We extend the strategies of the small-party computation protocols that tolerate at most one corruption to support higher resiliency in an honest-majority setting while keeping the efficiency of the online phase at the centre stage. In this regard, we design secure multiparty protocols (for a constant number of parties) in the honest-majority setting. We begin with a quick overview of the results, followed by the details.

– We construct an n-party semi-honest protocol, tolerating at most $t < n/2$ corruptions, in the preprocessing paradigm, which offers an improved online phase than the (optimized) protocol of [62] that appears in [29], without inflating its total cost. Henceforth, we refer to the optimized protocol of [62], that appears in [29], as DN07*. Moreover, our protocol reduces the number of active parties in the online phase, thereby improving the system's operational cost.

– We extend our semi-honest protocol to the malicious setting while retaining the benefits of requiring a reduced number of parties in the online phase for the majority of the computation. Our offer over the state-of-the-art protocol of [78] is a stronger security guarantee of fairness and at least 2× improvement in round complexity via a one-time verification at the end of protocol evaluation.

– We provide support for 3 and 4 input multiplication at the same online complexity as that of the 2 input multiplication. In addition to improving the communication cost over the approach of sequential multiplications, multi-input multiplication offers a 2× improvement in the round complexity, which is beneficial for high-latency networks. Moreover, the approach can be extended to an arbitrary number of inputs while retaining the same online communication, albeit requiring exponential communication in the preprocessing phase [194].

– We design building blocks for a range of applications, such as deep neural networks and genome sequence matching based on edit distance and Euclidean distance. When the applications are benchmarked, our semi-honest protocol witnesses a saving of up to 69% in monetary cost and has 3.5× to 4.6× improvements in online run time and throughput over DN07*. Interestingly, our maliciously secure protocols outperform the semi-honest protocol of DN07* in terms of online run time and throughput for the applications under consideration, achieving the goal of a fast online phase.

We now elaborate on the contributions and highlight the technical details and novelty.

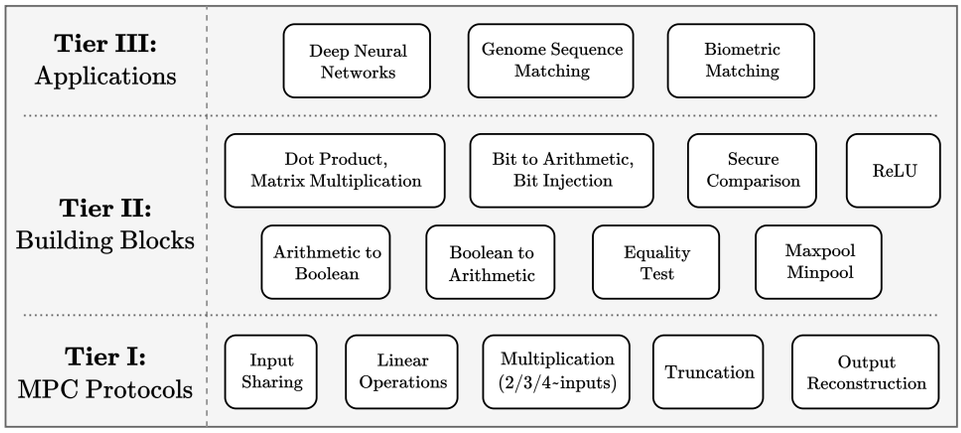


Figure 7.1: Hierarchy of primitives in our 3-tier framework.

The designed multiparty framework follows a 3-tier architecture (Fig. 7.1) to attain the final goal of privacy-conscious computations. The first tier comprises fundamental primitives such as input sharing, reconstruction, multiplication (with truncation), and multi-input multiplication. The second tier includes building blocks such as dot product, matrix multiplication, conversion between Boolean and arithmetic worlds, comparison, equality, and non-linear activation functions, to name a few, as required in the applications considered. Finally, the third tier is applications. Our main contributions lie in Tier I, and these are detailed below. Going ahead, we use 'multiparty protocols' to mean honest majority n-party protocols that tolerate $t > 1$ corruptions and thus do not include the tailor-made protocols in the 3 and 4-party settings.

Tier I - MPC protocols

Our goal is to design protocols with a fast online phase. Thus, working over $\mathbb{Z}_{2^\ell}$ and relying on replicated secret sharing (RSS), we design a semi-honest MPC protocol in the computational setting assuming a one-time shared-key setup for correlated randomness.

Note that the straightforward extension of the semi-honest multiplication protocol of DN07* to the preprocessing model, which can also be derived from the recent work of [78], incurs a communication of $3t$ elements in the preprocessing phase while communicating $2t$ elements in the online. This amounts to a $1.6\times$ overhead in the total cost over DN07*. Our contribution lies in ensuring a fast online phase without inflating the total communication cost of the protocol. Specifically, our protocol requires communicating only $2t$ ring elements in the online phase and t in the preprocessing for a multiplication gate. Thus, in the honest majority multiparty setting over rings, we are the first to achieve a communication cost of $2t$ in the online phase (unlike $3t$ in the prior works [62, 85]) without incurring any overhead in the total cost, i.e., our total cost still matches that of the best known (optimized) semi-honest honest-majority protocol [62, 85].

We extend our protocol to provide malicious security with *fairness* at the cost of additionally communicating t elements in the online phase and $2t$ in the preprocessing phase. Although (*abort*) protocol of [78] has the same communication as our maliciously secure protocol, we achieve a stronger security notion of fairness. Moreover, [78] requires an additional round of communication for consistency checks after each level, the absence of which results in a privacy breach (described in [94] and elaborated in §7.5.3), and necessitates participation from all parties. However, by relying on a variant of RSS, our protocol avoids the consistency check after each level of circuit evaluation and ensures privacy. Notably, we only require participation from all parties for a one-time verification at the end of the evaluation, thus reducing the number of rounds by d (d denotes circuit depth).

3 and 4 input multiplications. Following [194, 191, 138], to reduce the online communication cost and round complexity, we design protocols to enable the multiplication of 3 and 4 inputs in a single shot. Compared to the naive approach of performing sequential multiplications to multiply 3/4 inputs, the *multi-input multiplication* protocol enjoys the benefit of having the same online phase complexity as that of the 2-input multiplication protocol. This brings in a $2\times$ improvement in the online round complexity and improves the online communication cost. Support for multi-input multiplication enables usage of optimized adder circuits [194] for secure comparison and Boolean addition, thereby resulting in a faster online phase. The recent work of [95] also proposes a method to improve the round complexity of circuit evaluation by evaluating all gates in two consecutive layers in a circuit in parallel. We observe that their

method can be viewed as a variant of multi-input multiplication with 3 and 4 inputs. Thus, our protocols need not be limited to facilitating faster comparison and Boolean additions alone (as described above) but can be used to reduce the round and communication complexity of any general circuit evaluation. Note that [95] only improves the round complexity (2×) without inflating the communication cost when compared to DN07⋆. However, we focus on improving round complexity (2×) *as well as* communication of the online phase by trading off an increase in preprocessing.

Tier II - Building blocks

We design efficient protocols for several building blocks in semi-honest and malicious settings, which are stepping stones for Tier III applications. These are extensions from the small party setting [173, 193, 136, 194].

Tier III - Applications

To showcase the practicality of our framework and improvements of our protocols, we benchmark a range of applications that find use in the medical sector, such as neural networks (NN), which also includes the popular deep NN called VGG16 [213], genome sequence matching via edit distance and Euclidean distance, and are considered for the first time in the n-party honest-majority setting. We benchmark the applications in the WAN setting using Google Cloud instances. Owing to the inherent restrictions of RSS and keeping the focus on practical scenarios, we showcase the performance of our protocols for $n = 5, 7$, and 9 and compare them with the state-of-the-art semi-honest protocol of DN07⋆.

1. *Deep neural networks.* We benchmark inference phases of deep neural networks such as LeNet [147] and VGG16 [213]. We observe savings of up to 69% in monetary cost and improvements of up to 4.3× in online run-time and throughput, in comparison to DN07⋆.

2. *Genome sequence matching via edit distance.* We demonstrate an efficient protocol for similar sequence queries (SSQ), which can be used to perform secure genome matching. Our protocol is based on the protocol of [204], which works for 2 parties and uses an edit distance approximation [15]. We extend and optimize the protocol for the multiparty setting. In comparison to DN07⋆, we witness improvements of up to 4× in online run-time and throughput and savings of 66% in monetary cost.

3. *Biometric matching.* We propose efficient protocols for computing Euclidean distance (ED), which forms the basis for performing biometric matching. Continuing the trend, we witness a 4.6× improvement in online run-time and throughput over DN07* and savings of up to 85% in monetary cost.

7.2 Related work

Despite the interest in MPC for small population [11, 12, 84, 52, 3, 49, 193, 50, 38, 136, 220, 61, 138], MPC protocols for arbitrary number of parties (n) have been studied largely [78, 62, 93, 18, 23, 22, 31, 29, 34, 202, 7, 39, 25, 92] in the honest-majority ($t < n/2$) as well as the dishonest-majority ($t < n$) setting, where t denotes the maximum amount of allowed corruption. We restrict the related work to MPC protocols in the honest-majority setting, which is the focus of this work. In the honest majority setting, protocols can be categorized as working over the field algebraic structure [62, 85, 93, 95] or rings [29, 31, 78, 25, 18]. The field-based protocols, which mostly operate over Shamir secret sharing [206] scheme, have the advantage of having the share size linear in the number of parties. On the other hand, ring-based protocols are proven to be practically more efficient since they can leverage CPU optimizations [26, 66, 71, 67, 207]. In the following, we cover the field-based protocols first, followed by ring-based ones.

Field-based protocols: In the semi-honest case, [62, 85] provide MPC protocols over fields in the information-theoretic setting. ATLAS [95] further improves upon the communication complexity of [62] in the information-theoretic setting from $12t$ field elements to $8t$ field elements per multiplication gate. ATLAS [95] also provides another protocol variant, which improves the round complexity of [62] by 2× but requires slightly higher communication of $9t$ field elements. In the computational setting, the two protocol variants in ATLAS [95] roughly require communication of $4t$ and $5t$ field elements, respectively. The work of [78] demonstrates MPC protocols in the computational setting in the preprocessing model with malicious security. We observe that the semi-honest protocol derived from [78] requires communicating $2t$ elements in the online and $3t$ elements in the preprocessing phase.

In the malicious setting, the semi-honest protocol of [62] has served as the basis for obtaining malicious security for free (i.e. amortized communication cost of $3t$ field elements per multiplication gate) in the computational setting [31] as well as in the information-theoretic setting [93, 95]. These works follow the approach of executing a semi-honest protocol, followed by a verification phase to check the correctness of multiplication which involves heavy polynomial interpolation operations. As mentioned earlier, the work of [78] focuses on maliciously secure

protocols for honest majority setting in the preprocessing model. Their protocol relies on an instantiation of [93] in the preprocessing phase that requires communicating $3t$ field elements while requiring another $3t$ field elements communication in the online phase. However, their protocol is inefficient due to a consistency check required after each level of multiplication and introduces depth-dependent overhead in communication complexity. The absence of this check results in a privacy breach as described in [94] and is elaborated in §7.5.3.

Ring-based protocols: Operating over rings is challenging because they do not have inverses for every element, which fields do. One way to work with rings is to adapt a field-based protocol to work over rings, but this can be computationally intensive due to the use of an extension field [2, 78]. Another option is to use replicated secret shares (RSS) [114], which allows for direct operation over a ring without the need for extensions. However, this method results in share size becoming exponential in the number of parties due to the replication but can be more efficient when the number of parties is constant.

The work of [29] shows how the honest-majority semi-honest field-based MPC protocol of [62] can be optimized to work over rings using RSS. Operating in the computational setting using a one-time setup for correlated randomness, this optimized version of [62] has a communication cost of $3t$ ring elements per multiplication gate. This optimized honest-majority semi-honest protocol given in [29] is referred to as DN07* in the rest of the chapter. This protocol forms the state-of-the-art semi-honest protocol for the honest majority in the computational setting over rings and uses RSS. The work of [25, 18] also provides semi-honest MPC protocols over rings in the computational setting, which require *each* party to communicate roughly t elements per multiplication gate, resulting in quadratic communication in the number of parties. The work of [78], as described earlier, showcases how their field protocols can be extended to work over *rings* using Galois ring extensions. The semi-honest protocol derived from their maliciously secure variant requires communicating $2t$ and $3t$ extended ring elements in the online and preprocessing phases, respectively. In the malicious setting, [78] suffers from the privacy breach over rings as well (see §7.5.3 for details). Further, both [29, 31] provide protocols over rings. However, they rely on computationally heavy zero-knowledge machinery where expensive polynomial interpolation operations are carried out in the online phase.

Primitives: With respect to the primitives, note that these have been extensively studied in the literature and our contribution lies in adapting these for n-party setting, while incorporating improvements wherever possible. The relevant literature with respect to the primitives appears in §3.2, §4.2.

7.3 Preliminaries

7.3.1 System model

We consider both semi-honest and malicious adversarial models with *static* and, at most $t < n/2$ corruptions. For the rest of the chapter, we assume maximal corruption in this setting and thus $n = 2t+1$. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ denote the set of n parties which are connected by pair-wise private and authentic channels in a *synchronous* network. Additionally, our fair reconstruction protocol in the malicious setting relies on a broadcast channel, which can be instantiated using an existing broadcast protocol such as [74]. Set $\mathcal{E} = \{P_1, P_2, \ldots, P_{t+1}\}$, termed as the *evaluator* set, comprises parties that are active during the online phase. Set $\mathcal{D} = \{P_{t+2}, P_{t+3}, \ldots, P_n\}$, termed as the *helper* set, comprises parties which help in the preprocessing phase, and in the online verification in the malicious setting. Parties agree on a $P_{\mathsf{king}} \in \mathcal{E}$. Without loss of generality, let $P_{\mathsf{king}} = P_{t+1}$.

7.3.2 Sharing semantics

We use the following sharing semantics, based on replicated secret sharing (RSS) and additive sharing schemes, which facilitate a fast online phase.

1. $[\cdot]$-*sharing:* This denotes the replicated secret sharing of a value with threshold t. A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be RSS-shared with threshold t if for every subset $\mathcal{T} \subset \mathcal{P}$ of $n-t$ parties there exists $[\mathsf{a}]_{\mathcal{T}} \in \mathbb{Z}_{2^\ell}$ possessed by all $P_i \in \mathcal{T}$ such that $\mathsf{a} = \sum_{\mathcal{T}} [\mathsf{a}]_{\mathcal{T}}$.

   Alternatively, for every set of t parties, the residual $h = n - t$ parties forming the set $\mathcal{T}$, hold the share $[\mathsf{a}]_{\mathcal{T}}$. Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{\mathsf{q}} \subset \mathcal{P}$ be the distinct subsets of size h, where $\mathsf{q} = \binom{n}{h}$ represents the total number of shares. Since P_i belongs to $\binom{n-1}{h-1}$ such sets, it holds a tuple of $\binom{n-1}{h-1}$ shares, $\{[\mathsf{a}]_{\mathcal{T}}\}$. We denote this tuple of shares that it possesses as $[\mathsf{a}]_i$.

2. $\langle\cdot\rangle$-*sharing:* A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be $\langle\cdot\rangle$-shared (additively shared) among parties in $\mathcal{P}$ if $P_i \in \mathcal{P}$ possesses $\langle\mathsf{a}\rangle_i \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{a} = \langle\mathsf{a}\rangle_1 + \langle\mathsf{a}\rangle_2 + \ldots + \langle\mathsf{a}\rangle_n$.

3. ${}^{\mathcal{T}}\langle\cdot\rangle$-*sharing:* A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be ${}^{\mathcal{T}}\langle\cdot\rangle$-shared among $t+1$ parties in $\mathcal{T}$, if each $P_i \in \mathcal{T}$ holds ${}^{\mathcal{T}}\langle\mathsf{a}\rangle_i$ such that $\mathsf{a} = \sum_{P_i \in \mathcal{T}} {}^{\mathcal{T}}\langle\mathsf{a}\rangle_i$. We refer to this sharing scheme as $(t+1)$-additive sharing and use ${}^{\mathcal{E}}\langle\mathsf{a}\rangle$ to denote such a sharing among parties in $\mathcal{E}$.

4. $[\![\cdot]\!]$-*sharing:* A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be $[\![\cdot]\!]$-shared in the semi-honest setting if there exist values $\alpha_{\mathsf{a}}, \beta_{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$ such that $\beta_{\mathsf{a}} = \mathsf{a} + \alpha_{\mathsf{a}}$ where α_{a} is $[\cdot]$-shared among $\mathcal{P}$ and every

$P_i \in \mathcal{E}$ holds β_{a}. We denote the shares of $P_i \in \mathcal{D}$ by $[\![\mathsf{a}]\!]_i = [\alpha_{\mathsf{a}}]_i$ and that of $P_i \in \mathcal{E}$ as $[\![\mathsf{a}]\!]_i = (\beta_{\mathsf{a}}, [\alpha_{\mathsf{a}}]_i)$. In the malicious setting, β_{a} is held by all parties, and $[\![\mathsf{a}]\!]_i = (\beta_{\mathsf{a}}, [\alpha_{\mathsf{a}}]_i)$ for all $P_i \in \mathcal{P}$.

It is easy to see that all the sharing schemes mentioned above are linear. This allows parties to compute linear operations such as addition and multiplication with constants locally. The Boolean world operates over $\mathbb{Z}_2$, and we denote the corresponding Boolean sharing with a superscript $\mathbf{B}$.

7.3.2.1 Shared key setup

$\mathcal{F}_{\mathsf{Setup}}$ [11, 173, 193, 136] enables the establishment of common random keys for a pseudo-random function (PRF) F among parties. This aids in non-interactively generating correlated randomness. Here $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ is a secure PRF, with co-domain X being $\mathbb{Z}_{2^\ell}$. The semi-honest functionality, $\mathcal{F}_{\mathsf{Setup}}$ appears in Fig. 7.2. The functionality for the malicious case is similar, except that the adversary now has the capability to `abort`.

To sample a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ among a set of $t+1$ parties $\mathcal{T} = \{P_1, \ldots, P_{t+1}\}$ non-interactively, each $P_i \in \mathcal{T}$ invokes $F_{k_\mathcal{T}}(id_\mathcal{T})$ and obtains r. Here, $id_\mathcal{T}$ denotes a counter maintained by the parties in $\mathcal{T}$, and is updated after every PRF invocation. The appropriate keys used to sample are implicit from the context, from the identities of the parties that sample.

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

$\mathcal{F}_{\mathsf{Setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Setup}}$ picks random keys $k_\mathcal{T}, k_{\mathcal{T}'}$ for every set $\mathcal{T}, \mathcal{T}' \subseteq \mathcal{P}$ of $t+1, t+2$ parties, respectively. $\mathcal{F}_{\mathsf{Setup}}$ picks random keys k_{ij} for every pair of parties $P_i, P_j \in \mathcal{P}$ and $i < j$.

- Set $\mathsf{x}_s = \{k_{si}, k_{js}\}_{\forall i : s < i \leq n, \forall j : 1 \leq j < s}$.
- Set $\mathsf{y}_s = \{k_\mathcal{T}\}_{\forall \mathcal{T} \subset \mathcal{P} : |\mathcal{T}| = t+1}$ when $P_s \in \mathcal{T}$.
- Set $\mathsf{z}_s = \{k_{\mathcal{T}'}\}_{\forall \mathcal{T}' \subseteq \mathcal{P} | \mathcal{T}|' = t+2}$ when $P_s \in \mathcal{T}'$.

Output: Send $(\mathsf{Output}, \mathsf{x}_s, \mathsf{y}_s, \mathsf{z}_s)$ to every $P_s \in \mathcal{P}$.

Figure 7.2: Ideal functionality for shared-key setup.

7.3.3 Notations

Notations used in this chapter are summarized in Table 7.1.

Notation	Description
$n = 2t + 1$	Total number of parties with t corrupt and $h = t + 1$ honest
$\mathcal{T}_1, \ldots, \mathcal{T}_{\mathsf{q}}$	$\mathsf{q} = \binom{n}{h}$ distinct subsets of $\mathcal{P}$ with $t + 1$ parties each
q	Number of replicated secret shares (RSS) of a value
$\mathsf{g} = \binom{n-1}{h-1}$	Number of RSS shares of a value held by a party
$\mathcal{E}$	Evaluator parties $(P_1, \ldots, P_{t+1})$ that actively carry out the computation
$\mathcal{D}$	Helper parties $(P_{t+2}, \ldots, P_n)$
a_i	i^{th} element of vector $\boldsymbol{a}$
$\boldsymbol{a} \odot \boldsymbol{b}$	dot product of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
$\mathbf{A} \bigodot \mathbf{B}$	Multiplication of matrices $\mathbf{A}$ and $\mathbf{B}$
b^{R}	Arithmetic (Ring) equivalent over $\mathbb{Z}_{2^\ell}$ of bit $\mathsf{b} \in \mathbb{Z}_2$
$\mathsf{v}[i]$	i^{th} bit of ℓ-bit value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$
$\beta_{\mathsf{a}} = \mathsf{a} + \alpha_{\mathsf{a}}$	Masked value β_{a} for $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ with mask $\alpha_{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$
$\mathrm{M}_{\mathsf{a}_1\mathsf{a}_2\ldots\mathsf{a}_k}$	$\prod_{i=1}^{k} \beta_{\mathsf{a}_i}$; Product of masked values $\beta_{\mathsf{a}_1}, \ldots, \beta_{\mathsf{a}_k}$
$\Lambda_{\mathsf{a}_1\mathsf{a}_2\ldots\mathsf{a}_k}$	$\prod_{i=1}^{k} \alpha_{\mathsf{a}_i}$; Product of masks $\alpha_{\mathsf{a}_1}, \ldots, \alpha_{\mathsf{a}_k}$

Table 7.1: Notations used in this chapter.

7.3.4 Helper primitives

We use the primitives described in Table 7.2 from literature [29, 31, 193, 57] in our protocols, and their details are presented next. The Boolean variants of corresponding primitives are denoted with a superscript $\mathbf{B}$.

Primitive	Input	Output
$\Pi_{\langle 0 \rangle}$	-	$\langle\cdot\rangle$-sharing of 0
Π_{rand}	-	$[\cdot]$-sharing of a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$
Π_{pRand}	Identity of a party P_s	$[\cdot]$-sharing of a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ s.t. P_s learns all shares
$\Pi_{\cdot \to [\![\cdot]\!]}$	$\mathsf{a} \in \mathbb{Z}_{2^\ell}$ held by at least $t + 1$ parties	$[\![\mathsf{a}]\!]$-sharing
$\Pi_{[\cdot] \to {}^{\mathcal{T}}\langle\cdot\rangle}$	$[\mathsf{a}]$-sharing, $\mathcal{T} \subset \mathcal{P}$ s.t. $\lvert\mathcal{T}\rvert = t + 1$	${}^{\mathcal{T}}\langle\mathsf{a}\rangle$-sharing
$\Pi_{[\cdot] \to \langle\cdot\rangle}$	$[\mathsf{a}]$-sharing	$\langle\mathsf{a}\rangle$-sharing
$\Pi_{[\![\cdot]\!] \to {}^{\mathcal{T}}\langle\cdot\rangle}$	$[\![\mathsf{a}]\!]$-sharing, $\mathcal{T} \subset \mathcal{P}$ s.t. $\lvert\mathcal{T}\rvert = t + 1$	${}^{\mathcal{T}}\langle\mathsf{a}\rangle$-sharing
$\Pi_{[\![\cdot]\!] \to \langle\cdot\rangle}$	$[\![\mathsf{a}]\!]$-sharing	$\langle\mathsf{a}\rangle$-sharing
$\Pi_{[\![\cdot]\!] \to [\cdot]}$	$[\![\mathsf{a}]\!]$-sharing	$[\mathsf{a}]$-sharing
$\Pi_{[\cdot][\cdot] \to \langle\cdot\rangle}$	$[\mathsf{a}]$-sharing, $[\mathsf{b}]$-sharing	$\langle\mathsf{ab}\rangle$-sharing
Π_{agree}	$\mathcal{P}, \mathbf{v_1}, \ldots, \mathbf{v_n}$	'`continue`' if $\mathbf{v_i} = \mathbf{v_j}$ for all $P_i, P_j \in \mathcal{P}$, '`abort`' otherwise
$\Pi_{[\cdot]}$	private input $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ held by party P_s	$[\mathsf{a}]$-sharing

Table 7.2: Description of helper primitives – all are non-interactive, except Π_{agree}.

1. $\Pi_{\langle 0 \rangle} \to \langle 0 \rangle$ (Fig. 7.3): To generate $\langle \cdot \rangle$-shares of 0, each party non-interactively samples two values, each with one of its neighbouring parties. A party's shares of 0 are defined as the difference between these values.

Protocol $\Pi_{\langle 0 \rangle}$

1. P_i, P_{i+1}, for $i \in \{1, \ldots, n-1\}$, sample a random value $\mathsf{r}_i \in_R \mathbb{Z}_{2^\ell}$, while P_1, P_n sample a random value $\mathsf{r}_n \in_R \mathbb{Z}_{2^\ell}$, using their respective common PRF keys.
2. P_i for $i \in \{2, \ldots, n\}$ sets $\langle 0 \rangle_i = \mathsf{r}_i - \mathsf{r}_{i-1}$, while P_1 sets $\langle 0 \rangle_1 = \mathsf{r}_1 - \mathsf{r}_n$.

Figure 7.3: Generating $\langle \cdot \rangle$-shares of 0.

2. $\Pi_{\mathsf{rand}} \to [\mathsf{r}]$ (Fig. 7.4): To generate $[\cdot]$-shares of a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, every set of $t+1$ parties non-interactively sample a random value using keys established during the setup phase and define r to be the sum of these values.

Protocol Π_{rand}

1. Every $P_i \in \mathcal{T}_j$ for $j \in \{1, \ldots, \mathsf{q}\}$, samples $[\mathsf{r}]_{\mathcal{T}_j} \in_R \mathbb{Z}_{2^\ell}$ using the common PRF key.
2. Define $\mathsf{r} = \sum_{j=1}^{\mathsf{q}} [\mathsf{r}]_{\mathcal{T}_j}$.

Figure 7.4: Generating $[\cdot]$-shares of a random value.

3. $\Pi_{\mathsf{pRand}}(P_s) \to [\mathsf{r}]$ (Fig. 7.5): This protocol generates $[\cdot]$-shares of a random value r such that P_s learns all the shares. Every set of $t+1$ parties non-interactively samples a random value together with P_s, using the keys established (for every set of $t+2$ parties) during the setup phase.

Protocol $\Pi_{\mathsf{pRand}}(P_s)$

1. Every $P_i \in \mathcal{T}_j$ for $j \in \{1, \ldots, \mathsf{q}\}$, samples $[\mathsf{r}]_{\mathcal{T}_j} \in_R \mathbb{Z}_{2^\ell}$, together with P_s, using the common PRF key.
2. Define $\mathsf{r} = \sum_{j=1}^{\mathsf{q}} [\mathsf{r}]_{\mathcal{T}_j}$.

Figure 7.5: Generating $[\cdot]$-shares of a random value along with P_s.

4. $\Pi_{\to [\![\cdot]\!]}(\mathsf{a}) \to [\![\mathsf{a}]\!]$: This protocol generates $[\![\mathsf{a}]\!]$ when $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is held by at least $t+1$ parties, say parties in $\mathcal{E}$. For this, $P_i \in \mathcal{E}$ sets $\beta_\mathsf{a} = \mathsf{a}$ and $[\cdot]$-shares of α_a as 0. To generate $[\![\mathsf{a}]\!]$ in the malicious case where all parties hold a, parties set $\beta_\mathsf{a} = \mathsf{a}$ and shares of α_a as 0.
5. $\Pi_{[\cdot] \to {}^{\mathcal{T}}\langle \cdot \rangle}([\mathsf{a}]) \to {}^{\mathcal{T}}\langle \mathsf{a} \rangle$ (Fig. 7.6): This protocol enables parties in $\mathcal{T} = \{E_1, E_2, \ldots, E_{t+1}\}$ to generate ${}^{\mathcal{T}}\langle \mathsf{a} \rangle$ from $[\mathsf{a}]$. To generate ${}^{\mathcal{T}}\langle \mathsf{a} \rangle_i$, the idea is to sum up the shares in $[\mathsf{a}]_{\mathcal{T}_1}, \ldots, [\mathsf{a}]_{\mathcal{T}_\mathsf{q}}$, while ensuring that every share is accounted for and no share is incorporated more than once. Concretely, for share $[\mathsf{a}]_{\mathcal{T}_j}$ held by parties in $\mathcal{T}_j$ for $j \in \{1, \ldots, \mathsf{q}\}$, $E_i \in \mathcal{T}_j$ incorpo-

rates $[\mathsf{a}]_{\mathcal{T}_j}$ in its share of ${}^{\mathcal{T}}\langle \mathsf{a} \rangle_i$ if E_i has the least index in $\mathcal{T}_j$.

Protocol $\Pi_{[\cdot]\to{}^{\mathcal{T}}\langle\cdot\rangle}([\mathsf{a}])$

1. Let $\mathcal{T} = \{E_1, \ldots, E_{t+1}\}$.
2. $E_i \in \mathcal{T}$ computes ${}^{\mathcal{T}}\langle \mathsf{a} \rangle_i = \sum_{j=1}^{\mathsf{q}} [\mathsf{a}]_{\mathcal{T}_j} \cdot \mathsf{e}_j^i$, where $\mathsf{e}_j^i = 1$ if E_i has the least index in $\mathcal{T}_j$, and 0, otherwise.

Figure 7.6: Conversion from $[\cdot]$-share to ${}^{\mathcal{T}}\langle\cdot\rangle$-share.

6. $\Pi_{[\cdot]\to\langle\cdot\rangle}([\mathsf{a}]) \to \langle \mathsf{a} \rangle$: $[\cdot]$-share can be converted to $\langle\cdot\rangle$-share following similar procedure as $\Pi_{[\cdot]\to{}^{\mathcal{T}}\langle\cdot\rangle}$, and is denoted as $\Pi_{[\cdot]\to\langle\cdot\rangle}([\mathsf{a}])$. We omit the details due to similarity.
7. $\Pi_{[\![\cdot]\!]\to{}^{\mathcal{T}}\langle\cdot\rangle}([\![\mathsf{a}]\!]) \to {}^{\mathcal{T}}\langle \mathsf{a} \rangle$: Parties in $\mathcal{T}$ invoke $\Pi_{[\cdot]\to{}^{\mathcal{T}}\langle\cdot\rangle}$ on $-\alpha_{\mathsf{a}}$ to generate ${}^{\mathcal{T}}\langle -\alpha_{\mathsf{a}} \rangle$, followed by a designated $P_i \in \mathcal{T}$ that holds β_{a} setting ${}^{\mathcal{T}}\langle \mathsf{a} \rangle_i = \beta_{\mathsf{a}} + {}^{\mathcal{T}}\langle -\alpha_{\mathsf{a}} \rangle_i$.
8. $\Pi_{[\![\cdot]\!]\to\langle\cdot\rangle}([\![\mathsf{a}]\!]) \to \langle \mathsf{a} \rangle$: $\langle \mathsf{a} \rangle$ can be generated from $[\![\mathsf{a}]\!]$ similar to $\Pi_{[\![\cdot]\!]\to{}^{\mathcal{T}}\langle\cdot\rangle}$, and is denoted as $\Pi_{[\![\cdot]\!]\to\langle\cdot\rangle}([\![\mathsf{a}]\!])$.
9. $\Pi_{[\cdot]\to[\![\cdot]\!]}([\mathsf{a}]) \to [\![\mathsf{a}]\!]$: To convert $[\mathsf{a}]$ to $[\![\mathsf{a}]\!]$, set $\beta_{\mathsf{a}} = 0$ and set $[\alpha_{\mathsf{a}}] = -[\mathsf{a}]$.
10. $\Pi_{[\![\cdot]\!]\to[\cdot]}([\![\mathsf{a}]\!]) \to [\mathsf{a}]$: To convert $[\![\mathsf{a}]\!]$ to $[\mathsf{a}]$, set $[\mathsf{a}]_{\mathcal{T}_j} = -[\alpha_{\mathsf{a}}]_{\mathcal{T}_j}$ for $j \in \{1, \ldots, \mathsf{q}-1\}$ and $[\mathsf{a}]_{\mathcal{T}_{\mathsf{q}}} = \beta_{\mathsf{a}} - [\alpha_{\mathsf{a}}]_{\mathcal{T}_{\mathsf{q}}}$, where $\mathcal{T}_{\mathsf{q}} = \mathcal{E}$.
11. $\Pi_{[\cdot][\cdot]\to\langle\cdot\rangle}([\mathsf{a}], [\mathsf{b}]) \to \langle \mathsf{ab} \rangle$ (Fig. 7.7): Given $[\mathsf{a}], [\mathsf{b}]$, parties non-interactively compute $\langle \mathsf{ab} \rangle$ as follows. Observe that $\langle \mathsf{ab} \rangle = \sum_{j=1}^{\mathsf{q}} \langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle$. To generate $\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle$, the idea is to generate ${}^{\mathcal{T}_j}\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle$ and perform a conversion. Parties in $\mathcal{T}_j$ generate ${}^{\mathcal{T}_j}\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle$ as ${}^{\mathcal{T}_j}\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle = \left([\mathsf{a}]_{\mathcal{T}_j}\right) \cdot \left({}^{\mathcal{T}_j}\langle \mathsf{b} \rangle\right)$. To obtain $\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle$ from ${}^{\mathcal{T}_j}\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle$, $P_i \in \mathcal{P}$ sets $\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle_i = {}^{\mathcal{T}_j}\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle_i$ if $P_i \in \mathcal{T}_j$ and $\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle_i = 0$, otherwise.

Protocol $\Pi_{[\cdot][\cdot]\to\langle\cdot\rangle}(\mathcal{P}, [\mathsf{a}], [\mathsf{b}])$

1. For $j \in \{1, \ldots, \mathsf{q}\}$:
   - $P_i \in \mathcal{T}_j$ invokes $\Pi_{[\cdot]\to{}^{\mathcal{T}}\langle\cdot\rangle}$ on $[\mathsf{b}]$ to generate ${}^{\mathcal{T}_j}\langle \mathsf{b} \rangle_i$.
   - Set $\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle_i = \left([\mathsf{a}]_{\mathcal{T}_j}\right) \cdot \left({}^{\mathcal{T}_j}\langle \mathsf{b} \rangle_i\right)$ if $P_i \in \mathcal{T}_j$, and $\langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle_i = 0$, otherwise.
2. $P_i \in \mathcal{P}$ computes $\langle \mathsf{ab} \rangle_i = \sum_{j=1}^{\mathsf{q}} \langle [\mathsf{a}]_{\mathcal{T}_j} \mathsf{b} \rangle_i$.

Figure 7.7: $[\mathsf{a}], [\mathsf{b}]$ to $\langle \mathsf{ab} \rangle$.

12. $\Pi_{\mathsf{agree}}(\mathcal{P}, \{\mathbf{v_1}, \ldots, \mathbf{v_n}\}) \to \mathsf{continue}/\mathtt{abort}$: Allows parties to check if they hold the same set of values $\mathbf{v} = (\mathsf{v}_1, \ldots, \mathsf{v}_m)$, where parties $\mathsf{continue}$ if the values are same, and $\mathtt{abort}$ otherwise. We denote the version of $\mathbf{v}$ held by $P_i \in \mathcal{P}$ as $\mathbf{v_i}$. To check for consistency of $\mathbf{v}$, parties compute hash, $\mathsf{H} = \mathsf{H}(\mathsf{v}_1 || \ldots || \mathsf{v}_m)$, of the concatenation of all values $\mathsf{v}_1, \ldots, \mathsf{v}_m$, and exchange H among themselves. If any party receives inconsistent hashes, it $\mathtt{aborts}$;

else, it continues.

13. $\Pi_{[\cdot]}(P_s, \mathsf{a}) \to [\mathsf{a}]$: To enable P_s to generate $[\mathsf{a}]$, parties generate $[\mathsf{a}]_{\mathcal{T}_j}$ for $j \in \{1, \ldots, \mathsf{q}-1\}$ using Π_{pRand}, with P_s learning $[\mathsf{a}]_{\mathcal{T}_j}$ (i.e., $[\mathsf{a}]_{\mathcal{T}_j}$ are sampled using common key amongst $t+2$ parties). P_s sets $[\mathsf{a}]_{\mathcal{T}_\mathsf{q}} = \mathsf{a} - \sum_{j=1}^{\mathsf{q}-1} [\mathsf{a}]_{\mathcal{T}_j}$ and sends $[\mathsf{a}]_{\mathcal{T}_\mathsf{q}}$ to parties in $\mathcal{T}_\mathsf{q}$. For malicious case, this is followed by invoking $\Pi_{\mathsf{agree}}(\mathcal{P}, \{[\mathsf{a}]_{\mathcal{T}_\mathsf{q}}\})$ to check consistency of values sent by P_s.

7.4 Semi-honest protocol

The ideal functionality $\mathcal{F}_f$ for evaluating function f in the n-party setting with semi-honest security appears in Fig. 7.8. Details of its instantiation over the ring $\mathbb{Z}_{2^\ell}$ that comprises three phases—input sharing, evaluation (linear operations and multiplication), and output reconstruction—appear next.

Functionality $\mathcal{F}_f$

$\mathcal{F}_f$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}^{\mathsf{sh}}$. Let f denote the function to be computed. Let x_s be the input corresponding to the party P_s, and y_s be the corresponding output, i.e $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$.

Step 1: $\mathcal{F}_f$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$, and computes $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$.

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

Figure 7.8: Semi-honest: Ideal functionality for function f.

7.4.1 Input sharing and output reconstruction

To enable $P_s \in \mathcal{P}$ to $[\![\cdot]\!]$-share a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, parties first non-interactively sample $[\cdot]$-shares of α_v, relying on the shared-key setup, such that P_s learns all these shares in clear (via Π_{pRand}). This enables P_s to compute and send $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ to parties in $\mathcal{E}$, thereby generating $[\![\mathsf{v}]\!]$. The protocol for input sharing appears in Fig. 7.9.

Protocol $\Pi_{\mathsf{Sh}}(P_s, \mathsf{a})$

Preprocessing: Invoke $\Pi_{\mathsf{pRand}}(P_s)$ to generate $[\alpha_\mathsf{a}]$, with P_s learning $\alpha_\mathsf{a} \in \mathbb{Z}_{2^\ell}$.

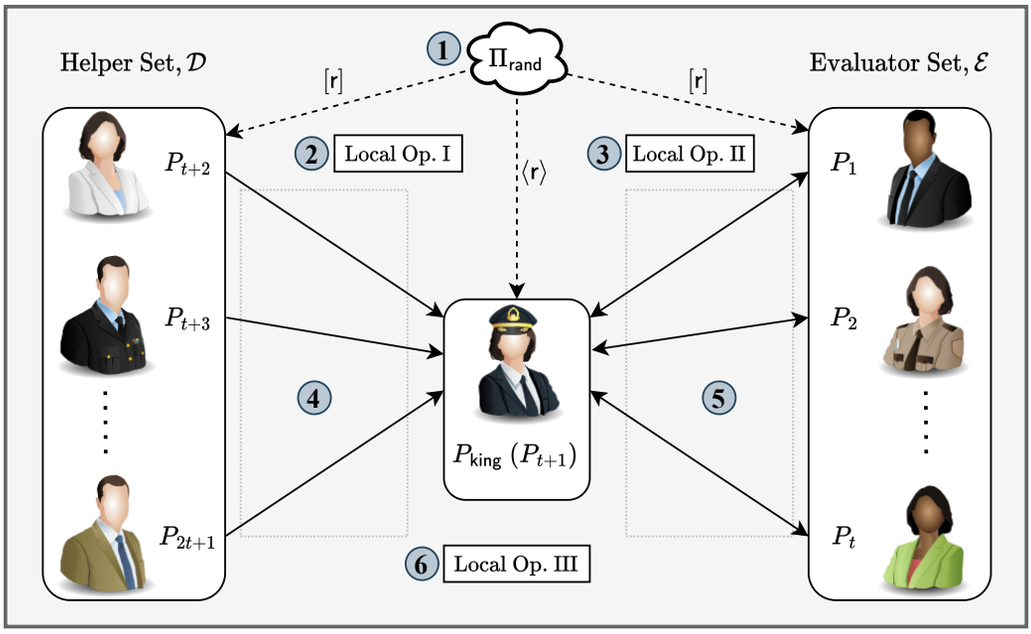
Online: P_s computes and sends $\beta_\mathsf{a} = \mathsf{a} + \alpha_\mathsf{a}$ to all $P_i \in \mathcal{E}$.

Figure 7.9: Semi-honest: Input sharing protocol.

To reconstruct v towards all parties given $[\![\mathsf{v}]\!]$, observe that parties in $\mathcal{E}$ possess sufficient shares to facilitate the same. Elaborately, parties in $\mathcal{E}$ can non-interactively generate additive shares, ${}^{\mathcal{E}}\langle \mathsf{v} \rangle$, among themselves (via $\Pi_{[\![\cdot]\!] \to {}^{\mathcal{E}}\langle\cdot\rangle}$). These parties can then send their additive shares to P_{king}, who computes and sends v to all parties. Reconstruction towards a single party, say P_s, can proceed similarly except that the protocol terminates after parties in $\mathcal{E}$ send their additive shares of v to $P_{\mathsf{king}} = P_s$, who then computes v.

7.4.2 Evaluation

The evaluation comprises linear operations of addition and multiplication with public constant and non-linear operations such as multiplication. Parties can non-interactively compute linear operations owing to the linearity of the $[\![\cdot]\!]$-sharing. Concretely, given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$ and public constants $\mathsf{c}_1, \mathsf{c}_2$, parties can non-interactively compute $[\![\mathsf{c}_1\mathsf{a} + \mathsf{c}_2\mathsf{b}]\!]$ as $\mathsf{c}_1[\![\mathsf{a}]\!] + \mathsf{c}_2[\![\mathsf{b}]\!]$.



① Generation of random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ ② Computing $\langle \mathsf{r} \rangle \& [\![\mathsf{r}]\!]$ ③ Computing ${}^{\mathcal{E}}\langle \alpha_{\mathsf{a}} \rangle, {}^{\mathcal{E}}\langle \alpha_{\mathsf{b}} \rangle$ ④ $\mathcal{D}$ sending $\{\langle \mathsf{z} - \mathsf{r} \rangle\}_{\mathcal{D}}$ to P_{king} ⑤ $\mathcal{E}$ sending $\{\langle \mathsf{z} - \mathsf{r} \rangle\}_{\mathcal{E}}$ to P_{king} and receiving result from P_{king} ⑥ Computing $[\![\mathsf{z}]\!]$

Figure 7.10: Steps of semi-honest multiplication protocol.

To compute $[\![\cdot]\!]$-shares for non-linear operations such as multiplication, say $\mathsf{z} = \mathsf{ab}$ given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$, parties proceed as follows. At a high level, the approach is to enable the generation of $[\![\mathsf{z} - \mathsf{r}]\!]$ and $[\![\mathsf{r}]\!]$ for a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, which enables parties to non-interactively compute $[\![\mathsf{z}]\!] = [\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$. Observe that $[\![\mathsf{r}]\!]$ can be generated non-interactively by locally sampling

each of its shares. To generate $[\![\mathsf{z}-\mathsf{r}]\!]$, we let parties in $\mathcal{E}$ obtain $\mathsf{z}-\mathsf{r}$, following which $[\![\mathsf{z}-\mathsf{r}]\!]$ can be generated non-interactively (this is achieved via $\Pi_{\cdot\to[\![\cdot]\!]}$ where all parties set their shares of $[\alpha_{\mathsf{z}-\mathsf{r}}]$ as 0, and parties in $\mathcal{E}$ set $\beta_{\mathsf{z}-\mathsf{r}} = \mathsf{z}-\mathsf{r}$). Observe that z remains private while revealing $\mathsf{z}-\mathsf{r}$ to parties in $\mathcal{E}$ since r is a random mask not known to the adversary.

To enable parties in $\mathcal{E}$ to obtain $\mathsf{z}-\mathsf{r}$, we let $\mathsf{z}-\mathsf{r} = \mathsf{D}+\mathsf{E}$, where D is additively shared among parties in $\mathcal{D}$ while E is additively shared among parties in $\mathcal{E}$ (D, E are defined in the following paragraphs). Thus, to reconstruct $\mathsf{z}-\mathsf{r}$ towards parties in $\mathcal{E}$, parties send their respective additive shares of D or E towards $P_{\mathsf{king}} \in \mathcal{P}$. P_{king} reconstructs D, E, and sends $\mathsf{z}-\mathsf{r} = \mathsf{D}+\mathsf{E}$ to parties in $\mathcal{E}$. Elaborately, as seen in [49, 138], $\mathsf{z}-\mathsf{r}$ can be computed as

$$\begin{aligned}\mathsf{z}-\mathsf{r} &= \mathsf{ab}-\mathsf{r} = (\beta_{\mathsf{a}}-\alpha_{\mathsf{a}})(\beta_{\mathsf{b}}-\alpha_{\mathsf{b}}) - \mathsf{r} = \mathrm{M}_{\mathsf{ab}} - \beta_{\mathsf{a}}\alpha_{\mathsf{b}} - \beta_{\mathsf{b}}\alpha_{\mathsf{a}} + \Lambda_{\mathsf{ab}} - \mathsf{r} \\ &= \underbrace{\mathrm{M}_{\mathsf{ab}} - \beta_{\mathsf{a}}\alpha_{\mathsf{b}} - \beta_{\mathsf{b}}\alpha_{\mathsf{a}} + (\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{E}}}_{\mathsf{E}} + \underbrace{(\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{D}}}_{\mathsf{D}}\end{aligned} \tag{7.1}$$

where $\Lambda_{\mathsf{ab}}-\mathsf{r} = (\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{D}} + (\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{E}}$. The multiplication protocol Π_{Mul} (Fig. 7.11) is detailed next, and its schematic representation is provided in Fig. 7.10.

– *Step* ①: Parties non-interactively generate $[\mathsf{r}]$ by locally sampling each of its shares (via Π_{rand}). Parties locally compute $\langle\mathsf{r}\rangle$ and $[\![\mathsf{r}]\!]$ from $[\mathsf{r}]$ using $\Pi_{[\cdot]\to\langle\cdot\rangle}$ and $\Pi_{[\cdot]\to[\![\cdot]\!]}$, respectively. Looking ahead, $\langle\mathsf{r}\rangle$ aids in generating additive shares of D, E, while $[\![\mathsf{r}]\!]$ aids in computing $[\![\mathsf{z}]\!]$ from $[\![\mathsf{z}-\mathsf{r}]\!]$.

– *Step* ②: This step involves computing additive shares of $\Lambda_{\mathsf{ab}}-\mathsf{r}$ among all parties. For this, parties non-interactively generate $\langle\Lambda_{\mathsf{ab}}\rangle$ from $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ (via $\Pi_{[\cdot],[\cdot]\to\langle\cdot\rangle}$). $P_i \in \mathcal{P}$ sets its additive share of $\Lambda_{\mathsf{ab}}-\mathsf{r}$ as $\langle\Lambda_{\mathsf{ab}}-\mathsf{r}\rangle_i = \langle\Lambda_{\mathsf{ab}}\rangle_i - \langle\mathsf{r}\rangle_i$. Observe that the shares $\langle\Lambda_{\mathsf{ab}}-\mathsf{r}\rangle_i$ of $P_i \in \mathcal{D}$ define the additive shares of $\mathsf{D} = (\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{D}}$ among parties in $\mathcal{D}$. Similarly, the shares $\langle\Lambda_{\mathsf{ab}}-\mathsf{r}\rangle_i$ of $P_i \in \mathcal{E}$ define the additive shares of $(\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{E}}$ among parties in $\mathcal{E}$ (i.e. ${}^{\mathcal{E}}\langle(\Lambda_{\mathsf{ab}}-r)_{\mathcal{E}}\rangle$).

– *Step* ③: Parties in $\mathcal{E}$ generate additive shares of $\alpha_{\mathsf{a}}, \alpha_{\mathsf{b}}$ among themselves (${}^{\mathcal{E}}\langle\cdot\rangle$-shares, via $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$). Looking ahead, ${}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle, {}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle$ aid in generating additive shares of E among $\mathcal{E}$.

– *Step* ④: Parties in $\mathcal{D}$ send their additive shares of D (as defined in step ②) to P_{king}, where the latter reconstructs D.

– *Step* ⑤: $P_i \in \mathcal{E} \setminus \{P_{\mathsf{king}}\}$ non-interactively generates additive share, ${}^{\mathcal{E}}\langle\mathsf{E}\rangle_i$, of E among parties in $\mathcal{E}$ as ${}^{\mathcal{E}}\langle\mathsf{E}\rangle_i = -\beta_{\mathsf{a}}{}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle_i - \beta_{\mathsf{b}}{}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_i + {}^{\mathcal{E}}\langle(\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{E}}\rangle_i$. Note that it suffices for only one designated party in $\mathcal{E}$ to add M_{ab} in its share of ${}^{\mathcal{E}}\langle\mathsf{E}\rangle$, and without loss of generality we let this designated party be P_{king}. For $P_{\mathsf{king}} = P_{t+1}$ in our case, ${}^{\mathcal{E}}\langle\mathsf{E}\rangle_{t+1} = \mathrm{M}_{\mathsf{ab}} - \beta_{\mathsf{a}}{}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle_{t+1} - \beta_{\mathsf{b}}{}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_{t+1} + {}^{\mathcal{E}}\langle(\Lambda_{\mathsf{ab}}-\mathsf{r})_{\mathcal{E}}\rangle_{t+1}$. Parties send their additive shares of E to P_{king}, who reconstructs E, and sends $\mathsf{z}-\mathsf{r} = \mathsf{D}+\mathsf{E}$ to parties in $\mathcal{E}$.

– *Step* ⑥: Parties non-interactively generate $[\![\mathsf{z}-\mathsf{r}]\!]$ (via $\Pi_{\cdot\to[\![\cdot]\!]}$) as explained earlier. Using $[\![\mathsf{r}]\!]$ generated in step ①, parties compute $[\![\mathsf{z}]\!] = [\![\mathsf{z}-\mathsf{r}]\!] + [\![\mathsf{r}]\!]$, as required.

Protocol $\Pi_{\mathsf{Mul}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!], \mathsf{isTr})$

$\mathsf{isTr} = 1$ denotes perform truncation, $\mathsf{isTr} = 0$ denotes otherwise.

Preprocessing:

① If $\mathsf{isTr} = 0$: invoke Π_{rand} to generate $[\mathsf{r}]$ where $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. Invoke $\Pi_{[\cdot]\to\langle\cdot\rangle}$ and $\Pi_{[\cdot]\to[\![\cdot]\!]}$ on $[\mathsf{r}]$ to generate $\langle\mathsf{r}\rangle$ and $[\![\mathsf{r}]\!]$, respectively.

– Else, invoke $\Pi_{\mathsf{dsBits}}(\mathcal{P}, 1)$ (Fig. 7.14) to generate $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$, and $\Pi_{[\![\cdot]\!]\to\langle\cdot\rangle}$ on $[\![\mathsf{r}]\!]$ to generate $\langle\mathsf{r}\rangle$.

② Invoke $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$ on $[\alpha_\mathsf{a}], [\alpha_\mathsf{b}]$ to generate $\langle\Lambda_{\mathsf{ab}}\rangle$, and compute $\langle\Lambda_{\mathsf{ab}} - \mathsf{r}\rangle = \langle\Lambda_{\mathsf{ab}}\rangle - \langle\mathsf{r}\rangle$. $P_i \in \mathcal{E}$ sets ${}^{\mathcal{E}}\langle(\Lambda_{\mathsf{ab}} - r)_{\mathcal{E}}\rangle_i = \langle\Lambda_{\mathsf{ab}} - r\rangle_i$.

③ $P_i \in \mathcal{E}$ invokes $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$ on $[\alpha_\mathsf{a}], [\alpha_\mathsf{b}]$ to generate ${}^{\mathcal{E}}\langle\alpha_\mathsf{a}\rangle_i, {}^{\mathcal{E}}\langle\alpha_\mathsf{b}\rangle_i$, respectively.

④ $P_i \in \mathcal{D}$ sends $\langle\Lambda_{\mathsf{ab}} - \mathsf{r}\rangle_i$ to P_{king}, who sets $\mathsf{D} = \sum_{i:P_i\in\mathcal{D}}\langle\Lambda_{\mathsf{ab}} - \mathsf{r}\rangle_i$.

Online:

⑤ $P_i \in \mathcal{E}$ computes ${}^{\mathcal{E}}\langle\zeta\rangle_i = -\beta_\mathsf{a}\,{}^{\mathcal{E}}\langle\alpha_\mathsf{b}\rangle_i - \beta_\mathsf{b}\,{}^{\mathcal{E}}\langle\alpha_\mathsf{a}\rangle_i + {}^{\mathcal{E}}\langle(\Lambda_{\mathsf{ab}} - \mathsf{r})_{\mathcal{E}}\rangle_i$, and sends ${}^{\mathcal{E}}\langle\zeta\rangle_i$ to P_{king}. P_{king} computes $\mathsf{E} = \mathrm{M}_{\mathsf{ab}} + \sum_{i:P_i\in\mathcal{E}}{}^{\mathcal{E}}\langle\zeta\rangle_i$ and sends $\mathsf{z} - \mathsf{r} = \mathsf{D} + \mathsf{E}$ to all parties in $\mathcal{E}$.

⑥ If $\mathsf{isTr} = 0$: invoke $\Pi_{\cdot\to[\![\cdot]\!]}$ on $\mathsf{z} - \mathsf{r}$ to generate $[\![\mathsf{z} - \mathsf{r}]\!]$, and compute $[\![\mathsf{z}]\!] = [\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$.

– Else, invoke $\Pi_{\cdot\to[\![\cdot]\!]}$ on $(\mathsf{z} - \mathsf{r})^d$ to generate $[\![(\mathsf{z} - \mathsf{r})^d]\!]$, and compute $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z} - \mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$.

Figure 7.11: Semi-honest: Multiplication protocol.

Lemma 7.1 *Protocol* Π_{Mul} *(Fig. 7.11) incurs a communication of* t *elements in the preprocessing phase and* $2t$ *elements in* 2 *rounds in the online phase for multiplication when* $\mathsf{isTr} = 0$.

Analysis: Observe that the communication towards P_{king} in steps ④ and ⑤, can be performed in parallel, resulting in the overall round complexity of the protocol being two. Further, communication of t elements is required in step ④, and $2t$ elements are required in ⑤ (since $P_{\mathsf{king}} \in \mathcal{E}$), thereby having a total communication complexity of $3t$ ring elements. This complexity resembles that of DN07*. However, our sharing semantics enables us to push some of the steps mentioned above to a preprocessing phase, resulting in a fast online phase, which is non-trivial to achieve in the case of DN07*. Elaborately, observe that since $\mathsf{r}, \alpha_\mathsf{a}, \alpha_\mathsf{b}$ are independent of the input (owing to our sharing semantics), computation involving these terms ranging from steps ① to ④ can thus be moved to a preprocessing phase. This improves the online communication complexity by slashing the inward communication towards P_{king} by half. Thus, the online phase

requires only $2t$ ring elements of communication while offloading t elements of communication to the preprocessing phase.

Note that a straightforward extension of the semi-honest multiplication of DN07* to the preprocessing model, which can be derived from [78], does not provide an efficient solution. Although such a protocol has the same online complexity ($2t$ elements) as our online phase, it has the drawback of inflating the overall communication cost by a factor of 1.6× over DN07*. Elaborately, the online communication cost of $2t$ elements can be attained by appropriately defining the sharing semantics and using the P_{king} approach, similar to our protocol. However, this requires parties to generate the sharing of $\Lambda_{\mathsf{ab}} = \alpha_{\mathsf{a}} \cdot \alpha_{\mathsf{b}}$ from the shares of α_{a} and α_{b} during the preprocessing phase, and requires a full-fledged multiplication, incurring a cost of $3t$ elements. This yields a protocol with a total cost of $5t$ elements in comparison to the $3t$ cost of the all-online DN07* protocol. Thus, departing from this approach, the novelty of our protocol lies in leveraging the interplay between the sharing semantics and redesigning the communication pattern among the parties to ensure that the total cost of $3t$ does not change.

Furthermore, our protocol design allows parties in $\mathcal{D}$ to remain shut in the online phase, thereby reducing the system's operational load. This is because parties in $\mathcal{D}$ only contribute towards the computation of D, which can be completed in the preprocessing phase. However, the preprocessing phase becomes function-dependent due to the linear gates, for which the α value for the output wires cannot be chosen randomly. Concretely, if c is the output of a linear gate, say addition, with inputs a, b, then α_{c} cannot be chosen randomly and should be defined as $\alpha_{\mathsf{c}} = \alpha_{\mathsf{a}} + \alpha_{\mathsf{b}}$.

Online-only mode: Note that in instances where the function description is not known beforehand, our protocol can be run as an all-online protocol with a cost matching that of DN07*. There are two approaches in which this can be achieved. The first approach is as described in steps ①–⑥ discussed above, with the communication towards P_{king} in steps ④ and ⑤ executed simultaneously. The second approach is to begin by performing steps comprising the preprocessing phase, followed by the online phase steps. Although the second approach requires an additional round in the beginning to perform the preprocessing steps, it has the advantage that after completing the preprocessing phase, parties in $\mathcal{D}$ can be shut down. This helps in improving the operational cost of the system.

7.4.3 The complete MPC protocol

The complete semi-honest secure MPC protocol, $\Pi^{\mathsf{sh}}_{\mathsf{MPC}}$, evaluating a function $f(\cdot)$ appears in Fig. 7.12.

Protocol $\Pi^{\mathsf{sh}}_{\mathsf{MPC}}(\mathcal{P}, f(\cdot))$

Let $f(\cdot)$ denote the function to be computed, which is represented as a circuit with linear gates (with inputs $\mathsf{a}, \mathsf{b} \in \mathbb{Z}_{2^\ell}$ and output $\mathsf{c} = c_1\mathsf{a} + c_2\mathsf{b}$, for constants $c_1, c_2 \in \mathbb{Z}_{2^\ell}$) and multiplication gates (with inputs $\mathsf{a}, \mathsf{b} \in \mathbb{Z}_{2^\ell}$, and output $\mathsf{c} = \mathsf{a} \cdot \mathsf{b}$). The circuit evaluation proceeds by evaluating the gates in a predetermined topological order. $\mathsf{isTr} = 1$ denotes perform truncation, $\mathsf{isTr} = 0$ denotes otherwise.

Preprocessing:

1. For each circuit input a held by P_s, parties execute the preprocessing steps of $\Pi_{\mathsf{Sh}}(P_s, \mathsf{a})$.
2. For each linear gate with input wires a, b, output c, and constants c_1, c_2, parties locally compute $[\alpha_\mathsf{c}] = c_1 [\alpha_\mathsf{a}] + c_2 [\alpha_\mathsf{b}]$.
3. For each multiplication gate with input wires a, b and output c, parties execute the preprocessing steps of $\Pi_{\mathsf{Mul}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!], \mathsf{isTr})$.

Online:

1. For each input wire, parties execute the online steps of $\Pi_{\mathsf{Sh}}(P_s, \mathsf{a})$, where P_s is the party designated to provide the input a.
2. For each linear gate with input wires a, b, output c, and constants c_1, c_2, parties locally compute $\beta_\mathsf{c} = c_1\beta_\mathsf{a} + c_2\beta_\mathsf{b}$.
3. For each multiplication gate with input wires a, b and output c, parties execute the online steps of $\Pi_{\mathsf{Mul}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!], \mathsf{isTr})$.
4. For each output wire a in the circuit, parties execute the steps for reconstruction towards each party P_s which is designated to receive the corresponding output.

Figure 7.12: Semi-honest: The complete MPC protocol.

7.4.4 Incorporating truncation

To deal with decimal values that arise in several applications, including the ones considered in this work, we operate on the fixed-point arithmetic (FPA) representation [44, 45], as described in §7.3. In this case, performing multiplication, $\mathsf{z} = \mathsf{ab}$, results in increasing the number of fractional bits in the result of multiplication, z, from d to $2\mathsf{d}$. To retain FPA semantics, it is required to truncate z by d bits, i.e. compute $\mathsf{z}^d = \mathsf{z}/2^\mathsf{d}$. For this, we extend the *probabilistic* truncation technique of [173, 136, 138] proposed in the small party domain to the n-party setting. Given $(\mathsf{r}, \mathsf{r}^d)$-pair, with $\mathsf{r}^d = \mathsf{r}/2^\mathsf{d}$, the truncated value of z can be obtained as $\mathsf{z}^d = (\mathsf{z} - \mathsf{r})^d + \mathsf{r}^d$. The accuracy and correctness of this method follow from [173, 169].

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

- Samples random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, and computes $\mathsf{r}^d = \mathsf{r}/2^{\mathsf{d}}$.
- Generates $[\![\cdot]\!]$-shares of r, r^d and set output share for $P_s \in \mathcal{P}$ as $\mathsf{y}_s = \{[\![\mathsf{r}]\!]_s, [\![\mathsf{r}^d]\!]_s\}$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$.

Figure 7.13: Ideal functionality $\mathcal{F}_{\mathsf{TrGen}}$.

Our multiplication protocol can be modified to additionally perform truncation by incorporating the following two changes—(i) generate $[\![\mathsf{r}^d]\!]$ in step ①, and (ii) compute $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z}-\mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$, instead, in step ⑥. For (i), we rely on the ideal functionality, $\mathcal{F}_{\mathsf{TrGen}}$ (Fig. 7.13), for computing $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$. In our work, we instantiate $\mathcal{F}_{\mathsf{TrGen}}$ using Π_{dsBits} (Fig. 7.14), which is a slightly modified version of the doubly-shared random bit generation protocol of [67], adapted to our n-party setting. Concretely, Π_{dsBits} generates ℓ doubly-shared random bits instead of a single bit, as done in the protocol of [67]. Here, a doubly-shared random bit is a bit which is arithmetic as well as Boolean shared. With respect to (ii), observe that it is a local operation, and hence performing truncation does not incur any additional overhead in the online phase. The details of Π_{dsBits}, which follow from the protocol of [67], are presented next.

Truncation - Instantiating $\mathcal{F}_{\mathsf{TrGen}}$ We rely on a modified version of the doubly shared random bit (a bit that is arithmetic as well as Boolean shared) generation protocol of [67], extended to our n-party setting, to generate $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$ as required to perform truncation. The resulting protocol is referred to as Π_{dsBits} (Fig. 7.14).

Protocol $\Pi_{\mathsf{dsBits}}(\mathcal{P}, \mathsf{isTr})$

If $\mathsf{isTr} = 1$, set $\mathsf{k} = \ell$ else set $\mathsf{k} = 1$. For $i \in \{0, \ldots, \mathsf{k}-1\}$:

1. Invoke Π_{rand} to generate $[\mathsf{u}_i]^{\ell+2}$ for $\mathsf{u}_i \in \mathbb{Z}_{2^{\ell+2}}$, and $\Pi_{\langle 0 \rangle}$ to generate $\langle 0 \rangle^{\ell+2}$.
2. Compute $[\mathsf{a}_i]^{\ell+2} = 2\,[\mathsf{u}_i]^{\ell+2} + 1$.
3. Invoke $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$ on $[\mathsf{a}_i]^{\ell+2}$ to generate $\langle \mathsf{e}_i \rangle^{\ell+2}$ where $\mathsf{e}_i = \mathsf{a}_i^2$.
4. Send $\langle \mathsf{e}_i \rangle^{\ell+2} + \langle 0 \rangle^{\ell+2}$ to P_{king}, who reconstructs $\mathsf{e}_i + 0 = \mathsf{e}_i$ and sends to all.
5. Let c_i be smallest root of e_i modulo $2^{\ell+2}$, and c_i^{-1} its inverse. Compute $[\mathsf{d}_i]^{\ell+2} = \mathsf{c}_i^{-1}\,[\mathsf{a}_i]^{\ell+2} + 1$.
6. P_j sets $[\mathsf{b}_i]_j^{\ell+2} = [\mathsf{d}_i]_j^{\ell+2}/2$, and $\left[\mathsf{b}_i^{\mathsf{R}}\right]_j, [\mathsf{b}_i]_j^{\mathbf{B}}$ as the least significant ℓ bits and the least significant bit of $[\mathsf{b}_i]_j^{\ell+2}$, respectively.
7. Invoke $\Pi_{[\cdot]\to[\![\cdot]\!]}$ on $\left[\mathsf{b}_i^{\mathsf{R}}\right], [\mathsf{b}_i]^{\mathbf{B}}$ to generate $[\![\mathsf{b}_i^{\mathsf{R}}]\!], [\![\mathsf{b}_i]\!]^{\mathbf{B}}$.

If $\mathsf{isTr} = 1$, set $([\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]) = \left(\sum_{i=0}^{\mathsf{k}-1} 2^i [\![\mathsf{b}_i^{\mathsf{R}}]\!], \sum_{i=d}^{\mathsf{k}-1} 2^{i-d} [\![\mathsf{b}_i^{\mathsf{R}}]\!]\right)$

Figure 7.14: Semi-honest: Doubly shared bits.

At a high-level, generation of doubly shared bits relies on the property that every non-zero quadratic residue has exactly one root when working over fields. The work of [67], operating over rings, shows that something similar holds over rings as well. Concretely, according to lemma 4.1 of [67]: *if* a *is such that* $\mathsf{a}^2 \equiv_\ell 1$*, then* a *is congruent mod* 2^ℓ *to either* $1, -1, -1+2^{\ell-1}, 1+2^{\ell-1}$. Thus, the doubly shared bit generation protocol of [67] proceeds as follows. Generate a^2 for $\mathsf{a} \in \mathbb{Z}_{2^{\ell+2}}$ such that $\mathsf{a}^2 \equiv_{\ell+2} 1$, and compute its smallest root c mod $2^{\ell+2}$. Compute $(\mathsf{c}^{-1}\mathsf{a})$, and by lemma 4.1 of [67] it follows that $\mathsf{c}^{-1}\mathsf{a} \in \{\pm 1, \pm 1 + 2^{\ell+1}\}$. That is, $(\mathsf{c}^{-1}\mathsf{a})$ is congruent to ± 1 modulo $2^{\ell+1}$. Thus, $\mathsf{d} = \mathsf{c}^{-1}\mathsf{a} + 1$ is congruent to 0 or 2 modulo $2^{\ell+1}$ with equal probability. Hence, setting $\mathsf{b} = \mathsf{d}/2$ outputs bit $\mathsf{b} = 0$ or bit $\mathsf{b} = 1$ with equal probability. Observe that the computation has to be performed over $\mathbb{Z}_{2^{\ell+2}}$. Hence, in the protocol description, we use $\ell + 2$ in the superscript to distinguish shares of x over $\mathbb{Z}_{2^{\ell+2}}$ from its shares over $\mathbb{Z}_{2^\ell}$.

The main change in Π_{dsBits} from that of the protocol in [67] is that to generate $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$ Π_{dsBits} generates ℓ random doubly shared bits $\mathsf{b}_0, \ldots, \mathsf{b}_{\ell-1} \in \mathbb{Z}_2$ instead of a single one, and composes these ℓ bits to generate r, and composes the higher $\ell - d$ bits to generate r^d, as follows.

$$([\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]) = \left(\sum_{i=0}^{\ell-1} 2^i [\![\mathsf{b}_i^{\mathsf{R}}]\!], \sum_{i=d}^{\ell-1} 2^{i-d} [\![\mathsf{b}_i^{\mathsf{R}}]\!]\right) \tag{7.2}$$

Looking ahead, Π_{dsBits} can also be used only to generate a single doubly shared random bit, which finds use in other building blocks such as bit to arithmetic conversion and arithmetic to Boolean conversion. Thus, to distinguish the case when $([\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!])$ has to be generated versus when only a single doubly shared bit is to be generated, Π_{dsBits} takes a bit isTr as input and gives as output a doubly shared bit $[\![\mathsf{b}^{\mathsf{R}}]\!], [\![\mathsf{b}]\!]^{\mathbf{B}}$ if $\mathsf{isTr} = 0$, and $([\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!])$ otherwise. The protocol appears in Fig. 7.14.

A final thing to note is that the computation in Π_{dsBits} proceeds over secret-shared data. Thus, to generate shares of the doubly shared bit b, one should be able to divide each share of d by 2, which necessitates d and its shares to be even. This holds true since $[\mathsf{d}]^{\ell+2} = \mathsf{c}^{-1}[\mathsf{a}]^{\ell+2} + 1 = \mathsf{c}^{-1}\left(2[\mathsf{u}]^{\ell+2} + 1\right) + 1 = 2\mathsf{c}^{-1}[\mathsf{u}]^{\ell+2} + \mathsf{c}^{-1} + 1$. Here, $2\mathsf{c}^{-1}[\mathsf{u}]^{\ell+2}$ is even due to multiplication by 2, while $\mathsf{c}^{-1} + 1$ is even since c^{-1} is odd by definition.

7.4.5 Dot product

Given $[\![\cdot]\!]$-shares of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of size n, dot product outputs $[\![\mathsf{z}]\!]$ where $\mathsf{z} = \boldsymbol{x} \odot \boldsymbol{y} = \sum_{k=1}^{\mathsf{n}} \mathsf{x}_k \mathsf{y}_k$ and $\odot$ denotes the dot product operation. The design of our multiplication protocol enables the easy extension to support dot product computation without incurring any overhead.

Concretely, similar to multiplication,

$$\begin{aligned}\mathsf{z}-\mathsf{r} &= (\boldsymbol{x}\odot\boldsymbol{y})-\mathsf{r}\\ &= \sum_{k=1}^{\mathsf{n}} \mathrm{M}_{\mathsf{x}_k\mathsf{y}_k} - \sum_{k=1}^{\mathsf{n}} \beta_{\mathsf{x}_k}\alpha_{\mathsf{y}_k} - \sum_{k=1}^{\mathsf{n}} \beta_{\mathsf{y}_k}\alpha_{\mathsf{x}_k} + \sum_{k=1}^{\mathsf{n}} \Lambda_{\mathsf{x}_k\mathsf{y}_k} - \mathsf{r} \end{aligned} \tag{7.3}$$

In each of the summands of $\mathsf{z}-\mathsf{r}$, each of the n product terms can be generated similar to that in the multiplication protocol, which can then be locally summed up before sending it towards P_{king}. The formal protocol details appear in Fig. 7.15. Looking ahead, for matrix multiplication, each element of the resultant matrix can be computed via a dot product.

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

Preprocessing:

1. Invoke Π_{rand} to generate $[\mathsf{r}]$ where $\mathsf{r}\in\mathbb{Z}_{2^\ell}$, followed by $\Pi_{[\cdot]\to\langle\cdot\rangle}$ to generate $\langle\mathsf{r}\rangle$.
2. Invoke $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$ on $[\alpha_{\mathsf{x}_k}], [\alpha_{\mathsf{y}_k}]$ to generate $\langle\Lambda_{\mathsf{x}_k\mathsf{y}_k}\rangle$ for $k\in\{1,\ldots,\mathsf{n}\}$, and compute $\langle\sum_{k=1}^{\mathsf{n}}\Lambda_{\mathsf{x}_k\mathsf{y}_k}-\mathsf{r}\rangle = \sum_{k=1}^{\mathsf{n}}\langle\Lambda_{\mathsf{x}_k\mathsf{y}_k}\rangle - \langle\mathsf{r}\rangle$.
3. $P_i\in\mathcal{E}$ invokes $\Pi_{[\cdot]\to^{\mathcal{E}}\langle\cdot\rangle}$ on $[\alpha_{\mathsf{x}_k}], [\alpha_{\mathsf{y}_k}]$ to generate ${}^{\mathcal{E}}\langle\alpha_{\mathsf{x}_k}\rangle_i, {}^{\mathcal{E}}\langle\alpha_{\mathsf{y}_k}\rangle_i$, respectively, for $k\in\{1,\ldots,\mathsf{n}\}$.
4. $P_i\in\mathcal{D}$ sends $\langle\sum_{k=1}^{\mathsf{n}}\Lambda_{\mathsf{x}_k\mathsf{y}_k}-\mathsf{r}\rangle_i$ to P_{king}, who sets $\mathsf{D}=\sum_{i:P_i\in\mathcal{D}}\langle\sum_{k=1}^{\mathsf{n}}\Lambda_{\mathsf{x}_k\mathsf{y}_k}-\mathsf{r}\rangle_i$.

Online:

1. $P_i\in\mathcal{E}$ computes ${}^{\mathcal{E}}\langle\zeta\rangle_i = \sum_{k=1}^{\mathsf{n}}\left(-\beta_{\mathsf{x}_k}{}^{\mathcal{E}}\langle\alpha_{\mathsf{y}_k}\rangle_i - \beta_{\mathsf{y}_k}{}^{\mathcal{E}}\langle\alpha_{\mathsf{x}_k}\rangle_i\right) + \langle\sum_{k=1}^{\mathsf{n}}\Lambda_{\mathsf{x}_k\mathsf{y}_k}-\mathsf{r}\rangle_i$, and sends ${}^{\mathcal{E}}\langle\zeta\rangle_i$ to P_{king}.
2. P_{king} computes $\mathsf{E}=\sum_{k=1}^{\mathsf{n}}\mathrm{M}_{\mathsf{x}_k\mathsf{y}_k}+\sum_{i:P_i\in\mathcal{E}}{}^{\mathcal{E}}\langle\zeta\rangle_i$ and sends $\mathsf{z}-\mathsf{r}=\mathsf{D}+\mathsf{E}$ to all parties in $\mathcal{E}$.
3. Invoke $\Pi_{\cdot\to[\![\cdot]\!]}$ on $\mathsf{z}-\mathsf{r}$ to generate $[\![\mathsf{z}-\mathsf{r}]\!]$, and compute $[\![\mathsf{z}]\!]=[\![\mathsf{z}-\mathsf{r}]\!]+[\![\mathsf{r}]\!]$.

Figure 7.15: Semi-honest: Dot product protocol.

7.4.6 Multi input multiplication

3-input and 4-input multiplication protocols have showcased their wide applicability in improving the online phase complexity [138, 194, 191]. Concretely, computing $\mathsf{z}=\mathsf{abc}$ (3-input) or $\mathsf{z}=\mathsf{abcd}$ (4-input) naively requires at least two sequential invocations of 2-input multiplication protocol in the online phase. Instead, 3-input and 4-input multiplication protocol, respectively, enables performing this computation with the same online complexity as that of a *single* 2-input multiplication. Thus, we design 3-input and 4-input multiplication protocols by extending the techniques of [194, 138] to the n-party setting. Designing these protocols require modifications in the preprocessing steps. Consider 3-input multiplication (Fig. 7.16) where the goal is to

generate $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{abc}$ given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!], [\![\mathsf{c}]\!]$. Note that

$$\begin{aligned}\mathsf{z} - \mathsf{r} = \mathsf{abc} - \mathsf{r} &= (\beta_\mathsf{a} - \alpha_\mathsf{a})(\beta_\mathsf{b} - \alpha_\mathsf{b})(\beta_\mathsf{c} - \alpha_\mathsf{c}) - \mathsf{r} \\ &= \mathrm{M}_\mathsf{abc} - \mathrm{M}_\mathsf{ac}\alpha_\mathsf{b} - \mathrm{M}_\mathsf{bc}\alpha_\mathsf{a} - \mathrm{M}_\mathsf{ab}\alpha_\mathsf{c} + \beta_\mathsf{a}\Lambda_\mathsf{bc} + \beta_\mathsf{b}\Lambda_\mathsf{ac} + \beta_\mathsf{c}\Lambda_\mathsf{ab} - \Lambda_\mathsf{abc} - \mathsf{r}\end{aligned}$$

We follow an approach closely related to 2-input multiplication, with the difference being that parties additionally require to generate the additive sharing of $\Lambda_\mathsf{bc}, \Lambda_\mathsf{ac}$ and Λ_abc during preprocessing. Given these sharings, parties proceed with a similar online phase as in Π_Mul to compute the 3-input multiplication without inflating the online cost. Specifically, the following steps are performed in the preprocessing phase.

– For generating ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ac}\rangle, {}^{\mathcal{E}}\langle\Lambda_\mathsf{bc}\rangle$ parties first compute the respective additive sharings ($\langle\cdot\rangle$) using $[\alpha_\mathsf{a}], [\alpha_\mathsf{b}]$ and $[\alpha_\mathsf{c}]$ (via two invocations of $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$). Following this parties in $\mathcal{D}$ communicate their share of $\langle\Lambda_\mathsf{ac}\rangle$ and $\langle\Lambda_\mathsf{bc}\rangle$ to P_king, each masked with a random $\langle\cdot\rangle$-sharing of 0 (generated using $\Pi_{\langle 0\rangle}$). This establishes ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ac}\rangle, {}^{\mathcal{E}}\langle\Lambda_\mathsf{bc}\rangle$ among parties in $\mathcal{E}$.

– For generating ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ab}\rangle$, a slightly different approach is taken where parties first generate $[\Lambda_\mathsf{ab}]$ using $[\alpha_\mathsf{a}], [\alpha_\mathsf{b}]$ (as explained later), followed by non-interactively generating ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ab}\rangle$ (via $\Pi_{[\cdot]\to{}^{\mathcal{T}}\langle\cdot\rangle}$). The reason for generating $[\Lambda_\mathsf{ab}]$ (instead of directly generating ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ab}\rangle$) is to facilitate the generation of ${}^{\mathcal{E}}\langle\Lambda_\mathsf{abc} - \mathsf{r}\rangle$ from $[\Lambda_\mathsf{ab}]$, $[\alpha_\mathsf{c}]$ and $\langle\mathsf{r}\rangle$, which closely follows the preprocessing phase of the 2-input multiplication. Specifically, parties can generate $\langle\Lambda_\mathsf{abc}\rangle$ using $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$ on $[\Lambda_\mathsf{ab}]$, $[\alpha_\mathsf{c}]$, followed by parties in $\mathcal{D}$ communicating their $\langle\Lambda_\mathsf{abc}\rangle$ shares masked with $\langle\cdot\rangle$-sharing of a random r to P_king. This generates ${}^{\mathcal{E}}\langle\Lambda_\mathsf{abc} + \mathsf{r}\rangle$-sharing required during online phase.

– Regarding generation of $[\Lambda_\mathsf{ab}]$, all parties generate $[\cdot]$-sharing of a random $\gamma \in \mathbb{Z}_{2^\ell}$ non-interactively and convert it to $\langle\gamma\rangle$. Parties then compute $\langle\Lambda_\mathsf{ab} + \gamma\rangle$ by computing $\langle\Lambda_\mathsf{ab}\rangle$ from $[\alpha_\mathsf{a}], [\alpha_\mathsf{b}]$ followed by summing it up with $\langle\gamma\rangle$. Parties reconstruct this value towards P_king, who then generates $[\Lambda_\mathsf{ab} + \gamma]$, from which parties compute $[\Lambda_\mathsf{ab}] = [\Lambda_\mathsf{ab} + \gamma] - [\gamma]$. On obtaining $[\Lambda_\mathsf{ab}]$, parties generate ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ab}\rangle$ by invoking $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$.

Similarly, for 4-input multiplication, parties need to generate the additive sharing of $\Lambda_\mathsf{ad}, \Lambda_\mathsf{bd}, \Lambda_\mathsf{cd}, \Lambda_\mathsf{abd}, \Lambda_\mathsf{acd}, \Lambda_\mathsf{bcd}, \Lambda_\mathsf{abcd}$ in addition to those required in the case of 3-input multiplication. Specifically, generation of ${}^{\mathcal{E}}\langle\cdot\rangle$-shares (additive shares) of $\Lambda_\mathsf{ac}, \Lambda_\mathsf{ad}, \Lambda_\mathsf{bc}, \Lambda_\mathsf{bd}$ can proceed similar to generation of ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ac}\rangle$ in $\Pi_{\mathsf{3\text{-}Mul}}$. Generation of ${}^{\mathcal{E}}\langle\cdot\rangle$-shares of $\Lambda_\mathsf{ab}, \Lambda_\mathsf{cd}$ is carried out by first generating its $[\cdot]$-shares. This enables generation of ${}^{\mathcal{E}}\langle\cdot\rangle$-shares of $\Lambda_\mathsf{abc}, \Lambda_\mathsf{abd}, \Lambda_\mathsf{acd}, \Lambda_\mathsf{bcd}$ following steps similar to generation of ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ac}\rangle$ in $\Pi_{\mathsf{3\text{-}Mul}}$. Finally, ${}^{\mathcal{E}}\langle\Lambda_\mathsf{abcd} - \mathsf{r}\rangle$ is generated similar to generating ${}^{\mathcal{E}}\langle\Lambda_\mathsf{abc} + \mathsf{r}\rangle$ in $\Pi_{\mathsf{3\text{-}Mul}}$. We omit formal details of the 4-input multiplication protocol, $\Pi_{\mathsf{4\text{-}Mul}}$, as it is very close to $\Pi_{\mathsf{3\text{-}Mul}}$. Table 7.3 compares the cost of computing $\mathsf{z} = \mathsf{abc}$ and $\mathsf{z} = \mathsf{abcd}$ via a 2-input, 3-input and 4-input multiplication.

Protocol $\Pi_{\mathsf{3\text{-}Mul}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!], [\![\mathsf{c}]\!])$

Preprocessing:

1. Invoke Π_{rand} to generate $[\mathsf{r}]$ and $[\gamma]$ where $\mathsf{r}, \gamma \in \mathbb{Z}_{2^\ell}$. Invoke $\Pi_{[\cdot]\to\langle\cdot\rangle}$ to generate $\langle\mathsf{r}\rangle, \langle\gamma\rangle$.
2. Invoke $\Pi_{\langle 0\rangle}$ to generate two different $\langle\cdot\rangle$-shares of 0: $\langle 0_1\rangle$, $\langle 0_2\rangle$.
3. Generation of ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ac}}\rangle, {}^{\mathcal{E}}\langle\Lambda_{\mathsf{bc}}\rangle$.
   - Invoke $\Pi_{[\cdot][\cdot]\to\langle\cdot\rangle}$ on $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{c}}]$ to generate $\langle\Lambda_{\mathsf{ac}}\rangle_i$, and compute $\langle\Lambda_{\mathsf{ac}} + 0_1\rangle_i = \langle\Lambda_{\mathsf{ac}}\rangle_i + \langle 0_1\rangle_i$.
   - $P_i \in \mathcal{D}$ sends $\langle\Lambda_{\mathsf{ac}} + 0_1\rangle_i$ to $P_{\mathsf{king}} (= P_{t+1})$.
   - Analogous steps are carried out to generate $\langle\Lambda_{\mathsf{bc}} + 0_2\rangle$.
   - $P_i \in \mathcal{E} \setminus P_{t+1}$ sets ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{bc}}\rangle_i = \langle\Lambda_{\mathsf{bc}} + 0_2\rangle_i$ and ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ac}}\rangle_i = \langle\Lambda_{\mathsf{ac}} + 0_1\rangle_i$.
   - P_{t+1} sets ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{bc}}\rangle_{t+1} = \langle\Lambda_{\mathsf{bc}} + 0_2\rangle_{t+1} + \sum_{i:P_i\in\mathcal{D}}\langle\Lambda_{\mathsf{bc}} + 0_2\rangle_i$ and ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ac}}\rangle_{t+1} = \langle\Lambda_{\mathsf{ac}} + 0_1\rangle_{t+1} + \sum_{i:P_i\in\mathcal{D}}\langle\Lambda_{\mathsf{ac}} + 0_1\rangle_i$.
4. Generation of ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ab}}\rangle$.
   - Invoke $\Pi_{[\cdot][\cdot]\to\langle\cdot\rangle}$ on $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ to generate $\langle\Lambda_{\mathsf{ab}}\rangle_i$, set $\langle\Lambda_{\mathsf{ab}} + \gamma\rangle_i = \langle\Lambda_{\mathsf{ab}}\rangle_i + \langle\gamma\rangle_i$, and send $\langle\Lambda_{\mathsf{ab}} + \gamma\rangle_i$ to P_{king}.
   - P_{king} reconstructs $\Lambda_{\mathsf{ab}} + \gamma$, and sends $\Lambda_{\mathsf{ab}} + \gamma$ to $P_i \in \mathcal{E}$. Parties non-interactively generate $[\Lambda_{\mathsf{ab}} + \gamma]$ via $\Pi_{\cdot\to[\![\cdot]\!]}$ and $\Pi_{[\![\cdot]\!]\to[\cdot]}$.
   - Compute $[\Lambda_{\mathsf{ab}}] = [\Lambda_{\mathsf{ab}} + \gamma] - [\gamma]$ and invoke $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$ on $[\Lambda_{\mathsf{ab}}]$ to generate ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ab}}\rangle$.
5. Generation of ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle$.
   - Invoke $\Pi_{[\cdot][\cdot]\to\langle\cdot\rangle}$ on $[\Lambda_{\mathsf{ab}}], [\alpha_{\mathsf{c}}]$ to generate $\langle\Lambda_{\mathsf{abc}}\rangle_i$, and compute $\langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_i = \langle\Lambda_{\mathsf{abc}}\rangle_i + \langle\mathsf{r}\rangle_i$.
   - $P_i \in \mathcal{D}$ sends $\langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_i$ to P_{king}.
   - $P_i \in \mathcal{E} \setminus P_{t+1}$ sets ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_i = \langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_i$.
   - P_{t+1} sets ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_{t+1} = \langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_{t+1} + \sum_{i:P_i\in\mathcal{D}}\langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_i$.
6. $P_i \in \mathcal{E}$ invoke $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$ on $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ and $[\alpha_{\mathsf{c}}]$ to generate ${}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_i, {}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle_i, {}^{\mathcal{E}}\langle\alpha_{\mathsf{c}}\rangle_i$, respectively.

Online:

1. $P_i \in \mathcal{E}$ computes and sends ${}^{\mathcal{E}}\langle\zeta\rangle_i = -\mathrm{M}_{\mathsf{ac}}{}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle_i - \mathrm{M}_{\mathsf{bc}}{}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_i - \mathrm{M}_{\mathsf{ab}}{}^{\mathcal{E}}\langle\alpha_{\mathsf{c}}\rangle_i + \beta_{\mathsf{a}}{}^{\mathcal{E}}\langle\Lambda_{\mathsf{bc}}\rangle_i + \beta_{\mathsf{b}}{}^{\mathcal{E}}\langle\Lambda_{\mathsf{ac}}\rangle_i + \beta_{\mathsf{c}}{}^{\mathcal{E}}\langle\Lambda_{\mathsf{ab}}\rangle_i - {}^{\mathcal{E}}\langle\Lambda_{\mathsf{abc}} + \mathsf{r}\rangle_i$ to P_{king}.
2. P_{king} computes and sends $\mathsf{z} - \mathsf{r} = \mathrm{M}_{\mathsf{abc}} + \sum_{i:P_i\in\mathcal{E}}{}^{\mathcal{E}}\langle\zeta\rangle_i$ to $P_i \in \mathcal{E}$.
3. Invoke $\Pi_{\cdot\to[\![\cdot]\!]}$ on $\mathsf{z} - \mathsf{r}$ to generate $[\![\mathsf{z} - \mathsf{r}]\!]$, and compute $[\![\mathsf{z}]\!] = [\![(\mathsf{z} - \mathsf{r})]\!] + [\![\mathsf{r}]\!]$.

Figure 7.16: Semi-honest: 3-input multiplication protocol.

The recent work of [95] provides a method to reduce the round complexity of circuit evaluation. They group the (distinct) consecutive layers in the circuit into pairs and perform a parallel evaluation of all gates in the two layers in a group. Consider a multiplication gate with inputs x, y (obtained as output from a previous layer) and output z. Their approach considers

Multiplication type	Building Block	Communication		Online Rounds
		Prep.	Online	
$z = abc$	2-input mult.	$2t\ell$	$4t\ell$	4
	3-input mult.	$6t\ell$	$2t\ell$	2
$z = abcd$	2-input mult.	$3t\ell$	$6t\ell$	4
	4-input mult.	$15t\ell$	$2t\ell$	2

Table 7.3: Semi-honest: Communication and rounds for multi-input multiplications.

three cases: (i) if x and y are not the outputs of a multiplication gate, (ii) exactly one among x, y is the output of a multiplication gate, and (iii) both x, y are outputs of a multiplication gate. We observe that case (ii) and (iii) in their approach resembles multi-input multiplication, which allows evaluating the second layer of multiplication ($z = x \cdot y$) non-interactively, thereby saving on rounds. For instance, consider a 2-layer sub-circuit as in Fig. 7.17 where $x = a \cdot b, y = c \cdot d$ are outputs of a multiplication gate which are fed as input to a multiplication gate in the next level. The approach of [95] allows computation of $z = (a \cdot b) \cdot (c \cdot d)$ in a single shot, which is equivalent to computing z via a 4-input multiplication in our case. Similarly, when only one of the inputs (either x or y) is the output of multiplication, computation of $z = x \cdot y$ resembles a 3-input multiplication. Thus, cases (i), (ii), and (iii) correspond to 2-input, 3-input, and 4-input multiplication, respectively, in our work and are sufficient to reduce the round complexity of any circuit evaluation by half. Hence, we restrict our focus to 3 and 4-input multiplication, although our technique can be generalized to gates with arbitrarily large fan-in.

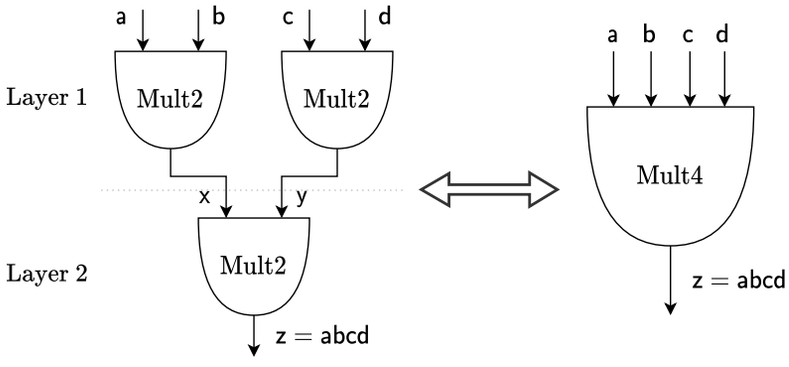


Figure 7.17: 4-input multiplication.

7.5 Extending to malicious security

The ideal functionality $\mathcal{F}_f$ for evaluating a function f in the n-party setting while providing malicious security (with fairness) appears in Fig. 7.18.

Functionality $\mathcal{F}_{\text{n-PC}}^{\text{mal}}$

$\mathcal{F}_f$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}^{\text{mal}}$. Let f denote the function to be computed. Let x_s be the input of party P_s, and y_s be the corresponding output, i.e $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$. $\mathcal{S}^{\text{mal}}$ is also allowed to send a special command, `abort`, which indicates that none of the parties should receive the output.

Step 1: $\mathcal{F}_f$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$. If $(\mathsf{Input}, *)$ is already received from P_s, then ignore the current message. Otherwise, record $x'_s = x_s$ internally.

Step 2: Compute $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$.

Step 3: Send $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$. Here, $\mathsf{y}_s = \texttt{abort}$ for $P_s \in \mathcal{P}$ if $\mathcal{S}^{\text{mal}}$ sent $(\mathsf{Signal}, \texttt{abort})$.

Figure 7.18: Malicious: Ideal functionality for evaluating function f with fairness.

The input sharing and output reconstruction protocols for the malicious setting can be obtained efficiently from the semi-honest protocol following standard approaches [193, 136, 78]. However, the same cannot be said about multiplication. Note that although a maliciously secure multiplication protocol can be achieved by compiling our semi-honest protocol using compiler techniques such as [3, 31], the resultant protocol has an expensive online phase. For instance, using the compiler of [3] yields a protocol that requires computation over *extended* rings and communicating $4t$ extended ring elements in the online phase. This is not favourable compared to working over plain rings, especially in the online phase. Further, compilers such as that in [31] require heavy computational machinery like reliance on zero-knowledge proofs in the online phase, which is also not desirable. Thus, to attain a computation and communication efficient online phase, departing from the aforementioned compiler-based approaches, we design a maliciously secure multiplication protocol that requires communicating $3t$ *ring* elements in each phase. It is worth noting that we can do this while retaining the benefits of requiring only $t + 1$ parties in the online phase (for most of the computation). The remaining t parties are required to come online only for a short one-time verification phase, which is deferred to the end of the computation. Deferring verification may result in a privacy breach [94]. However, we describe later why the privacy breach does not arise in our protocol. With this background, in what follows next, we begin with describing the input sharing and output reconstruction protocols and then focus on discussing the challenges encountered and their resolutions for

obtaining a maliciously secure *multiplication* protocol.

7.5.1 Input sharing

This protocol is similar to the semi-honest one, where to enable P_s to generate $[\![\mathsf{a}]\!]$, parties generate $[\alpha_\mathsf{a}]$ such that P_s learns α_a, followed by P_s sending the masked value $\beta_\mathsf{a} = \mathsf{a} + \alpha_\mathsf{a}$ to all. However, note that a corrupt P_s can cause inconsistency among the honest parties by sending different masked values. To ensure the same value is received by all, parties perform a hash-based consistency check, denoted by Π_{agree} (§7.3), where each party sends a hash of the received masked value(s) to every other party and `aborts` if it receives inconsistent hashes. Note that this check for all the inputs can be combined, thereby amortizing the cost. The formal protocol appears in Fig. 7.19.

Protocol $\Pi^{\mathsf{M}}_{\mathsf{Sh}}(P_s, \mathsf{a})$

Preprocessing: Invoke $\Pi_{\mathsf{pRand}}(P_s)$ to generate $[\alpha_\mathsf{a}]$, with P_s learning $\alpha_\mathsf{a} \in \mathbb{Z}_{2^\ell}$.

Online: P_s computes and sends $\beta_\mathsf{a} = \mathsf{a} + \alpha_\mathsf{a}$ to all $P_i \in \mathcal{P}$.

Verification: Invoke Π_{agree} on $\{\beta_\mathsf{a}\}$.

Figure 7.19: Malicious: Input sharing protocol.

7.5.2 Reconstruction

To reconstruct $[\![\cdot]\!]$-shared value a towards $P_s \in \mathcal{P}$, observe that each share that P_s misses is held by $t+1$ other parties. Each of these parties sends the missing share to P_s. If the received values for a share are consistent, P_s uses this value to perform reconstruction, and `aborts` otherwise. As an optimization, one party can send the missing share while reconstructing several values, and t others can send its hash.

Fairness is a stronger security notion than security with abort, where, during reconstruction, either all parties learn the output or none do. For fair reconstruction, we extend the techniques in [193] to the n-party setting, where commitments are generated on each share of the mask of the output z (required to reconstruct z) by $t+1$ parties in the preprocessing phase. During the online phase, these commitments are opened towards the respective parties if all the parties are alive (did not `abort`). Since each share of the mask is held by $t+1$ parties and there is at least one honest party among every set of $t+1$ parties, it is guaranteed that parties will obtain the correct opening for the commitment of the missing share from the honest party, and all honest

parties can reconstruct the output. Else, if the adversary misbehaved at some step during the protocol, none of the honest parties will share the opening information, and none will obtain the output. The formal protocol $\Pi^{\mathsf{fair}}_{\mathsf{Rec}}$ appears in Fig. 7.20.

Protocol $\Pi^{\mathsf{fair}}_{\mathsf{Rec}}([\![\mathsf{z}]\!])$

Preprocessing:

1. For $j \in \{1, \ldots, \mathsf{q}\}$:
   - Each $P_i \in \mathcal{T}_j$ generates commitments on $[\alpha_\mathsf{z}]_{\mathcal{T}_j}$ using the common randomness, and sends to all other parties.
   - $P_i \notin \mathcal{T}_j$ `aborts` if commitments for $[\alpha_\mathsf{z}]_{\mathcal{T}_j}$ are inconsistent.

Online:

1. Each party $P_i \in \mathcal{P}$ which has not aborted broadcasts a bit $\mathsf{alive} = 1$.
2. If all the parties broadcast $\mathsf{alive} = 1$, $P_i \in \mathcal{P}$ sends the opening of the commitments to the shares in $[\alpha_\mathsf{z}]_i$ to the respective parties.
3. Parties use the valid decommitment to obtain the missing share of α_z, reconstruct α_z, and compute $\mathsf{z} = \beta_\mathsf{z} - \alpha_\mathsf{z}$.

Figure 7.20: Malicious: Fair reconstruction protocol.

7.5.3 Multiplication

To enable generation of $[\![\mathsf{z}]\!] = [\![\mathsf{ab}]\!]$ from $[\![\mathsf{a}]\!]$ and $[\![\mathsf{b}]\!]$, we retain the high-level ideas from the semi-honest protocol. Our task reduces to (i) generating additive shares of Λ_ab among parties in $\mathcal{E}$ (i.e. ${}^{\mathcal{E}}\langle\Lambda_\mathsf{ab}\rangle$) given $[\alpha_\mathsf{a}]$ and $[\alpha_\mathsf{b}]$, in the preprocessing phase, and (ii) reconstructing $\mathsf{z}-\mathsf{r}$ in the online phase. Given (i), computing ${}^{\mathcal{E}}\langle\mathsf{z}-\mathsf{r}\rangle$ in the online phase is a local operation. Given (ii), parties can invoke $\Pi_{\cdot\to[\![\cdot]\!]}$ to generate $[\![\mathsf{z}-\mathsf{r}]\!]$, and compute $[\![\mathsf{z}]\!] = [\![\mathsf{z}-\mathsf{r}]\!] + [\![\mathsf{r}]\!]$, where $[\![\mathsf{r}]\!]$ is generated in the preprocessing phase, as discussed in the semi-honest case.

For task (i), our idea for the semi-honest case, of making parties in $\mathcal{D}$ send their shares to P_{king}, does not work in the presence of a malicious adversary. To address this, we make black-box use of a maliciously secure multiplication protocol, abstracted as a functionality $\mathcal{F}_{\mathsf{MulPre}}$ in Fig. 7.21, that computes $[\Lambda_\mathsf{ab}]$ from $[\alpha_\mathsf{a}]$, $[\alpha_\mathsf{b}]$. In this work, we instantiate $\mathcal{F}_{\mathsf{MulPre}}$ with the state-of-the-art multiplication protocol of [31] that provides `abort` security and requires $3t$ elements of (amortized) communication. Note that although the protocol of [31] relies on zero-knowledge proofs, this computation is carried out in the preprocessing phase of our multiplication protocol. Moreover, since preprocessing is done for many instances in one shot, the zero-knowledge proof

can benefit from amortization. The parties then invoke $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$ to obtain ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ab}}\rangle$ from $[\Lambda_{\mathsf{ab}}]$. Looking ahead, $[\Lambda_{\mathsf{ab}}]$ also aids in performing the online verification check.

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

$\mathcal{F}_{\mathsf{MulPre}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}^{\mathsf{mal}}$. Let $\mathcal{T}_i$ be the set of the honest parties.

Input: $\mathcal{F}_{\mathsf{MulPre}}$ receives the $[\cdot]$-shares of a, b from the parties. It also receives $[\cdot]$-shares of $\mathsf{z} = \mathsf{ab}$ of corrupt parties from $\mathcal{S}^{\mathsf{mal}}$. $\mathcal{S}^{\mathsf{mal}}$ is also allowed to send a special command, $(\mathtt{abort}, \mathsf{P})$, which indicates that parties in $\mathcal{P}$ with indices in P should `abort`.

$\mathcal{F}_{\mathsf{MulPre}}$ proceeds as follows.

– Reconstruct a, b using the shares received from honest parties, and compute $\mathsf{z} = \mathsf{ab}$.

– Compute the $[\cdot]$-share of z to be held by the set of honest parties as the difference between z and the sum of $[\cdot]$-shares of z received from corrupt parties.

– Let y_s denote the $[\cdot]$-shares of z for party $P_s \in \mathcal{P}$. If received $(\mathtt{abort}, \mathsf{P})$ from $\mathcal{S}^{\mathsf{mal}}$, set $\mathsf{y}_s = \mathtt{abort}$ for P_s, where $s \in \mathsf{P}$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 7.21: Ideal functionality $\mathcal{F}_{\mathsf{MulPre}}$.

For task (ii), in the online phase, we retain the idea of parties in $\mathcal{E}$ optimistically reconstructing $\mathsf{z}-\mathsf{r}$ from their additive shares (${}^{\mathcal{E}}\langle\cdot\rangle$-share) to ensure that only the parties in $\mathcal{E}$ remain active for most of the computation. Moreover, this optimistic reconstruction requires only $\mathcal{O}(t)$-element communication rather than the $\mathcal{O}(t^2)$ required for reconstruction from $[\cdot]$-shares (which is what will be used later for performing verification, albeit to perform only one such reconstruction). Thus, similar to the semi-honest protocol, parties in $\mathcal{E}$ optimistically reconstruct $\mathsf{z}-\mathsf{r}$ towards P_{king}, who further sends the reconstructed value to the parties in $\mathcal{E}$. In the malicious setting, this approach requires additional care since a malicious party may send a wrong ${}^{\mathcal{E}}\langle\cdot\rangle$-share of $\mathsf{z}-\mathsf{r}$ to P_{king} or a malicious P_{king} may send an incorrectly reconstructed (inconsistent) $\mathsf{z}-\mathsf{r}$ to the parties. To account for these behaviours, the protocol is augmented with a short one-off verification phase to verify the consistency and correctness of $\mathsf{z}-\mathsf{r}$. This phase is executed at the end of the protocol and requires the presence of *all* parties, and hence the possession of $\mathsf{z}-\mathsf{r}$ by all. This is in contrast to the semi-honest protocol where $\mathsf{z}-\mathsf{r}$ is given to only parties in $\mathcal{E}$. To keep $\mathcal{D}$ disengaged for most of the online phase, sending $\mathsf{z}-\mathsf{r}$ to them is deferred till the end of the protocol. This send is a one-off and can be combined for all multiplication gates. Details of verification protocol Π_{Vrfy} (Fig. 7.22) are given next.

Protocol $\Pi_{\mathsf{Vrfy}}\left(\mathcal{P}, \{[\![\mathsf{a}_i]\!], [\![\mathsf{b}_i]\!], \mathsf{z}_i - \mathsf{r}_i, [\Lambda_{\mathsf{a}_i\mathsf{b}_i}], [\mathsf{r}_i]\}_{i=1}^{m}\right)$

Let $(\mathsf{a}_1, \mathsf{b}_1, \mathsf{z}_1), \ldots, (\mathsf{a}_m, \mathsf{b}_m, \mathsf{z}_m)$ denote the inputs and outputs of the m multiplication gates to be verified.

1. *Consistency Check.* Invoke Π_{agree} on $\{\mathsf{z}_1 - \mathsf{r}_1, \ldots, \mathsf{z}_m - \mathsf{r}_m\}$.
2. *Correctness Check.* Repeat the following κ times.
   - Generate random $\theta_1, \ldots, \theta_m \in \mathbb{Z}_{2^\ell}$ and compute

$$[\Omega] = \sum_{i=1}^{m} \theta_i \left(\mathsf{z}_i - \mathsf{r}_i - \left(\mathrm{M}_{\mathsf{a}_i\mathsf{b}_i} - \beta_{\mathsf{a}_i}[\alpha_{\mathsf{b}_i}] - \beta_{\mathsf{b}_i}[\alpha_{\mathsf{a}_i}] + [\Lambda_{\mathsf{a}_i\mathsf{b}_i}] - [\mathsf{r}_i]\right)\right)$$

   - For each $[\cdot]$-share of Ω, the $t+1$ parties possessing this share send it to every party that misses this share. If the recipient party receives inconsistent values for any missing share, it `aborts`.
   - Reconstruct Ω and `abort` if $\Omega \neq 0$.

Figure 7.22: Malicious: Verification protocol for all multiplication gates.

Verification comprises two checks—a *consistency* check to first verify that P_{king} has indeed sent the same $\mathsf{z} - \mathsf{r}$ to all the parties, followed by a *correctness* check to verify the correctness of the $\mathsf{z} - \mathsf{r}$. For the former, parties perform a hash-based consistency check of $\mathsf{z} - \mathsf{r}$, and abort in case of any inconsistency. If $\mathsf{z} - \mathsf{r}$ is consistent, parties verify its correctness. The high-level idea for verifying correctness is to *robustly* reconstruct $\mathsf{z} - \mathsf{r}$, but now from its $[\cdot]$-shares (can be computed given $[\alpha_\mathsf{a}], [\alpha_\mathsf{a}], [\Lambda_{\mathsf{ab}}]$ that are generated in the preprocessing phase). Parties can then verify if this reconstructed value equals the value received from P_{king}. Concretely, this is equivalent to robustly reconstructing $[\Omega] = [\mathsf{z} - \mathsf{r} - (\mathrm{M}_{\mathsf{ab}} - \beta_\mathsf{a}\alpha_\mathsf{b} - \beta_\mathsf{b}\alpha_\mathsf{a} + \Lambda_{\mathsf{ab}} - \mathsf{r})]$, where $\mathsf{z} - \mathsf{r}$ is the value received from P_{king}, and verifying if $\Omega = 0$. For robust reconstruction of $[\Omega]$, every party sends its $[\cdot]$-share to every other party who misses this share, and `aborts` in case of inconsistencies in the received values. Elaborately, reconstruction of Ω towards $P_s \in \mathcal{P}$ proceeds as follows. For each missing $[\cdot]$-share of Ω at P_s, each of the $t+1$ parties holding this share sends it to P_s. P_s uses this share for reconstruction if all the $t+1$ received values are consistent, else it `aborts`. Presence of at least one honest party among the $t+1$ guarantees that inconsistency, if any, can be detected. Since each share in $[\Omega]$ is held by $t+1$ parties, comprising at least one honest party, any cheating by up to t corrupt parties is guaranteed to be detected. Since reconstruction should happen towards at least $t+1$ parties, communicating a missing share towards all these $t+1$ parties requires $\mathcal{O}(t^2)$ communication in total, and there are $m = \binom{n}{h} - \binom{n-1}{h-1}$ such missing shares. Note that the cost of this reconstruction can be optimized using standard optimization techniques [3, 52], where the correctness of $\mathsf{z} - \mathsf{r}$ for several multiplication gates can be verified with a single reconstruction by reconstructing a linear combination of Ω for several gates and verifying equality with 0. Thus, only one

robust reconstruction from $[\cdot]$-shares is required for several multiplication gates, whose cost gets amortized due to verification across multiple gates.

It is worth noting that this random linear combination technique does not trivially work over rings. This is due to the existence of zero divisors which results in the linear combination being 0 with a probability 1/2 (which denotes the cheating probability of the adversary) [3]. Hence, to obtain the desired security, the verification check is repeated κ times where κ is the security parameter. This bounds the cheating probability of adversary to $1/2^{\kappa}$. Another approach is to perform the verification over extended rings [29, 30]. Specifically, verification operations are carried out over a ring $\mathbb{Z}_{2^\ell}/f(x)$, which is a ring of all polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ modulo a degree d polynomial $f(x)$ that is irreducible over $\mathbb{Z}_2$. Each element of $\mathbb{Z}_{2^\ell}$ is lifted to a degree d polynomial in $\mathbb{Z}_{2^\ell}[x]/f(x)$, which increases the communication required to perform verification by a factor of d.

Protocol $\Pi^{\mathsf{M}}_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!], \mathsf{isTr})$

$\mathsf{isTr} = 1$ denotes that truncation is required and $\mathsf{isTr} = 0$ denotes otherwise.

Preprocessing:

1. If $\mathsf{isTr} = 0$: invoke Π_{rand} to generate $[\mathsf{r}]$ where $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. Invoke $\Pi_{[\cdot]\to\langle\cdot\rangle}$ and $\Pi_{[\cdot]\to[\![\cdot]\!]}$ on $[\mathsf{r}]$ to generate $\langle\mathsf{r}\rangle$ and $[\![\mathsf{r}]\!]$, respectively.
   - Else, invoke $\Pi^{\mathsf{M}}_{\mathsf{dsBits}}(\mathcal{P}, 1)$ (Fig. 7.14) to generate $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$, and $\Pi_{[\![\cdot]\!]\to\langle\cdot\rangle}$ on $[\![\mathsf{r}]\!]$ to generate $\langle\mathsf{r}\rangle$.
2. Invoke Π_{multPre} on $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ to generate $[\Lambda_{\mathsf{ab}}]$.
3. $P_i \in \mathcal{E}$ invokes $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$ on $[\Lambda_{\mathsf{ab}}]$, $[\alpha_{\mathsf{a}}]$, $[\alpha_{\mathsf{b}}]$ and $[\mathsf{r}]$ to generate ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ab}}\rangle$, ${}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle$, ${}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle$ and ${}^{\mathcal{E}}\langle\mathsf{r}\rangle$, respectively.

Online:

1. $P_i \in \mathcal{E}$ computes ${}^{\mathcal{E}}\langle\zeta\rangle_i = -\beta_{\mathsf{a}}\,{}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle_i - \beta_{\mathsf{b}}\,{}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_i + {}^{\mathcal{E}}\langle\Lambda_{\mathsf{ab}} - \mathsf{r}\rangle_i$, and sends ${}^{\mathcal{E}}\langle\zeta\rangle_i$ to P_{king}.
2. P_{king} reconstructs ζ, computes and sends $\mathsf{z} - \mathsf{r} = \zeta + \mathrm{M}_{\mathsf{ab}}$ to all parties[a].
3. If $\mathsf{isTr} = 0$: invoke $\Pi_{\cdot\to[\![\cdot]\!]}$ on $\mathsf{z} - \mathsf{r}$ to generate $[\![\mathsf{z} - \mathsf{r}]\!]$, and compute $[\![\mathsf{z}]\!] = [\![\mathsf{z} - \mathsf{r}]\!] + [\![\mathsf{r}]\!]$.
   - Else, invoke $\Pi_{\cdot\to[\![\cdot]\!]}$ on $(\mathsf{z} - \mathsf{r})^d$ to generate $[\![(\mathsf{z} - \mathsf{r})^d]\!]$, and compute $[\![\mathsf{z}^d]\!] = [\![(\mathsf{z} - \mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$.

Verification for all multiplication gates: Invoke Π_{Vrfy} on $[\![\cdot]\!]$-shares of $(\mathsf{a}_1, \mathsf{b}_1, \mathsf{z}_1), \ldots, (\mathsf{a}_m, \mathsf{b}_m, \mathsf{z}_m)$ which denote the inputs and outputs of the m multiplication gates whose correctness is to be verified.

[a] $\mathsf{z} - \mathsf{r}$ is sent to parties in $\mathcal{E}$ during the online phase computation whereas it is sent to parties in $\mathcal{D}$ in a single shot before verification begins.

Figure 7.23: Malicious: Multiplication protocol.

The maliciously secure multiplication protocol (Fig. 7.23) can be broken down into the following:

– Preprocessing phase which involves generation of $[\Lambda_{\mathsf{ab}}]$ by invoking $\mathcal{F}_{\mathsf{MulPre}}$. Malicious behaviour, if any, will be caught by $\mathcal{F}_{\mathsf{MulPre}}$. $[\Lambda_{\mathsf{ab}}]$ is non-interactively converted into ${}^{\mathcal{E}}\langle\cdot\rangle$-shares of α_{ab}. ${}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle, {}^{\mathcal{E}}\langle\alpha_{\mathsf{b}}\rangle$ is also generated non-interactively.

– Generation of ${}^{\mathcal{E}}\langle\cdot\rangle$-shares of $\alpha_{\mathsf{a}}, \alpha_{\mathsf{b}}, \Lambda_{\mathsf{ab}}$ during preprocessing enables computation of ${}^{\mathcal{E}}\langle\mathsf{z}-\mathsf{r}\rangle$ in the online phase, and thereby reconstruction of $\mathsf{z}-\mathsf{r}$ via P_{king}. The crucial point to note here is that this requires the presence of only parties in $\mathcal{E}$ in the online phase. This is followed by non-interactive generation of $[\![\mathsf{z}-\mathsf{r}]\!]$ from which $[\![\mathsf{z}]\!]$ is computed as $[\![\mathsf{z}]\!] = [\![\mathsf{z}-\mathsf{r}]\!] + [\![\mathsf{r}]\!]$, where $[\![\mathsf{r}]\!]$ is generated during preprocessing.

– Finally, to catch malicious behaviour in the online phase, if any, in the verification phase, the correctness of the generated $[\![\mathsf{z}]\!]$ is checked simultaneously, for each z that is the output of a multiplication gate. This is done by invoking Π_{Vrfy}. Note that before this verification begins, P_{king} sends $\mathsf{z}-\mathsf{r}$ corresponding to all multiplication gates to parties in $\mathcal{D}$ in a single shot.

As pointed out in [94], deferring the correctness check to later may result in a privacy breach when using a sharing scheme that allows for redundancy (such as RSS or Shamir sharing). We next discuss this breach and explain how it is overcome in our case. We begin with explaining the attack that a malicious adversary can launch if reconstruction towards P_{king} is performed by relying on RSS (or Shamir sharing), naively. Consider a circuit with two sequential multiplication gates with the output of the first gate, say a, going as input to the second gate. Let b denote the other input to the second multiplication gate, and z denote its output. In a P_{king} based approach for multiplication, t parties send their respective (RSS/Shamir) share of a masked value to P_{king}. In particular, for the first multiplication gate in the circuit mentioned above, t parties send their corresponding share of $\mathsf{a}-\mathsf{r}_{\mathsf{a}}$ to P_{king}, which reconstructs it and sends it back to all. Delaying the verification allows a malicious P_{king} to send an inconsistent value of $\mathsf{a}-\mathsf{r}_{\mathsf{a}}$ to the parties, using which it can learn the private input b, as follows. Suppose P_{king} sends the correct $\mathsf{a}-\mathsf{r}_{\mathsf{a}}$ to all but one out of the remaining t online parties, to which it sends $\mathsf{a}-\mathsf{r}_{\mathsf{a}}+\delta$. Owing to this, for the next multiplication gate P_{king} receives the shares of $\mathsf{z}-\mathsf{r}_{\mathsf{z}}$ from the former $t-1$ parties and a share of $(\mathsf{a}+\delta)\mathsf{b}-\mathsf{r}_{\mathsf{z}} = \mathsf{z}+\delta\mathsf{b}-\mathsf{r}_{\mathsf{z}}$ from the latter party. Having obtained these and additionally using the shares of $\mathsf{z}-\mathsf{r}_{\mathsf{z}}$ and $\mathsf{z}+\delta\mathsf{b}-\mathsf{r}_{\mathsf{z}}$ corresponding to the t corrupt parties including itself, a malicious P_{king} can reconstruct $\mathsf{z}-\mathsf{r}_{\mathsf{z}}$ as well as $\mathsf{z}+\delta\mathsf{b}-\mathsf{r}_{\mathsf{z}}$, thus learning b in clear. The crux of this attack lies in the fact that a malicious adversary corrupting t parties, including P_{king}, already possesses t shares each of $\mathsf{z}-\mathsf{r}_{\mathsf{z}}$ and $\mathsf{z}+\delta\mathsf{b}-\mathsf{r}_{\mathsf{z}}$. Thus, an additional share of these obtained from the online parties allows it to carry out the attack successfully. However, this attack does not hold when working with additive (${}^{\mathcal{E}}\langle\cdot\rangle$) sharing, which is what

prevents our protocol from falling prey to this attack.

Elaborately, recall that in our protocol, during reconstruction towards P_{king}, any redundancy due to $[\![\cdot]\!]$-sharing is eliminated with parties switching to ${}^{\mathcal{E}}\langle\cdot\rangle$-sharing (additive sharing among parties in $\mathcal{E}$). Due to this, even if P_{king} sends inconsistent values to the parties, the ${}^{\mathcal{E}}\langle\cdot\rangle$-share of $\mathsf{z}-\mathsf{r}_{\mathsf{z}}$ or $\mathsf{z}+\delta\mathsf{b}-\mathsf{r}_{\mathsf{z}}$ that it receives, corresponds to an additive share defined with respect to parties in $\mathcal{E}$. Hence, this additionally received additive share cannot be combined with the shares held by the t corrupt parties to perform the reconstruction. Thus, the earlier strategy of P_{king} of using these additional shares in conjunction with the t corrupt shares to reconstruct $\mathsf{z}-\mathsf{r}_{\mathsf{z}}$ and $\mathsf{z}+\delta\mathsf{b}-\mathsf{r}_{\mathsf{z}}$ does not hold. The primary reason which prevents the attack is the elimination of redundancy in the sharing scheme by switching to $(t+1)$-out-of-$(t+1)$ additive sharing (${}^{\mathcal{E}}\langle\cdot\rangle$-sharing) for the set of parties in $\mathcal{E}$, which is known to withstand this attack [94]. However, this privacy breach persists in the protocol of [78].

Discussion about [78]: The above attack can be circumvented by making P_{king} broadcast the reconstructed value to all the parties, as discussed in [78]. To further optimize the protocol by requiring only $t+1$ parties to be active in the online phase, they rely on broadcast with abort, which comprises two phases—(i) *send*: where P_{king} sends the value to the recipients, and (ii) *verification*: where the recipients exchange hash of the received value among themselves, and abort in case of inconsistency. However, for amortization, they defer the verification (even with respect to broadcast) towards the end of the protocol, thus making their protocol susceptible to the aforementioned attack. We observe that one fix is to perform the verification with respect to broadcast after each level in the circuit. This, however, requires all the parties to be online. An optimization to let only the $t+1$ parties in the online phase to perform this verification after each level, thereby allowing the remaining t parties to be shut off. Specifically, this involves performing *verification* where the online parties exchange the hash of the received value and abort in case of inconsistency. When the remainder t (offline) parties come online towards the end of the protocol for verifying the correctness of the multiplication gates, this verification should be preceded by first verifying the consistency of the values broadcast by P_{king} to the offline parties (and involves the participation of all n parties). Since the online phase involves broadcasting the reconstructed value to t other online parties, this amounts to an exchange of $\mathcal{O}(t^2)$ hashes after each level, thereby incurring a circuit depth-dependent overhead in the communication cost as well as the rounds. In order for the communication cost to get amortized, it is required that the circuit has $\mathcal{O}(t^2)$ gates at each level. However, the overhead in terms of the number of rounds persists.

Lemma 7.2 *Protocol* $\Pi^{\mathsf{M}}_{\mathsf{mult}}$ *(Fig. 7.23) incurs a communication of* $3t$ *elements in the prepro-*

cessing phase and $3t$ *elements in* 2 *rounds in the online phase for multiplication when* $\mathsf{isTr} = 0$.

7.5.4 The complete MPC protocol

The complete maliciously secure MPC protocol, $\Pi^{\mathsf{mal}}_{\mathsf{MPC}}$, evaluating a function $f(\cdot)$ appears in Fig. 7.24.

Protocol $\Pi^{\mathsf{mal}}_{\mathsf{MPC}}(\mathcal{P}, f(\cdot))$

Let $f(\cdot)$ denote the function to be computed, which is represented as a circuit with linear gates (with inputs $\mathsf{a}, \mathsf{b} \in \mathbb{Z}_{2^\ell}$ and output $\mathsf{c} = c_1\mathsf{a} + c_2\mathsf{b}$, for constants $c_1, c_2 \in \mathbb{Z}_{2^\ell}$) and multiplication gates (with inputs $\mathsf{a}, \mathsf{b} \in \mathbb{Z}_{2^\ell}$, and output $\mathsf{c} = \mathsf{a} \cdot \mathsf{b}$). The circuit evaluation proceeds by evaluating the gates in a predetermined topological order. $\mathsf{isTr} = 1$ denotes perform truncation, $\mathsf{isTr} = 0$ denotes otherwise.

Preprocessing:

1. For each circuit input a held by P_s, parties execute the preprocessing steps of $\Pi^{\mathsf{M}}_{\mathsf{Sh}}(P_s, \mathsf{a})$.
2. For each linear gate with input wires a, b, output c, and constants c_1, c_2, parties locally compute $[\alpha_\mathsf{c}] = c_1 [\alpha_\mathsf{a}] + c_2 [\alpha_\mathsf{b}]$.
3. For each multiplication gate with input wires a, b and output c, parties execute the preprocessing steps of $\Pi^{\mathsf{M}}_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!], \mathsf{isTr})$.
4. For each output gate z, execute the preprocessing steps of $\Pi^{\mathsf{fair}}_{\mathsf{Rec}}([\![\mathsf{z}]\!])$.

Online:

1. For each input wire, parties execute the online steps of $\Pi^{\mathsf{M}}_{\mathsf{Sh}}(P_s, \mathsf{a})$, where P_s is the party designated to provide the input a.
2. For each linear gate with input wires a, b, output c, and constants c_1, c_2, parties locally compute $\beta_\mathsf{c} = c_1\beta_\mathsf{a} + c_2\beta_\mathsf{b}$.
3. For each multiplication gate with input wires a, b and output c, parties execute the online steps of $\Pi_{\mathsf{Mul}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!], \mathsf{isTr})$.
4. For each output wire z in the circuit, parties execute the online steps $\Pi^{\mathsf{fair}}_{\mathsf{Rec}}([\![\mathsf{z}]\!])$.

Figure 7.24: Malicious: The complete MPC protocol.

7.5.5 Multiplication with truncation

Similar to the semi-honest protocol, truncation can be incorporated in the malicious multiplication as well without inflating the online communication. For this, we rely on maliciously secure ideal functionality, $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ (Fig. 7.25), to generate the $[\![\cdot]\!]$-shares of $(\mathsf{r}, \mathsf{r}^d)$.

Functionality $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$

$\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}^{\mathsf{mal}}$.

Input: $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ optionally receives a special command, $(\mathtt{abort}, \mathsf{P})$, from $\mathcal{S}^{\mathsf{mal}}$ indicating that honest parties in $\mathcal{P}$ with indices in P should $\mathtt{abort}$.

$\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ proceeds as follows.

- Samples random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, and computes $\mathsf{r}^d = \mathsf{r}/2^{\mathsf{d}}$.
- Generates $[\![\cdot]\!]$-shares of r, r^d.
- Let y_s denote the $[\![\cdot]\!]$-shares of r, r^d for party $P_s \in \mathcal{P}$. If received $(\mathtt{abort}, \mathsf{P})$ from $\mathcal{S}^{\mathsf{mal}}$, set $\mathsf{y}_s = \mathtt{abort}$ for P_s, where $s \in \mathsf{P}$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 7.25: Ideal functionality $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$.

$\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ (Fig. 7.25) can be realized using the maliciously secure variant of Π_{dsBits} (Fig. 7.14), denoted as $\Pi^{\mathsf{M}}_{\mathsf{dsBits}}$ [67]. This protocol is similar to the semi-honest protocol except with the following differences to account for malicious behaviour. The $[\cdot]$-shares of $\mathsf{e}_i = \mathsf{a}^2$ are generated by invoking Π_{multPre} instead of relying on $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$. This ensures the generation of correct $[\cdot]$-shares of e_i, and malicious behaviour, if any, will lead to an $\mathtt{abort}$. Following this, e_i is either correctly reconstructed towards all, or parties $\mathtt{abort}$. This ensures that an adversary cannot lead to the reconstruction of an incorrect e_i. Concretely, for reconstruction, similar to multiplication, every party sends its $[\cdot]$-share to every other party, and $\mathtt{aborts}$ in case of inconsistencies in the received values[1]. The rest of the protocol steps (which are non-interactive) remain unchanged, and hence a formal protocol is omitted.

7.5.6 Dot product

To generate $[\![\mathsf{z}]\!]$ for $\mathsf{z} = \mathbf{x} \odot \mathbf{y}$ where $\mathbf{x}$ and $\mathbf{y}$ are vectors of size n and are $[\![\cdot]\!]$-shared, protocol $\Pi^{\mathsf{M}}_{\mathsf{dp}}$ proceeds similar to the semi-honest variant Π_{dp} (Fig. 7.15). During the preprocessing phase, parties in $\mathcal{E}$ obtain ${}^{\mathcal{E}}\langle\cdot\rangle$-shares of $\Lambda_{\mathbf{x}\odot\mathbf{y}} = \sum_{k=1}^{\mathsf{n}} \alpha_{\mathsf{x}_k}\alpha_{\mathsf{y}_k}$ and $\alpha_{\mathsf{x}_k}, \alpha_{\mathsf{y}_k}$ for $k \in \{1, \ldots, \mathsf{n}\}$. Although the latter two can be computed by parties locally with an invocation of $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$ (Fig. 7.6), computation of the former differs significantly from the semi-honest protocol. For this, we extend the ideas from [136] and generate $[\Lambda_{\mathbf{x}\odot\mathbf{y}}]$, by executing a maliciously secure dot product protocol Π_{dotPre} over $[\cdot]$-shares (abstracted as a functionality $\mathcal{F}_{\mathsf{DotPPre}}$ in Fig. 7.26).

[1]This can be optimized similar to the online phase of the multiplication protocol, where the value is first reconstructed towards P_{king} who sends the reconstructed value to all, followed by verifying its correctness via the verification check.

Specifically, parties invoke Π_{dotPre} on $[\cdot]$-shares of $\boldsymbol{\alpha}_{\mathbf{x}} = (\alpha_{\mathsf{x}_1}, \ldots, \alpha_{\mathsf{x}_n})$ and $\boldsymbol{\alpha}_{\mathbf{y}} = (\alpha_{\mathsf{y}_1}, \ldots, \alpha_{\mathsf{y}_n})$ to compute $[\Lambda_{\mathbf{x}\odot\mathbf{y}}]$, followed by an invocation of $\Pi_{[\cdot]\to^{\mathcal{E}}\langle\cdot\rangle}$ to obtain ${}^{\mathcal{E}}\langle\Lambda_{\mathbf{x}\odot\mathbf{y}}\rangle$. Having computed the necessary preprocessing data, the online phase proceeds similarly to the semi-honest protocol Π_{dp} (Fig. 7.15), where parties reconstruct $\mathsf{z} - \mathsf{r}$ via P_{king} as per equation (7.3). To account for misbehaviour, the protocol is augmented with a verification phase similar to that in malicious multiplication.

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

$\mathcal{F}_{\mathsf{DotPPre}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}^{\mathsf{mal}}$. Let $\mathcal{T}_i$ be the set of the honest parties.

Input: $\mathcal{F}_{\mathsf{DotPPre}}$ receives the $[\cdot]$-shares of the vectors $\mathbf{a} = (\mathsf{a}_1, \ldots, \mathsf{a}_n)$ and $\mathbf{b} = (\mathsf{b}_1, \ldots, \mathsf{b}_n)$ from the parties. $\mathcal{F}_{\mathsf{DotPPre}}$ also receives $[\cdot]$-shares of $\mathsf{z} = \mathbf{a} \odot \mathbf{b}$ of corrupt parties from $\mathcal{S}^{\mathsf{mal}}$. $\mathcal{S}^{\mathsf{mal}}$ is also allowed to send a special command, $(\mathtt{abort}, \mathsf{P})$, which indicates that parties in $\mathcal{P}$ with indices in P should `abort`.

$\mathcal{F}_{\mathsf{DotPPre}}$ proceeds as follows.

– Reconstruct $\mathsf{a}_k, \mathsf{b}_k$ for $k \in \{1, \ldots, \mathsf{n}\}$ using the shares received from honest parties and compute $\mathsf{z} = \sum_{k=1}^{\mathsf{n}} \mathsf{a}_k \cdot \mathsf{b}_k$.

– Compute the $[\cdot]$-share of z to be held by the set of honest parties as the difference between z and the sum of $[\cdot]$-shares of z received from corrupt parties.

– Let y_s denote the $[\cdot]$-shares of z for party $P_s \in \mathcal{P}$. If received $(\mathtt{abort}, \mathsf{P})$ from $\mathcal{S}^{\mathsf{mal}}$, set $\mathsf{y}_s = \mathtt{abort}$ for P_s, where $s \in \mathsf{P}$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Figure 7.26: Ideal functionality for Π_{dotPre}.

Observe that a trivial realization of $\mathcal{F}_{\mathsf{DotPPre}}$ can be reduced to n instances of multiplication. However, we extend the ideas from [30, 31, 136] and rely on a distributed zero-knowledge proof [31] to eliminate the vector-size dependency in the preprocessing phase. Concretely, we instantiate $\mathcal{F}_{\mathsf{DotPPre}}$ using a semi-honest dot product protocol [93] whose cost matches that of semi-honest multiplication [62] (and thus is independent of the vector-size), followed by a verification phase to verify the correctness of the dot product computation. For the verification, we extend the verification technique for multiplication in [31], to now verify the correctness of the dot product, such that the cost due to verification can be amortized away for multiple dot products, thereby resulting in vector-size independent preprocessing.

Elaborately, the semi-honest dot product protocol in [93] takes as input $[\boldsymbol{x}], [\boldsymbol{y}]$ where $\boldsymbol{x}, \boldsymbol{y}$

are vectors of size n, and outputs $[\mathsf{z}] = [\boldsymbol{x} \odot \boldsymbol{y}]$. For this, parties invoke $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$ on each element in $\boldsymbol{x}, \boldsymbol{y}$ and sum these up to generate $\langle\rho\rangle = \langle\boldsymbol{x} \odot \boldsymbol{y}\rangle$. These shares are randomized by summing with $\langle\mathsf{r}\rangle$ (converted from $[\mathsf{r}]$) for a random r, and the sum $\mathsf{z} + \mathsf{r} = (\boldsymbol{x} \odot \boldsymbol{y}) + \mathsf{r}$ is reconstructed towards P_{king}, who sends the reconstructed $\mathsf{z} + \mathsf{r}$ to parties in $\mathcal{E}$. All parties then non-interactively generate $[\mathsf{z} + \mathsf{r}]$ by setting one of its shares as $\mathsf{z} + \mathsf{r}$ and the others as 0. Given $[\mathsf{z} + \mathsf{r}], [\mathsf{r}]$, parties can compute $[\mathsf{z}] = [\mathsf{z} + \mathsf{r}] - [\mathsf{r}]$. Observe that communication of $\langle\mathsf{z}+\mathsf{r}\rangle$ to P_{king} requires $2t$ elements, while communicating $\mathsf{z} + \mathsf{r}$ to parties in $\mathcal{E}$ requires t elements, resulting in a matching cost of $3t$ elements as that required for semi-honest multiplication [62]. The correctness of m dot product triples $(\boldsymbol{x_1}, \boldsymbol{y_1}, \mathsf{z}_1), \ldots, (\boldsymbol{x_m}, \boldsymbol{y_m}, \mathsf{z}_m)$, can be verified by taking a random linear combination,

$$\beta = \sum_{k=1}^{m} \theta_k \cdot \left(\mathsf{z}_k - \sum_{j=1}^{\mathsf{n}} \mathsf{x}_{kj} \cdot \mathsf{y}_{kj} \right)$$

where $\{\theta_k\}_{k=1}^m$ is randomly chosen by all the parties and checking if $\beta = 0$. Given $[\cdot]$-shares of $\boldsymbol{x}_k, \boldsymbol{y}_k, \mathsf{z}_k$ for $k \in \{1, \ldots, m\}$, parties can compute an additive share ($\langle\cdot\rangle$-share) of β by invoking $\Pi_{[\cdot]\cdot[\cdot]\to\langle\cdot\rangle}$. However, since $\langle\cdot\rangle$-sharing does not allow for robust reconstruction, the approach is to generate $[\beta]$ and then robustly reconstruct it and check equality with 0. To generate $[\beta]$, parties first $[\cdot]$-share (via $\Pi_{[\cdot]}$, §7.3.4) their $\langle\cdot\rangle$-share of

$$\psi = \sum_{k=1}^{m} \theta_k \cdot \sum_{j=1}^{\mathsf{n}} \mathsf{x}_{kj} \cdot \mathsf{y}_{kj}.$$

Let ψ^i denote the $\langle\cdot\rangle$-share of ψ held by P_i. Given $[\psi^i]$ for $i \in \{1, \ldots, n\}$, parties can compute

$$[\beta] = \sum_{k=1}^{m} \theta_k \cdot [\mathsf{z}_k] - \sum_{i=1}^{n} [\psi^i]$$

and reconstruct β. It is, however, required to ensure that every party P_i $[\cdot]$-shares the correct ψ^i. To check the correctness of ψ^i, parties need to verify if

$$\psi^i - \sum_{k=1}^{m} \theta_k \left(\sum_{j=1}^{\mathsf{n}} \mathsf{x}^i_{kj} \cdot \mathsf{y}^i_{kj} \right) = 0 \tag{7.4}$$

where $\mathsf{x}^i_{kj}, \mathsf{y}^i_{kj}$ denote the $[\cdot]$-share of $\mathsf{x}_{kj}, \mathsf{y}_{kj}$ held by P_i. Note that following along the lines of $\Pi_{\cdot\to[\![\cdot]\!]}$, parties can generate these $[\cdot]$-share of $\mathsf{x}^i_{kj}, \mathsf{y}^i_{kj}$ from $[\cdot]$-shares of $\mathsf{x}_{kj}, \mathsf{y}_{kj}$, non-interactively. Now, setting $\mathsf{a}_{kj} = \theta_k \mathsf{x}^i_{kj}, \mathsf{b}_{kj} = \mathsf{y}^i_{kj}, \mathsf{c} = \psi^i$, for $k \in \{1, \ldots, m\}$, Equation (7.4), can re-written

as

$$\mathsf{c} - \sum_{k=1}^{m} \sum_{j=1}^{\mathsf{n}} \mathsf{a}_{kj}\mathsf{b}_{kj} = 0 \implies \mathsf{c} - \sum_{l=1}^{m\mathsf{n}} \tilde{\mathsf{a}}_l \tilde{\mathsf{b}}_l = 0 \tag{7.5}$$

The correctness of Equation (7.5) can be verified by invoking $\mathcal{F}^{\mathtt{abort}}_{\mathsf{proveDeg2Rel}}$ (see section 3 of [31] for the definition and its instantiation), which takes as input $[\cdot]$-shares of $\tilde{\mathsf{a}}_l, \tilde{\mathsf{b}}_l, \mathsf{c}$ for $l \in \{1, \ldots, m\mathsf{n}\}$, which are known in clear to party P_i, and verifies if Equation (7.5) holds. The protocol realizing $\mathcal{F}^{\mathtt{abort}}_{\mathsf{proveDeg2Rel}}$ for all n parties requires communicating $\mathcal{O}(n \log(m\mathsf{n}) + n)$ extended ring elements per party. Further, since steps other than $\mathcal{F}^{\mathtt{abort}}_{\mathsf{proveDeg2Rel}}$ require sharing and reconstructing one element, it adds a small constant cost, resulting in the communication cost for verifying m dot products for vector size n being $\mathcal{O}(n \log(m\mathsf{n}) + n)$ extended ring elements per party.

7.5.7 Multi input multiplication

This protocol is similar to its semi-honest counterpart with the difference that the preprocessing phase relies on invoking $\mathcal{F}_{\mathsf{MulPre}}$ for generating the required multiplicative terms. At a high level, the malicious variant of the multi-input multiplication protocol can be viewed as an amalgamation of the semi-honest multi-input multiplication and the malicious multiplication protocol. For the case of 3-input multiplication, recall that the semi-honest protocol to compute $[\![\mathsf{z}]\!]$ given $[\![\mathsf{a}]\!], [\![\mathsf{b}]\!]$ and $[\![\mathsf{c}]\!]$ where $\mathsf{z} = \mathsf{abc}$ requires parties to obtain ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{ab}}\rangle, {}^{\mathcal{E}}\langle\Lambda_{\mathsf{ac}}\rangle, {}^{\mathcal{E}}\langle\Lambda_{\mathsf{bc}}\rangle$ and ${}^{\mathcal{E}}\langle\Lambda_{\mathsf{abc}}\rangle$ in the preprocessing phase, which is then used to reconstruct β_{z} in the online phase.

Since parties in $\mathcal{E}$ are required to hold the correct ${}^{\mathcal{E}}\langle\cdot\rangle$-sharings before the online phase begins, as in the case of multiplication, the techniques from the semi-honest protocol fail in this setting. Hence, our protocol uses 4 instances of $\mathcal{F}_{\mathsf{MulPre}}$ in the preprocessing phase, one each to compute $[\Lambda_{\mathsf{ab}}], [\Lambda_{\mathsf{ac}}], [\Lambda_{\mathsf{bc}}]$ and $[\Lambda_{\mathsf{abc}}]$. Each of the $[\cdot]$-sharing is further converted to ${}^{\mathcal{E}}\langle\cdot\rangle$-sharing using $\Pi_{[\cdot]\to{}^{\mathcal{E}}\langle\cdot\rangle}$ to ensure active participation of only $t+1$ parties in the online phase for the reconstruction of $\mathsf{z} - \mathsf{r}$. Further, to detect malicious behaviour during the reconstruction of $\mathsf{z} - \mathsf{r}$, a verification check similar to the multiplication protocol is performed such that parties `abort` if the check fails. For 4-input multiplication, parties obtain $[\![\cdot]\!]$-sharing of $\mathsf{z} = \mathsf{abcd}$ using $\mathsf{z} - \mathsf{r} = (\beta_{\mathsf{a}} - \alpha_{\mathsf{a}})(\beta_{\mathsf{b}} - \alpha_{\mathsf{b}})(\beta_{\mathsf{c}} - \alpha_{\mathsf{c}})(\beta_{\mathsf{d}} - \alpha_{\mathsf{d}}) - \mathsf{r}$. The protocol proceeds in a similar manner as the 3-input case by delegating the computation of product terms to the preprocessing phase.

7.6 Building blocks

For completeness, we discuss the building blocks used in our framework. These blocks are known from the literature [138, 194], and we show how these can be extended to our setting.

7.6.1 Semi-honest building blocks

7.6.1.1 Bit to arithmetic

Given Boolean shares $[\![\mathsf{b}]\!]^{\mathbf{B}}$ of bit b, protocol Π_{Bit2A} generates its arithmetic shares, $[\![\mathsf{b}^{\mathsf{R}}]\!]$ over $\mathbb{Z}_{2^\ell}$ (Fig. 7.27). Here, b^{R} denotes the arithmetic value of b over the ring $\mathbb{Z}_{2^\ell}$. The approach is to generate a randomized version, $\zeta = \mathsf{b} \oplus \mathsf{r}$ of b, and then recover arithmetic shares of b by performing the arithmetic equivalent of XOR of $\mathsf{b} = \zeta \oplus \mathsf{r}$. Specifically, the arithmetic equivalent of $\mathsf{x} \oplus \mathsf{y}$ is given as $\mathsf{x}^{\mathsf{R}} + \mathsf{y}^{\mathsf{R}} - 2\mathsf{x}^{\mathsf{R}}\mathsf{y}^{\mathsf{R}}$.

Protocol $\Pi_{\mathsf{Bit2A}}(\mathcal{P}, [\![\mathsf{b}]\!]^{\mathbf{B}})$

Preprocessing:

1. Invoke Π_{dsBits} to generate $[\![\mathsf{r}^{\mathsf{R}}]\!], [\![\mathsf{r}]\!]^{\mathbf{B}}$ for $\mathsf{r} \in \mathbb{Z}_2$.
2. Invoke the preprocessing phase of Π_{Mul}.

Online:

1. Compute $[\![\zeta]\!]^{\mathbf{B}} = [\![\mathsf{b}]\!]^{\mathbf{B}} \oplus [\![\mathsf{r}]\!]^{\mathbf{B}}$.
2. $P_i \in \mathcal{E}$ invokes $\Pi^{\mathbf{B}}_{[\![\cdot]\!] \to {}^{\mathcal{E}}\langle\cdot\rangle}$ to generate ${}^{\mathcal{E}}\langle\zeta\rangle^{\mathbf{B}}_i$ and sends ${}^{\mathcal{E}}\langle\zeta\rangle^{\mathbf{B}}_i$ to P_{king}, who reconstructs ζ and generates $[\![\zeta^{\mathsf{R}}]\!]$.
3. Invoke the online phase of Π_{Mul} to generate $[\![\zeta^{\mathsf{R}}\mathsf{r}^{\mathsf{R}}]\!]$, and compute $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\zeta^{\mathsf{R}}]\!] + [\![\mathsf{r}^{\mathsf{R}}]\!] - 2[\![\zeta^{\mathsf{R}}\mathsf{r}^{\mathsf{R}}]\!]$.

Figure 7.27: Semi-honest: Bit to arithmetic.

7.6.1.2 Bit injection

This protocol, denoted as Π_{BitInj}, facilitates generation of $[\![\mathsf{b}^{\mathsf{R}} \cdot \mathsf{v}]\!]$ given $[\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{v}]\!]$ for $\mathsf{b} \in \mathbb{Z}_2$ and $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. As seen in [138],

$$\mathsf{b}^{\mathsf{R}}\mathsf{v} = (\beta_{\mathsf{b}} \oplus \alpha_{\mathsf{b}})^{\mathsf{R}}(\beta_{\mathsf{v}} - \alpha_{\mathsf{v}}) = {\beta_{\mathsf{b}}}^{\mathsf{R}}\beta_{\mathsf{v}} - {\beta_{\mathsf{b}}}^{\mathsf{R}}\alpha_{\mathsf{v}} + (2{\beta_{\mathsf{b}}}^{\mathsf{R}} - 1)({\alpha_{\mathsf{b}}}^{\mathsf{R}}\alpha_{\mathsf{v}} - \beta_{\mathsf{v}}{\alpha_{\mathsf{b}}}^{\mathsf{R}})$$

Given ${}^{\mathcal{E}}\langle\cdot\rangle$-shares of $\alpha_{\mathsf{v}}, {\alpha_{\mathsf{b}}}^{\mathsf{R}}, {\alpha_{\mathsf{b}}}^{\mathsf{R}}\alpha_{\mathsf{v}}, \mathsf{r}$, together with $[\![\mathsf{r}]\!]$ where $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, and the knowledge that $\beta_{\mathsf{v}}, {\beta_{\mathsf{b}}}^{\mathsf{R}}$ is held by all parties in $\mathcal{E}$, parties can non-interactively compute ${}^{\mathcal{E}}\langle\mathsf{b}^{\mathsf{R}}\mathsf{v} + \mathsf{r}\rangle$, reconstruct it via P_{king} and generate $[\![\mathsf{b}^{\mathsf{R}}\mathsf{v} + \mathsf{r}]\!]$. $[\![\mathsf{b}^{\mathsf{R}}\mathsf{v}]\!]$ can then be computed as $[\![\mathsf{b}^{\mathsf{R}}\mathsf{v}]\!] = [\![\mathsf{b}^{\mathsf{R}}\mathsf{v} + \mathsf{r}]\!] - [\![\mathsf{r}]\!]$. To facilitate this, in the preprocessing phase parties generate ${}^{\mathcal{E}}\langle\cdot\rangle$-shares of $\mathsf{r}, \alpha_{\mathsf{v}}, {\alpha_{\mathsf{b}}}^{\mathsf{R}}, {\alpha_{\mathsf{b}}}^{\mathsf{R}}\alpha_{\mathsf{v}}$, and $[\![\mathsf{r}]\!]$. Here, ${}^{\mathcal{E}}\langle\mathsf{r}\rangle$, ${}^{\mathcal{E}}\langle\alpha_{\mathsf{v}}\rangle$ and $[\![\mathsf{r}]\!]$ are generated as in the preprocessing of multiplication, and ${}^{\mathcal{E}}\langle{\alpha_{\mathsf{b}}}^{\mathsf{R}}\rangle$ is generated via Π_{Bit2A} followed by invoking $\Pi_{[\![\cdot]\!] \to {}^{\mathcal{E}}\langle\cdot\rangle}$. Following this, ${}^{\mathcal{E}}\langle{\alpha_{\mathsf{b}}}^{\mathsf{R}}\alpha_{\mathsf{v}}\rangle$ is generated as done in the preprocessing of multiplication.

7.6.1.3 Arithmetic to Boolean sharing

Extending the techniques from [138], protocol Π_{A2B} generates $[\![\mathsf{x}]\!]^{\mathbf{B}}$ from $[\![\mathsf{x}]\!]$ for $\mathsf{x} \in \mathbb{Z}_{2^\ell}$. For this, given arithmetic and Boolean shares of $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, Boolean shares of x are computed as $(\mathsf{x}+\mathsf{r}) - \mathsf{r}$ by evaluating a parallel prefix adder (PPA) circuit [194, 173]. The PPA circuit takes as input two Boolean values ($\mathsf{x}+\mathsf{r}$, $-\mathsf{r}$ in this case) and outputs their sum. The protocol appears in Fig. 7.28. Looking ahead, Π_{A2B} is used in the preprocessing phase in the applications considered. Hence, we rely on the PPA circuit from [173] as it provides a good trade-off between rounds and communication as opposed to the circuit from [194] which is optimized to provide a fast online phase at the expense of a higher preprocessing cost (yielding a higher total cost than [173]).

Protocol $\Pi_{\mathsf{A2B}}(\mathcal{P}, [\![\mathsf{x}]\!])$

Preprocessing:

1. Invoke $\Pi_{\mathsf{dsBits}}(\mathcal{P}, 0)$ to generate $[\![(\mathsf{r}[i])^{\mathsf{R}}]\!]$ and $[\![\mathsf{r}[i]]\!]^{\mathbf{B}}$ where $\mathsf{r}[i] \in \mathbb{Z}_2$ for $i \in \{0, \ldots, \ell-1\}$, and set $[\![\mathsf{r}]\!] = \sum_{i=0}^{\ell-1} 2^i [\![(\mathsf{r}[i])^{\mathsf{R}}]\!]$.
2. Execute the preprocessing phase for the PPA circuit which computes $[\![\mathsf{x}]\!]^{\mathbf{B}} = [\![\mathsf{x}+\mathsf{r}]\!]^{\mathbf{B}} - [\![\mathsf{r}]\!]^{\mathbf{B}}$.

Online:

1. Compute $[\![\mathsf{x}+\mathsf{r}]\!] = [\![\mathsf{x}]\!] + [\![\mathsf{r}]\!]$
2. Parties in $\mathcal{E}$ invoke $\Pi_{[\cdot]\to\mathcal{E}\langle\cdot\rangle}$ on $[\![\mathsf{x}+\mathsf{r}]\!]$ to generate ${}^{\mathcal{E}}\langle\mathsf{x}+\mathsf{r}\rangle$ and send their share to P_{king}.
3. P_{king} reconstructs and sends $\mathsf{x}+\mathsf{r}$ to all parties in $\mathcal{E}$.
4. Invoke $\Pi^{\mathbf{B}}_{\cdot\to[\![\cdot]\!]}$ to generate $[\![\mathsf{x}+\mathsf{r}]\!]^{\mathbf{B}}$, and execute the online phase of the PPA circuit to compute $[\![\mathsf{x}]\!]^{\mathbf{B}} = [\![\mathsf{x}+\mathsf{r}]\!]^{\mathbf{B}} - [\![\mathsf{r}]\!]^{\mathbf{B}}$.

Figure 7.28: Semi-honest: Arithmetic to Boolean.

7.6.1.4 Boolean to arithmetic sharing

This protocol generates $[\![\mathsf{x}]\!]$ from $[\![\mathsf{x}]\!]^{\mathbf{B}}$ where $\mathsf{x} \in \mathbb{Z}_{2^\ell}$. Inspired from [138, 136], observe that $\mathsf{x} = \sum_{i=0}^{\ell-1} 2^i (\mathsf{x}[i])^{\mathsf{R}}$. Thus, we invoke Π_{Bit2A} on $\mathsf{x}[i]$ for $i \in \{0, \ldots, \ell-1\}$ to generate $[\![\mathsf{x}[i]^{\mathsf{R}}]\!]$ followed by locally combining it as per the above equation to generate $[\![\mathsf{x}]\!]$. Optimizations in [138] carry forward to our setting as well.

7.6.1.5 Comparison

To compare $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ in FPA, we extend the technique of [173, 193, 136, 50, 138, 194], where checking $\mathsf{x} < \mathsf{y}$ is equivalent to checking if the most significant bit (msb) of $\mathsf{v} = \mathsf{x} - \mathsf{y}$ is 1. To

extract the msb from $[\![\mathsf{v}]\!]$, we rely on Π_{Bitext} which takes as input $[\![\mathsf{v}]\!]$ and outputs the $[\![\cdot]\!]^{\mathbf{B}}$-share of the msb of v, denoted as $[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}}$. The optimized bit extraction circuit from [194] is used for computing the msb whose inputs are two $[\![\cdot]\!]^{\mathbf{B}}$-shared values and output is the $[\![\cdot]\!]^{\mathbf{B}}$-shared msb of the sum of these two inputs. Observe that, given $[\![\mathsf{v}]\!]$, v can be written as $\mathsf{v} = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}}$, and hence $[\![\cdot]\!]^{\mathbf{B}}$-shares of β_{v} and α_{v} constitute the two inputs to the circuit. While $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$ can be generated non-interactively by invoking $\Pi^{\mathbf{B}}_{\cdot\to[\![\cdot]\!]}$ in the online phase, $[\![\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$ is generated by performing arithmetic to boolean conversion in the preprocessing phase. Evaluation of the bit extraction circuit then gives $[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}}$.

7.6.1.6 Equality check

Given $[\![\cdot]\!]$-shared $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$, this protocol outputs a $[\![\cdot]\!]^{\mathbf{B}}$-shared bit, which is set to 1 if $\mathsf{x} = \mathsf{y}$, and 0 otherwise. The approach is to obtain the bit decomposition of $\mathsf{v} = \mathsf{x} - \mathsf{y}$ by performing Π_{A2B}, and checking if all bits of v are 0. For this, parties non-interactively obtain 1's complement of the bits of v, denoted as $\bar{\mathsf{v}}$, by setting the corresponding $\beta_{\bar{\mathsf{v}}} = 1 \oplus \beta_{\mathsf{v}}$ and $\alpha_{\bar{\mathsf{v}}} = \alpha_{\mathsf{v}}$. Parties proceed to compute an AND of all the bits in $\bar{\mathsf{v}}$ following the standard tree-based approach where we use the 4-input multiplication to save on rounds and communication. If $\mathsf{v} = 0$, then the AND outputs 1 else it outputs a 0. The protocol appears in Fig. 7.29.

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

Preprocessing:

1. Perform preprocessing phase of Π_{A2B} and the preprocessing of $\Pi_{\mathsf{4\text{-}Mul}}$.

Online:

1. Compute $[\![\mathsf{v}]\!] = [\![\mathsf{x}]\!] - [\![\mathsf{y}]\!]$ and invoke Π_{A2B} to generate $[\![\mathsf{v}]\!]^{\mathbf{B}}$.
2. Generate $[\![\bar{\mathsf{v}}]\!]^{\mathbf{B}}$ by setting $\beta_{\bar{\mathsf{v}}} = 1 \oplus \beta_{\mathsf{v}}$ and $\alpha_{\bar{\mathsf{v}}} = \alpha_{\mathsf{v}}$.
3. Set $[\![\mathsf{b}]\!]^{\mathbf{B}} = \wedge_{i=1}^{\ell} \bar{\mathsf{v}}_i$ by relying on a tree-based approach for computing AND and online phase of $\Pi_{\mathsf{4\text{-}Mul}}$.

Figure 7.29: Semi-honest: Equality check protocol.

7.6.1.7 Maxpool / Minpool

Maxpool allows parties to compute $[\![\cdot]\!]$-share of the maximum value $\mathsf{x}_{\max}$ among a vector of values $\boldsymbol{x} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$. For this, we proceed along the lines of [138]. Observe that the maximum among two values $\mathsf{x}_i, \mathsf{x}_j$ can be computed by first using the secure comparison protocol to obtain $[\![\mathsf{b}]\!]^{\mathbf{B}}$ such that $\mathsf{b} = 0$ if $\mathsf{x}_i \geq \mathsf{x}_j$ and 1 otherwise. Following this, parties can compute $\mathsf{b}(\mathsf{x}_j - \mathsf{x}_i) + \mathsf{x}_i$ using the bit injection protocol, to obtain the maximum value as the output. To compute the

maximum among a vector of values, parties follow the standard binary tree-based approach where consecutive pairs of values are compared in a level-by-level manner. We refer to the resulting protocol as Π_{max}. A protocol Π_{min} for minpool can be worked out similarly.

7.6.1.8 ReLU

The ReLU function, $\mathsf{ReLU}(\mathsf{v}) = \mathsf{max}(0, \mathsf{v})$, can be written as $\mathsf{ReLU}(\mathsf{v}) = \overline{\mathsf{b}} \cdot \mathsf{v}$, where bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. Here $\overline{\mathsf{b}}$ denotes the complement of b. Given $[\![\mathsf{v}]\!]$, parties invoke Π_{Bitext} on $[\![\mathsf{v}]\!]$ to obtain $[\![\mathsf{b}]\!]^{\mathbf{B}}$. The $[\![\cdot]\!]^{\mathbf{B}}$-sharing of $\overline{\mathsf{b}}$ is then computed, non-interactively, by setting $\beta_{\overline{\mathsf{b}}} = 1 \oplus \beta_{\mathsf{b}}$. Given $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$, ReLU can be computed using Π_{BitInj}.

7.6.2 Malicious building blocks

Note that the malicious variants for the building blocks, such as bit to arithmetic, Boolean to arithmetic, and arithmetic to Boolean conversion, bit extraction, secure comparison, secure equality check, ReLU, maxpool, and convolutions, follow along similar lines to that of the semi-honest protocols with the difference that the underlying protocols used are replaced with their maliciously secure variants. Moreover, for steps that involve opening values via P_{king}, the reconstructed values are sent to all and are accompanied by a verification check similar to the one in the multiplication protocol.

7.6.3 Communication cost

Table 7.4 summarises the communication cost and online round complexity of the semi-honest and maliciously secure protocols.

7.7 Applications & benchmarks

To evaluate the performance of our protocols, we benchmark some of the popular applications such as deep neural networks (NN), similar sequence queries (SSQ), and biometric matching where MPC is used to achieve privacy. While these applications have been looked at in the small party setting [174, 136, 208, 15, 204, 220, 194, 173], we believe the n-party setting is a better fit for reasons described in the introduction. To the best of our knowledge, we are the first to benchmark these in the multiparty honest-majority setting for more than four parties.

Building Block	Semi-honest			Malicious		
	Communication		Rounds Online	Communication		Rounds Online
	Preprocessing	Online		Preprocessing	Online	
Sharing	–	$(t+1)\ell$	1	–	$2t\ell$	1
Reconstruction[a]	–	$3t\ell$	2	–	$n(\mathsf{q}-\mathsf{g})\ell$	1
Multiplication	$t\ell$	$2t\ell$	2	$3t\ell$	$3t\ell$	2
3-input multiplication	$6t\ell$	$2t\ell$	2	$12t\ell$	$3t\ell$	2
4-input multiplication	$15t\ell$	$2t\ell$	2	$33t\ell$	$3t\ell$	2
Doubly shared bits	$4t(\ell+2)$	–	–	$6t(\ell+2)$	–	–
Multiplication with truncation	$4t(\ell+2)\ell+t\ell$	$2t\ell$	2	$3t\ell+6t(\ell+2)\ell$	$3t\ell$	2
Dot product	$t\ell$	$2t\ell$	2	$3t\ell$	$3t\ell$	2
Bit to arithmetic	$4t(\ell+2)+t\ell$	$4t\ell$	4	$6t(\ell+2)+3t\ell$	$6t\ell$	4
Bit injection	$4t(\ell+2)+6t\ell$	$2t\ell$	2	$6t(\ell+2)+12t\ell$	$3t\ell$	2
Arithmetic to Boolean	$4t(\ell+2)\ell$ $+t\ell\log_2\ell$	$2t\ell(1+\log_2\ell)$	$2+2\log_2\ell$	$6t(\ell+2)\ell$ $+3t\ell\log_2\ell$	$3t\ell(1+\log_2\ell)$	$2+2\log_2\ell$
Boolean to arithmetic	$4t(\ell+2)\ell$	$2t\ell$	2	$6t(\ell+2)\ell$	$3t\ell$	2
Comparison[b]	$\mathsf{u}_1+4t(\ell+2)\ell+$ $3t\ell\log_2\ell+2t\ell$	$2t\mathsf{u}_2$	$2\log_4\ell$	$6t(\ell+2)\ell+6t\ell\log_2\ell$ $+3t\ell+\mathsf{u}_1$	$3t\mathsf{u}_2$	$2\log_4\ell$

ℓ - size of ring in bits.
[a]Accounts for reconstruction towards all; $\mathsf{q}=\binom{n}{h}$, $\mathsf{g}=\binom{n-1}{h-1}$. [b]$\mathsf{u}_1=3t\mathsf{n}_2+12t\mathsf{n}_3+33t\mathsf{n}_4$, $\mathsf{u}_2=\mathsf{n}_2+\mathsf{n}_3+\mathsf{n}_4$, $\mathsf{n}_2=41,\mathsf{n}_3=27,\mathsf{n}_4=47$ denote the number of AND gates in the bit extraction circuit of ABY2 [194] with $2,3,4$ inputs, respectively.

Table 7.4: Communication and round complexity of protocols: semi-honest and malicious.

Benchmark environment

The performance of our protocols is analyzed using a prototype implementation building over the ENCRYPTO library [59] in C++17. We chose 64 bit ring ($\mathbb{Z}_{2^{64}}$) for our arithmetic world, and the operations over extended ring were carried out using the NTL library[2]. Since the correctness and accuracy of the applications considered in the secure computation setting are already established, our benchmark aims to demonstrate our protocols' performance and is not fully functional. Moreover, we believe that incorporating state-of-the-art code optimizations like GPU-assisted computing can enhance the efficiency of our protocols, which is left as future work. Since there is no defined way to capture an adversary's misbehaviour, following standard practice [173, 136, 61], we benchmark honest executions of the protocols, which also include the steps performed for verification in the malicious case. We use multi-threading, wherever possible, to facilitate efficient computation and communication among the parties. The parties in the computation are emulated using Google Cloud (n1-standard-64 instances, 2.0 GHz Intel Xeon Skylake, 64 vCPUs, 240 GB RAM) with machines located in East Australia, South Asia, South East Asia, and West Europe. All our experiments are run for 5, 7, and 9 parties, each. We would like to note that our protocols can be scaled to a larger number of parties. However,

[2]https://libntl.org

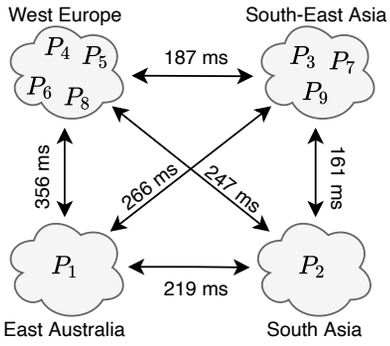


Figure 7.30: Round trip time (rtt).

recall that reliance on RSS will result in increasing the share size with an increasing number of parties. Further, we note that the performance of our semi-honest protocol in the special setting of $t = 1$ is on par with tailor-made protocols such as [49]. For the malicious setting, note that customized protocols with $t = 1$ such as [136, 50, 138] are tailor-made for their setting and hence are more efficient. Specifically, they benefit from a single online round of interaction per multiplication gate, as opposed to two in our case, while having the same communication cost. We estimate this will roughly double the latency of our malicious protocol in this setting of $t = 1$. While a single-round protocol can be designed for the multiparty case, the two-round protocol ensures that communication complexity remains linear in the number of parties, as opposed to quadratic in the former. Since our focus is on attaining protocols that tolerate $t > 1$, we omit to provide performance comparisons for these customized protocols in the $t = 1$ setting.

Benchmark parameters

We report the run-time and communication of the online phase and total (= preprocessing + online). Note that the reported costs only consider the evaluation phase and do not account for the cost of input sharing and output reconstruction phases (because the latter phases amount to a one-time cost). Hence, for the malicious setting, the reported numbers do not account for the cost of broadcast required for the fair reconstruction. To capture the effect of online round complexity and communication in one go, we also report the throughput (TP [11, 173, 136]) of the online phase. TP denotes the number of operations that can be performed in one minute. Finally, when deployed in the outsourced setting, one pays the price for the communication and up-time of the hired servers. To demonstrate how our protocols fare in this scenario, we additionally report the monetary cost (Cost) [171, 138] for the applications considered. This cost is estimated using Google Cloud Platform [205] pricing, where 1 GB and 1 hour of usage

costs USD 0.08 and USD 3.04, respectively.

7.7.1 Comparison with DN07*

In this section, we benchmark our semi-honest and malicious protocols over synthetic circuits comprising one million multiplications with varying depths of 1, 100, and 1000, and compare them against the optimized ring variant of DN07* [29]. The gates are distributed equally across each level in the circuit.

7.7.1.1 Communication

The communication cost for 1 million multiplications is tabulated in Table 7.5 for the 5, 7, and 9 party settings. As can be observed, the online phase of our semi-honest protocol enjoys the benefits of pushing 33% communication to a preprocessing phase compared to DN07*. The observed values corroborate the claimed improvement in the online complexity of our protocol. Our malicious protocol retains the online communication cost of DN07* while incurring a similar overhead in the preprocessing phase.

Ref.	$n = 5$	$n = 7$	$n = 9$
DN07* (semi)	(0, 45.78)	(0, 68.66)	(0, 91.55)
This (semi)	(15.26, 30.52)	(22.88, 45.78)	(30.51, 61.04)
This (mal)	(45.79, 45.78)	(68.67, 68.67)	(91.57, 91.57)

Table 7.5: Communication (Preprocessing, Online) in MB for 1 million multiplications.

Note that pushing the communication to the preprocessing phase has several benefits. First, communication with respect to several instances can happen in a single shot and leverage the benefit of serialization. Second, with respect to resource-constrained devices such as mobile phones, preprocessing communication can occur whenever they have access to a high-bandwidth Wi-Fi network (for instance, when the device is at home overnight). These benefits facilitate a fast online phase, as observed, that may happen over a low-bandwidth network.

7.7.1.2 Run time

The time taken to evaluate circuits of different depths appears in Table 7.6. Since the time for the 5, 7, and 9 party settings vary within the range [0, 0.5], we report values only for the 7-party setting in Table 7.6. With respect to the online run-time, our semi-honest protocol's time is expected to be similar to that of DN07*. However, DN07* demonstrates around 1.5×

higher run-time. This difference can be attributed to the asymmetry in the rtt among parties, which vanished when benchmarked over a symmetric rtt setting. Compared to the semi-honest protocol, the malicious variant incurs a minimal overhead of less than one second in the online run-time due to the one-time verification phase. However, the overhead is higher for the case of the overall run-time. Concretely, it is around 10 seconds and is due to the distributed zero-knowledge proof computation in the preprocessing phase. Note that this overhead is independent of the circuit depth and gets amortized for deeper circuits as evident from Table 7.6 (depth 1 vs. 1000).

Ref.	$d = 1$	$d = 100$	$d = 1000$
DN07* (semi)	(0, 0.65)	(0, 54.97)	(0, 549.69)
This (semi)	(0.47, 0.45)	(0.47, 30.75)	(0.47, 307.48)
This (mal)	(10.52, 1.36)	(10.53, 68.67)	(10.54, 308.39)

Table 7.6: Latency in seconds (Preprocessing, Online) for varying depth (d) circuits with 1 million multiplications for $n = 7$.

7.7.1.3 Monetary cost

Another key highlight of our protocols is their improved monetary cost, as evident from Fig. 7.31[3]. Concretely, for 9 parties (semi-honest), we observe a saving of 17% over DN07* for a depth-1 circuit, and it increases up to 72% for circuits with depth 1000. This is primarily due to the reduction in the number of online parties over DN07*. Comparing our semi-honest and malicious variants, the latter has an overhead of 8× for a depth-1 circuit, and it reduces to 1.14× for a depth-1000 circuit. This is justified because the verification cost is amortized for deeper circuits, as mentioned earlier. Interestingly, our malicious variant outperforms even the semi-honest DN07* upon reaching circuit depths of 100 and above. A similar analysis holds in the symmetric rtt setting as well, where the saving is up to 56% (for $d = 1000$).

7.7.1.4 Online throughput (TP):

Owing to the asymmetric rtt as described earlier, our semi-honest variant witnesses up to 1.78× improvements in TP (for a single execution) over DN07*, which vanishes in the symmetric rtt setting. However, recall that our protocol requires only $t + 1$ active parties in the online phase,

[3]Bars in solid colours denote computation over network given in Fig. 7.30, while the area represented via crosshatch pattern denotes the additional cost incurred in the symmetric rtt setting (356 ms).

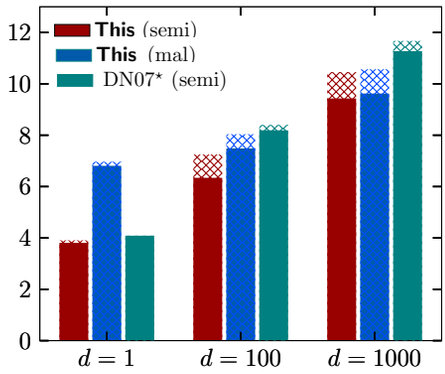


Figure 7.31: Monetary cost (in USD) for evaluating circuits (1000 instances) of various depths (d) for $n = 9$ parties. The values are reported in $\log_2$ scale.

which leaves several channels among the parties underutilized. Hence, we can leverage the load-balancing technique where parties' roles are interchanged across various parallel executions. For instance, one approach is to make every party act as P_{king}, i.e., in 5PC, in one execution, $P_{\mathsf{king}} = P_1, \mathcal{E} = \{P_1, P_2, P_3\}, \mathcal{D} = \{P_4, P_5\}$, while in another execution $P_{\mathsf{king}} = P_2, \mathcal{E} = \{P_2, P_3, P_4\}, \mathcal{D} = \{P_5, P_1\}$, and so on. To analyse the effect of load balancing, we performed experiments with similar rtt among the parties and observed a 1.5× improvement in our semi-honest variant over DN07⋆. This is justified as we communicate over four channels among the parties as opposed to six in DN07⋆. We note that while enhancing the security from semi-honest to malicious, we observe a significant drop in TP, which is about 3× for the depth-1 circuit. This is primarily due to increased run time owing to the verification in the online phase for malicious setting. However, this drop tends to zero for deeper circuits (as verification cost gets amortized), making the online phase of our maliciously secure protocol on par with the semi-honest one.

7.7.2 Deep neural networks (DNN)

We begin by discussing the architectural details of the neural networks under consideration, followed by the benchmarks.

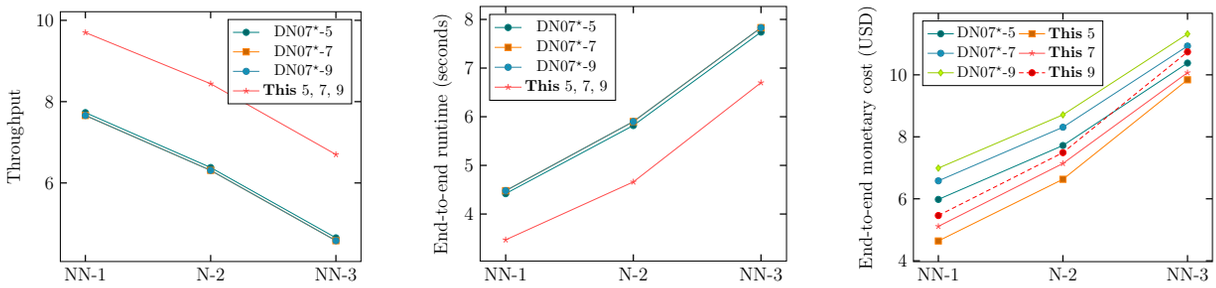
7.7.2.1 Neural network architecture

We benchmark three different neural networks (NN) [173, 193, 220] with an increasing number of parameters—(i) NN-1: a 3-layer fully connected network with ReLU activation after each layer, as considered in [174, 173, 193, 136], (ii) NN-2: the LeNet [147] architecture, which contains two convolutional layers and two fully connected layers with ReLU activation after each layer, and maxpool operation after convolutional layers, and (iii) NN-3: VGG16 [213]

architecture, that comprises 16 layers in total, which includes fully connected, convolutional, ReLU activation, and maxpool layers. The last 2 NNs were considered in [220]. We benchmark the inference phase of the above NNs, which comprises computing activation matrices, followed by applying an activation function or pooling operation, depending on the network architecture. NN-1 and NN-2 are benchmarked over MNIST dataset [146] while NN-3 is benchmarked using CIFAR-10 dataset [141].

7.7.2.2 Analysis

To analyse the improvement of our protocols, we also benchmark (semi-honest) DN07⋆ for the applications by adapting our building blocks to their setting. The semi-honest benchmarks for the different NNs appear in Table 7.7 while the malicious ones appear in Table 7.8. Fig. 7.32 gives a pictorial view of the trends observed while comparing the semi-honest variants and are described next.



(a) Online throughput. (b) End-to-end runtime. (c) Monetary cost (1000 queries).

Figure 7.32: Comparison for deep NN between our semi-honest protocol and DN07⋆ (values plotted are logarithmic in base 2).

We incur a very minimal overhead in the run-time of our protocols when moving from five to nine parties over all the networks considered. Hence, we use $\pm\delta$ to denote this variation in the table. The trends witnessed in synthetic circuit benchmarks (§7.7.1) carry forward to neural networks as well due to reasons discussed previously. For instance, the improvement in the online run-time for our semi-honest variant is up to 4.3× over DN07⋆. The effect of reduced run-time and improved communication results in a significant improvement in the online throughput of our protocol over DN07⋆. Concretely, the gain ranges up to 4.3×. Further, the improved run-time coupled with the reduced number of online parties for our case brings in a saving of up to 69% in monetary cost for NN-1. However, the improvement drops to 33% for deep network NN-3. The reduction in savings is due to improved run-time getting nullified

by increased communication from NN-1 to NN-3, making communication the dominant factor in determining monetary cost.

	Ref.	n	NN-1			NN-2			NN-3		
			Comm	Time	TP[a]	Comm	Time	TP	Comm	Time	TP
Online	DN07⋆	5	0.16	18.55 ±.4	211.69	15.58	46.20	83.12	228.07	152.95	25.11
		7	0.24		202.48	23.39	48.39	79.35	342.24	160.10	23.99
		9	0.33		202.49	31.18	48.40	79.35	456.33	160.14	23.99
	This	5	0.02	4.61 ±.02	832.61 ±.04	1.92	11.08 ±.02	346.60 ±.3	29.70	36.92 ±.02	104.01 ±.04
		7	0.03			2.88			44.55		
		9	0.05			3.84			59.40		
			Comm	Time	Cost[b]	Comm	Time	Cost	Comm	Time	Cost
End-to-end	DN07⋆	5	3.41	21.46	0.06	269.23	56.44	0.21	4288.26	213.77	1.34
		7	5.11	22.29	0.10	403.85	59.60	0.32	6432.39	227.28	1.96
		9	6.81	22.31	0.13	538.47	59.61	0.42	8576.52	227.33	2.56
	This	5	3.41	11.09 ±.02	0.02	269.50	25.34 ±.03	0.10	4292.06	104.09 ±.03	0.91
		7	5.11		0.03	404.25		0.14	6438.09		1.08
		9	6.81		0.04	539.00		0.18	8584.12		1.71

Communication in MB and time in seconds.
[a]TP denotes throughput [b]monetary cost in USD

Table 7.7: Semi-honest: Benchmarks for deep NN.

Observe that, unlike the case in synthetic circuits (Table 7.5), the total communication here is an order of magnitude higher. This is primarily due to the higher communication cost incurred for performing the truncation operation—specifically, the generation of the doubly-shared bits (Π_{dsBits}) in the preprocessing phase. It is worth noting that Π_{dsBits} is used as a black box, and an improved instantiation for it will lower the communication.

	n	NN-1			NN-2			NN-3		
		Comm	Time	TP[a]	Comm	Time	TP	Comm	Time	TP
Online	5	0.04	5.44 ±.02	706.40 ±.04	2.88	11.93 ±.03	322.63 ±.2	44.56	37.91 ±.02	101.27 ±.04
	7	0.06			4.32			66.84		
	9	0.08			5.77			89.12		
		Comm	Time	Cost[b]	Comm	Time	Cost	Comm	Time	Cost
End-to-end	5	3.59	22.96 ±.02	0.07	286.18	37.71 ±.04	0.15	4535.95	124.54	1.04
	7	5.39		0.10	429.28		0.22	6804.06	126.69	1.53
	9	7.20		0.11	571.98		0.27	9066.43	129.42	1.94

Communication in MB and time in seconds.
[a]TP denotes throughput [b]monetary cost in USD

Table 7.8: Malicious: Benchmarks for deep NN.

Compared to our semi-honest variant for evaluating NNs, the malicious variant incurs a 2× higher online communication cost for NN-1 and NN-2. However, this difference closes in with deeper NNs, with the communication being 1.5× for NN-3. The drop in the difference can be attributed to the one-time cost of verification required in the malicious variant, which gets amortized over deeper circuits. Due to the same reason, in comparison to the semi-honest case, the malicious variant has an overhead of around 1 second in the online run-time, which in turn reflects in the reduced throughput. Similar to the semi-honest evaluation of NNs, the overall communication is an order of magnitude higher than the online communication due to the cost incurred for truncation during preprocessing. Also, analogous to the trend observed for synthetic circuits, the overhead in overall run-time is approximately 11 seconds owing to the distributed zero-knowledge proof verification required in the preprocessing phase.

7.7.3 Genome sequence matching

Given a genome sequence as a query, genome matching aims to identify the most similar sequence from a database of sequences. This task is also known as similar sequence query (SSQ). An SSQ algorithm on two sequences s and q, requires the computation of Edit Distance (ED), which quantifies how different two sequences are by identifying the minimum number of additions, deletions, and substitutions needed to transform one sequence to the other. To compute the ED, we extend the (2-party) protocol from [204], which builds on top of the approximation from [15], to the n-party setting. We proceed to describe the high-level idea of the approximation algorithm for ED computation for a query sequence q against a database of sequences $\{\mathsf{s}_1, \ldots, \mathsf{s}_m\}$.

The ED approximation algorithm has a non-interactive phase, during which the database owner with the sequences $\mathsf{s}_1, \ldots, \mathsf{s}_m$, generates a Look-Up-Table (LUT) for each sequence. These LUTs are then secret-shared among all the parties. To generate the LUT, the sequences in the database are aligned with respect to a common reference genome sequence (using the Wagner-Fischer algorithm [221]), and divided into blocks of a fixed, predetermined size. Based on the most frequently occurring block sequences in the database, an LUT is constructed consisting of these block values and their distance from each other. Specifically, for a database of m sequences $\{\mathsf{s}_1, \ldots, \mathsf{s}_m\}$, each of length ω blocks, an LUT_i is constructed for each s_i. Each LUT has m columns, one corresponding to each s_i in the database, and ω rows, one corresponding to each block of a sequence, where $\mathsf{LUT}_{\mathsf{s}}[i][j]$ corresponds to the ED between block i of the sequence s and s_j. This completes the non-interactive phase of the ED approximation algorithm.

Given the LUTs, when a new query q has to be processed, its ED must be computed from

every sequence s in the database. For this, similar to the non-interactive phase, the query is first aligned with the reference sequence and broken down into blocks of the same fixed size. Then, the i^{th} block from the query is matched with the i^{th} block of each sequence in the LUT for a sequence s. If the block values match, then the precomputed distance is taken as the output for that block; otherwise, the output is taken to be 0. Finally, the resultant sum of distances for all the blocks is taken to be the approximated ED between q and the sequence s. Computing the ED to all such sequences s in the database then allows the identification of the most similar sequence for the query using the minpool operation. Algorithms for ED computation between two sequences, and SSQ appear in Fig. 7.33, Fig. 7.34, respectively, where accuracy and correctness follow from [15]. Since the generation of LUTs happens non-interactively, we only focus on the computation of ED with respect to the new query q, which requires interaction, and benchmark the same.

Protocol $\Pi_{\mathsf{ED}}(\mathcal{P}, [\![\mathsf{LUT_s}]\!], [\![\mathsf{q}]\!])$

1. For $i = 1$ to ω
- For $j = 1$ to m
  - Invoke Π_{Eq} on $[\![\mathsf{LUT_s}[i][j]]\!]$ and $[\![\mathsf{q}[i]]\!]$ to generate $[\![\mathsf{b}_j]\!]^{\mathbf{B}}$.
  - Invoke Π_{Bit2A} on $[\![\mathsf{b}_j]\!]^{\mathbf{B}}$ and generate $[\![\mathsf{b}_j]\!]$.
- Let $\mathbf{b} = \{\mathsf{b}_1, \ldots, \mathsf{b}_m\}$. Compute $[\![\mathsf{dist}_i]\!] = \Pi_{\mathsf{dp}}([\![\mathbf{b}]\!], [\![\mathsf{LUT_s}[i][\cdot]]\!])$.

2. Compute $[\![\mathsf{dist}]\!] = \sum_{i=1}^{\omega} [\![\mathsf{dist}_i]\!]$.

Figure 7.33: Edit distance between query q and sequence s with respect to a database of m sequences and ω blocks.

Protocol $\Pi_{\mathsf{SSQ}}(\mathcal{P}, \{[\![\mathsf{LUT_s}]\!]\}_{s=1}^{m}, [\![\mathsf{q}]\!])$

1. For $\mathsf{s} = 1$ to m
   - Invoke Π_{ED} on $[\![\mathsf{LUT_s}]\!]$ and $[\![\mathsf{q}]\!]$ to generate $[\![\mathsf{dist_s}]\!]$.
2. Invoke $\Pi_{\min}$ on $[\![\mathsf{dist}_1]\!], \ldots, [\![\mathsf{dist}_m]\!]$ to generate $[\![\mathsf{s}_{\min}]\!] \in \{\mathsf{dist}_1, \ldots, \mathsf{dist}_m\}$.

Figure 7.34: Similar sequence queries.

The benchmarks for genome sequence matching appear in Table 7.9. Following [204], we consider three cases with different numbers of sequences in the database (m) and different block lengths (ω). We witness similar trends here, where our semi-honest protocol has improvements of up to 4× in both online run-time and throughput over DN07*. Our malicious variant incurs a minimal overhead in the range of 5-6% in online run-time and total communication over the semi-honest counterpart.

	Ref.	n	$m = 1000, \omega = 25$			$m = 2000, \omega = 30$			$m = 4000, \omega = 35$		
			Comm[a]	Time	TP[b]	Comm	Time	TP	Comm	Time	TP
Online	DN07⋆	5	10.85	60.58	63.39	25.82	66.33	57.89	59.87	72.08	53.27
		7	16.28	63.60	60.38	38.75	69.63	55.15	89.86	75.65	50.76
		9	21.71	63.62	60.37	51.67	69.66	55.09	119.81	75.67	50.72
	This (semi)	5	6.42	16.12 ±.01	236.21 ±.15	15.39	17.61 ±.02	217.93 ±0.2	35.87	19.34 ±.02	198.55 ±.35
		7	9.63			23.08			53.80		
		9	12.84			30.78			71.74		
	This (mal)	5	9.51	16.8 ±.1	228.71	22.79	18.3 ±.2	209.84	53.11	20.11 ±.06	190.95 ±.4
		7	14.14		228.44	33.88		209.49	78.96		
		9	18.40		226.82	44.06		207.23	102.68		
			Comm[c]	Time[d]	Cost[e]	Comm	Time	Cost	Comm	Time	Cost
End-to-End	DN07⋆	5	0.17	74.13	0.25	0.40	82.24	0.31	0.92	92.04	0.43
		7	0.25	77.76	0.37	0.60	86.99	0.47	1.39	98.90	0.64
		9	0.33	77.79	0.50	0.80	87.39	0.62	1.85	98.93	0.84
	This (semi)	5	0.17	19.08 ±.02	0.07	0.40	21.69 ±.02	0.11	0.92	25.97 ±.03	0.21
		7	0.25		0.10	0.60		0.16	1.39		0.31
		9	0.33		0.13	0.80		0.21	1.85		0.39
	This (mal)	5	0.18	31.21 ±.08	0.12	0.42	34.52 ±.2	0.17	0.99	41.83 ±.06	0.29
		7	0.27		0.16	0.64		0.25	1.48		0.42
		9	0.36		0.21	0.85		0.30	1.97		0.52

[a]communication in MB [b]TP denotes throughput [c]communication in GB [d]Time in seconds [e]monetary cost in USD

Table 7.9: Benchmarks for genome sequence matching.

For the monetary cost (Fig. 7.35), our semi-honest protocol has up to 66% saving over DN07⋆, and malicious variant has around 42%-54% overhead over the semi-honest counterpart.

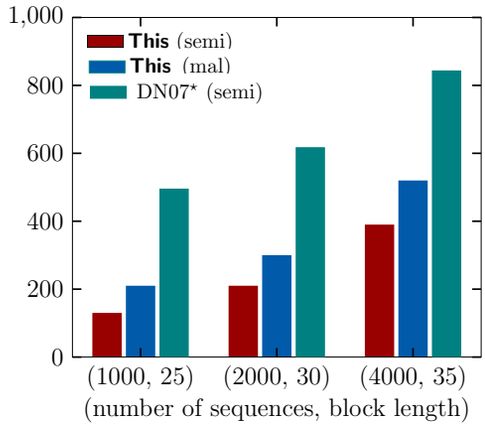


Figure 7.35: Monetary cost for SSQ evaluation for varying number of sequences and block lengths ((1000,25), (2000, 30), (4000,35)) for $n = 9$ parties. Costs for 1000 instances are reported in USD.

7.7.4 Biometric matching

We extend support for biometric matching, which finds application in many real-world tasks such as face recognition [77] and fingerprint matching [106]. Given a database of m biometric samples $(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m)$ each of size n, and a user holding its sample $\boldsymbol{u}$, the goal of biometric matching is to identify the sample from the database that is "closest" to $\boldsymbol{u}$. The notion of "closeness" can be formalized by various distance metrics, of which Euclidean Distance (**EuD**) is the most widely used. Following the general trend, we reduce our biometric matching problem to that of finding the sample from the database which has least **EuD** with user's sample $\boldsymbol{u}$. We follow [174, 194] where the **EuD** between two vectors $\boldsymbol{x}, \boldsymbol{y}$ each of length n is given as

$$\mathbf{EuD}_{\boldsymbol{xy}} = \sum_{i=1}^{i=\mathsf{n}} (\mathsf{x}_i - \mathsf{y}_i)^2 = \boldsymbol{z} \odot \boldsymbol{z} \tag{7.6}$$

where $\boldsymbol{z} = ((\mathsf{x}_1 - \mathsf{y}_1), \ldots, (\mathsf{x}_\mathsf{n} - \mathsf{y}_\mathsf{n}))$.

To achieve this goal of performing biometric matching securely, each $\boldsymbol{s}_i$, for all $i \in \{1, \ldots, m\}$ in the database is $[\![\cdot]\!]$-shared among the n parties participating in the computation. Specifically, each component $\boldsymbol{s}_{i_j}$, for all $j \in \{1, \ldots, \mathsf{n}\}$ is $[\![\cdot]\!]$-shared among all the parties. Similarly, the user also $[\![\cdot]\!]$-shares its sample $\boldsymbol{u}$. The parties compute a $[\![\cdot]\!]$-shared distance vector **DV** of size m, where the i^{th} component corresponds to the **EuD** between $\boldsymbol{u}$ and $\boldsymbol{s}_i$. For this, each party locally obtains $[\![\boldsymbol{z}_i]\!] = [\![\boldsymbol{s}_i]\!] - [\![\boldsymbol{u}]\!]$ and computes $[\![\mathbf{DV}_i]\!]$ according to Eq. 7.6 using the dot product operation. The final step is then to identify the minimum of these m components of **DV**, which can be performed using the protocol Π_{min} for minpool operation.

The benchmarks for biometric matching appear in Table 7.10 for a varying number of sequences. As is evident from Table 7.10, our semi-honest protocol witnesses a 4.6× improvement over DN07* in both online run-time and throughput. Further, in terms of monetary cost, we observe a saving of around 85%. With respect to our maliciously secure protocol, we incur a minimal overhead of around 9.5% in terms of total communication and around 4% in online throughput over our semi-honest variant. We note that our malicious variant outperforms semi-honest DN07* in both online run-time and throughput, thereby achieving our goal of a fast online phase.

7.8 Security proofs

Security proofs are given in the real-world/ideal-world simulation-based paradigm [155]. Let $\mathcal{A}^{\mathsf{sh}}, \mathcal{A}^{\mathsf{mal}}$ denote the real-world semi-honest, malicious adversary, respectively, corrupting at

	Ref.	n	#seq = 1024			#seq = 4096			#seq = 16384			#seq = 65536		
			Comm	Time	TP[a]	Comm	Time	TP	Comm	Time	TP	Comm	Time	TP
Online	DN07⋆	5	0.63	55.52	69.17	2.51	66.51	57.73	10.03	77.51	49.54	40.14	88.64	43.32
		7	0.94	58.27	65.90	3.76	69.81	55.00	15.06	81.35	47.20	60.23	93.04	41.27
		9	1.25	58.30	65.88	5.02	69.87	54.97	20.07	81.36	47.14	80.31	93.10	41.16
	This (semi)	5	0.09	12.61 ±.02	304.62 ±.03	0.35	15.07 ±.02	254.86 ±.03	1.41	17.53 ±.02	219.16 ±.04	5.62	19.99 ±.02	192.09 ±.04
		7	0.13			0.53			2.11			8.44		
		9	0.18			0.70			2.81			11.25		
	This (mal)	5	0.14	13.43 ±.02	285.93 ±.2	0.53	15.89 ±.02	241.66 ±.1	2.11	18.35 ±.02	209.24 ±.06	8.44	20.86 ±.02	183.88 ±.07
		7	0.21			0.80			3.17			12.67		
		9	0.28			1.07			4.23			16.89		
			Comm	Time	Cost[b]	Comm	Time	Cost	Comm	Time	Cost	Comm	Time	Cost
End-to-End	DN07⋆	5	6.92	66.55	0.20	27.70	79.85	0.24	110.83	93.47	0.29	443.34	108.53	0.40
		7	10.38	69.32	0.30	41.55	83.24	0.36	166.24	97.70	0.45	665.00	114.45	0.59
		9	13.84	69.35	0.40	55.40	83.30	0.48	221.66	97.71	0.59	886.67	114.55	0.79
	This (semi)	5	6.93	14.79 ±.02	0.03	27.74	17.35 ±.03	0.04	110.99	20.24 ±.05	0.06	443.99	24.62 ±.1	0.13
		7	10.40		0.05	41.61		0.06	166.49		0.09	665.99		0.18
		9	13.86		0.06	55.49		0.08	221.99		0.11	887.99		0.23
	This (mal)	5	7.61	26.67 ±.02	0.08	30.43	29.27 ±.02	0.09	121.71	32.30 ±.03	0.11	486.85	37.33 ±.05	0.18
		7	11.42		0.11	45.65		0.13	182.58		0.16	730.28		0.26
		9	15.22		0.14	60.81		0.16	243.19		0.20	972.72		0.33

Communication in MB and time in seconds.
[a]TP denotes throughput [b]monetary cost in USD

Table 7.10: Benchmarks for biometric matching.

most t parties in $\mathcal{P}$, denoted by $\mathcal{C}$. Let $\mathcal{S}^{\mathsf{sh}}, \mathcal{S}^{\mathsf{mal}}$ denote the corresponding ideal world semi-honest, malicious adversary, respectively. Security proofs are given in the $\{\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{TrGen}}\}$-hybrid (and $\{\mathcal{F}_{\mathsf{Broadcast}}, \mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}, \mathcal{F}_{\mathsf{MulPre}}, \mathcal{F}_{\mathsf{DotPPre}}\}$-hybrid for malicious setting) model. For modularity, we provide simulation steps for each protocol separately.

7.8.1 Semi-honest security

The following is the strategy for simulating the computation of function f (represented by a circuit ckt). The simulator $\mathcal{S}^{\mathsf{sh}}$ knows the input and output of the adversary $\mathcal{A}^{\mathsf{sh}}$, and sets the inputs of the honest parties to be 0. $\mathcal{S}^{\mathsf{sh}}$ emulates $\mathcal{F}_{\mathsf{Setup}}$ and gives the respective keys to the $\mathcal{A}^{\mathsf{sh}}$. Knowing all the inputs and randomness, $\mathcal{S}^{\mathsf{sh}}$ can compute all the intermediate values for each building block in the clear. Thus, $\mathcal{S}^{\mathsf{sh}}$ proceeds to simulate each building block in topological order using the aforementioned values (input and output of $\mathcal{A}^{\mathsf{sh}}$, randomness and intermediate values). We provide the simulation steps for each of the sub-protocols separately for modularity. When carried out in the respective order, these steps result in the simulation steps for the entire computation. To distinguish the simulators for various protocols, we use the corresponding protocol name as the subscript of $\mathcal{S}^{\mathsf{sh}}$.

Sharing and Reconstruction Simulation for input sharing (Fig. 7.9) and reconstruction appears in Fig. 7.36, Fig. 7.37, respectively.

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

Preprocessing:

- Emulate $\mathcal{F}_{\mathsf{Setup}}$ and give the respective shared keys to $\mathcal{A}^{\mathsf{sh}}$.
- Sample shares of α_{a} commonly held with $\mathcal{A}^{\mathsf{sh}}$ using the respective PRF keys while other values are sampled randomly.

Online:

- If $P_s \in \mathcal{C}$, receive β_{a} from $\mathcal{A}^{\mathsf{sh}}$ on behalf of honest parties in $\mathcal{E}$. Else, set $\mathsf{a} = 0$, $\beta_{\mathsf{a}} = \alpha_{\mathsf{a}}$ and sends β_{a} to $\mathcal{A}^{\mathsf{sh}}$ on behalf of P_s if there exists a corrupt party in $\mathcal{E}$.

Figure 7.36: Semi-honest: Simulation for the input sharing protocol Π_{Sh} by P_s.

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

- If $P_{\mathsf{king}} \in \mathcal{C}$, use the output a, and β_{a} and ${}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_j$ held by corrupt $P_j \in \mathcal{C} \cap \mathcal{E}$ to compute the shares ${}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_i$ of each honest $P_i \in \mathcal{E}$ such that $\beta_{\mathsf{a}} - \mathsf{a} = \sum_{P_i \in \mathcal{E} \setminus \mathcal{C}} {}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_i + \sum_{P_j \in \mathcal{C} \cap \mathcal{E}} {}^{\mathcal{E}}\langle\alpha_{\mathsf{a}}\rangle_j$. Send the shares of the honest parties in $\mathcal{E}$ to $\mathcal{A}^{\mathsf{sh}}$.
- If P_{king} is honest, send output a to $\mathcal{A}^{\mathsf{sh}}$ on behalf of P_{king}.

Figure 7.37: Semi-honest: Simulation for the reconstruction protocol towards all the parties.

Multiplication Simulation steps for multiplication (Fig. 7.11) are provided in Fig. 7.38.

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

Preprocessing:

- If $\mathsf{isTr} = 0$: Sample $[\cdot]$-shares of r commonly held with $\mathcal{A}^{\mathsf{sh}}$ using the respective shared keys while other values are sampled randomly.
- Else if $\mathsf{isTr} = 1$: Emulate $\mathcal{F}_{\mathsf{TrGen}}$ to generate $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$.
- On behalf of every honest $P_i \in \mathcal{D}$, send a random value for $\langle\Lambda_{\mathsf{ab}} - \mathsf{r}\rangle_i$ to $\mathcal{A}^{\mathsf{sh}}$ if $P_{\mathsf{king}} \in \mathcal{C}$.

Online:

- If $P_{\mathsf{king}} \in \mathcal{C}$, send random value for ${}^{\mathcal{E}}\langle\zeta\rangle_i$ to $\mathcal{A}^{\mathsf{sh}}$ on behalf of the honest $P_i \in \mathcal{E}$.
- If $P_{\mathsf{king}} \notin \mathcal{C}$, send a random $\mathsf{z} - \mathsf{r}$ to $\mathcal{A}^{\mathsf{sh}}$, if there exists a corrupt party in $\mathcal{E}$.

Figure 7.38: Semi-honest: Simulation for the multiplication protocol Π_{Mul}.

Observe that the adversary's view in the simulation is indistinguishable from its view in the real world since it only receives random value in each step of the protocol.

Other building blocks Simulation steps for the remaining building blocks can be obtained analogously by simulating the steps for the respective underlying protocols in their order of invocations.

Complete MPC protocol Simulation for the complete semi-honest MPC protocol $\Pi^{\mathsf{sh}}_{\mathsf{MPC}}$ (Fig. 7.12) appears in Fig. 7.39.

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

Preprocessing:

– Execute the preprocessing steps of simulators $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Sh}}$, $\mathcal{S}^{\mathsf{sh}}_{\mathsf{mult}}$ in the sequence of the gates in the circuit. For each addition gate with input wires a and b and output wire c, compute $[\alpha_{\mathsf{c}}]_{\mathcal{T}} = [\alpha_{\mathsf{a}}]_{\mathcal{T}} + [\alpha_{\mathsf{b}}]_{\mathcal{T}}$ for each $\mathcal{T}$ such that some honest $P_i \in \mathcal{T}$.

Online:

– Execute the online steps of simulators $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Sh}}$. For each input wire x_s for which $P_s \in \mathcal{C}$ provides the input, using the $[\alpha_{\mathsf{x}_s}]$ shares of honest parties chosen during preprocessing and β_{x_s} received from P_s in the online, compute the value x_s.

– Invoke $\mathcal{F}_f$ with $(\mathsf{Input}, \mathsf{x}_s)$ for each $P_s \in \mathcal{C}$. Receive from $\mathcal{F}_f$ the output of each $P_s \in \mathcal{C}$.

– Execute the online steps of $\mathcal{S}^{\mathsf{sh}}_{\mathsf{mult}}$ for all multiplication gates excluding those in the last layer.

– Execute the steps of $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Rec}}$ for each output wire in the circuit.

Figure 7.39: Semi-honest: Simulation for the complete MPC protocol $\Pi^{\mathsf{sh}}_{\mathsf{MPC}}$.

Theorem 7.1 *Protocol $\Pi^{\mathsf{sh}}_{\mathsf{MPC}}$ (Fig. 7.12) realises $\mathcal{F}_f$ (Fig. 7.8) with computational security against a semi-honest adversary $\mathcal{A}^{\mathsf{sh}}$ in the $\{\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{TrGen}}\}$-hybrid model.*

Proof: We prove that the adversary's view in the simulation is indistinguishable from its view in the real world via a sequence of hybrids.

Hybrid$_0$: Execution of protocol $\Pi^{\mathsf{sh}}_{\mathsf{MPC}}$ in the real world.

Hybrid$_1$: In this hybrid, the execution of Π_{Sh} is replaced by the simulation of $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Sh}}$. The two hybrids differ only in the case of inputs x_s of each honest P_s. Note that for the input of an honest P_s, the adversary's view consists of $(\beta_{\mathsf{x}_s}, [\alpha_{\mathsf{x}_s}]_i)$ for each $P_i \in \mathcal{C}$. Of these, $[\alpha_{\mathsf{x}_s}]_i$ consists of random values selected using the shared key setup among parties and hence is indistinguishable in both the hybrids. Moreover, β_{x_s} remains a random value from the adversary's view in both

the hybrids due to the share $[\alpha_{\mathsf{x_s}}]_{\mathcal{T}}$ where $\mathcal{T} \subseteq \mathcal{P} \setminus \mathcal{C}$ unknown to the adversary. Hence the distributions of $\mathbf{Hybrid_0}$ and $\mathbf{Hybrid_1}$ are indistinguishable.

$\mathbf{Hybrid_2}$**:** In this hybrid, the execution of Π_{Mul} is replaced with the simulation of $\mathcal{S}^{\mathsf{sh}}_{\mathsf{mult}}$ for all the multiplication gates. The adversary's view here may consist of the reconstructed value $\mathsf{z}-\mathsf{r}$ if some corrupt party belongs to $\mathcal{E}$. However, note that it remains a random value from the adversary's view in both the hybrids due to the randomly chosen r. Moreover, r is unknown to the adversary due to the common share held by $n-t$ honest parties. Hence, the distributions of $\mathbf{Hybrid_1}$ and $\mathbf{Hybrid_2}$ are indistinguishable.

$\mathbf{Hybrid_3}$**:** In this hybrid, the reconstruction protocol is replaced with the simulation of $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Rec}}$. Note that this is exactly the execution in the ideal world. The transcript of a corrupt party for an output wire a consists of shares of $[\alpha_{\mathsf{a}}]$ and a. As described in $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Rec}}$, the simulator obtains the output wire value a from the functionality and adjusts the shares of α_{a} held only by the honest parties to ensure a sharing that is consistent with the output a. Since α_{a} is random and unknown to the adversary, β_{a} is also random. Hence, the adversary's view is indistinguishable in both these executions.

Thus we conclude that the view of the adversary is indistinguishable in $\mathbf{Hybrid_0}$, which is the execution of the protocol in the real world and $\mathbf{Hybrid_3}$ corresponding to the execution in the ideal world. □

7.8.2 Malicious security

The following is the strategy for simulating the computation of function f (represented by a circuit ckt). The simulator emulates $\mathcal{F}_{\mathsf{Setup}}$ and gives the respective keys to the malicious adversary, $\mathcal{A}^{\mathsf{mal}}$. This is followed by the input sharing phase in which $\mathcal{S}^{\mathsf{mal}}$ extracts the input of $\mathcal{A}^{\mathsf{mal}}$, using the known keys, and sets the inputs of the honest parties to be 0. Knowing all the inputs, $\mathcal{S}^{\mathsf{mal}}$ can compute all the intermediate values for each building block in the clear. $\mathcal{S}^{\mathsf{mal}}$ proceeds to simulate each building block in topological order using the aforementioned values (inputs of $\mathcal{A}^{\mathsf{mal}}$, intermediate values). Finally, depending on whether $\mathcal{A}^{\mathsf{mal}}$ misbehaved, which $\mathcal{S}^{\mathsf{mal}}$ can detect using the aforementioned information, $\mathcal{S}^{\mathsf{mal}}$ invokes $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ to obtain the function output and forwards it to $\mathcal{A}^{\mathsf{mal}}$. As before, we provide the simulation steps for each of the sub-protocols separately for modularity. When carried out in the respective order, these steps result in the simulation steps for the entire computation. To distinguish the simulators for various protocols, the corresponding protocol name appears as the subscript of $\mathcal{S}^{\mathsf{mal}}$.

Sharing Simulation for input sharing appears in Fig. 7.40.

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

Preprocessing:

– Emulate $\mathcal{F}_{\mathsf{Setup}}$ and give the respective shared keys to $\mathcal{A}^{\mathsf{mal}}$.

– Sample shares of α_{a} commonly held with $\mathcal{A}^{\mathsf{mal}}$ using the respective PRF keys while other values are sampled randomly.

Online:

– For $P_s \in \mathcal{C}$, receive β_{a} from $\mathcal{A}^{\mathsf{mal}}$ on behalf of honest parties in $\mathcal{E}$, and obtain $\mathsf{a} = \beta_{\mathsf{a}} - \alpha_{\mathsf{a}}$ (since $\mathcal{S}^{\mathsf{mal}}$ knows all the PRF keys, it knows α_{a}). Invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Input}, \mathsf{a})$ on behalf of $\mathcal{A}^{\mathsf{mal}}$.

– On behalf of the honest parties, set its input $\mathsf{a} = 0$, $\beta_{\mathsf{a}} = \alpha_{\mathsf{a}}$ and send β_{a} to $\mathcal{A}^{\mathsf{mal}}$ if there exists a corrupt party in $\mathcal{E}$.

Verification: Send $\mathsf{H}(\beta_{\mathsf{a}})$ to $\mathcal{A}^{\mathsf{mal}}$ on behalf of the honest parties. If inconsistent β_{a}s were received with respect to a corrupt party, invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$.

Figure 7.40: Malicious: Simulation for the input sharing protocol $\Pi^{\mathsf{M}}_{\mathsf{Sh}}$ by P_s.

Reconstruction Simulation for reconstruction (with fairness) appears in Fig. 7.41.

Simulator $\mathcal{S}^{\mathsf{mal}}_{\mathsf{fairRec}}$

Preprocessing:

– $\mathcal{S}^{\mathsf{mal}}$ generates commitments on $[\alpha_{\mathsf{z}}]_{\mathcal{T}_j}$, for $j \in \{1, \ldots, \mathsf{q}\}$, and sends to $\mathcal{A}^{\mathsf{mal}}$ on behalf of honest parties.

– If $\mathcal{A}^{\mathsf{mal}}$ sends inconsistent commitment, send $(\mathsf{Signal}, \mathtt{abort})$ to $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$.

Online:

– $\mathcal{S}^{\mathsf{mal}}$ emulates $\mathcal{F}_{\mathsf{Broadcast}}$ as a sender to broadcast an alive bit on behalf of each honest party that has not aborted the computation so far. $\mathcal{S}^{\mathsf{mal}}$ emulates $\mathcal{F}_{\mathsf{Broadcast}}$ as a receiver to receive the alive bits broadcast by $\mathcal{A}^{\mathsf{mal}}$ on behalf of corrupt parties.

– If there exists some corrupt party for which $\mathcal{A}^{\mathsf{mal}}$ does not broadcast an alive bit, then $\mathcal{S}^{\mathsf{mal}}$ invokes $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$. Else, use output z obtained from $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$, to adjust the masked value β_{x} as $\beta_{\mathsf{x}} = \mathsf{z} + \alpha_{\mathsf{x}}$ with respect to the output multiplication gate[a]. Send opening of the commitments with respect to the shares of the output on behalf of the honest parties to $\mathcal{A}^{\mathsf{mal}}$.

[a]Without loss of generality, here the output gate is assumed to be multiplication gate. If not, the mask corresponding to the last multiplication gate, whose output goes as inputs to the output gate, should be adjusted to ensure that the final output matches z.

Figure 7.41: Malicious: Simulation for the fair reconstruction protocol $\Pi^{\mathsf{fair}}_{\mathsf{Rec}}$ towards all the parties.

Multiplication Simulation steps for multiplication (Fig. 7.23) are provided in Fig. 7.42.

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

Preprocessing:

- If $\mathsf{isTr} = 0$: Sample $[\cdot]$-shares of r commonly held with $\mathcal{A}^{\mathsf{mal}}$ using the respective shared keys while other values are sampled randomly.
- Else if $\mathsf{isTr} = 1$: Emulate $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ to generate $[\![\mathsf{r}]\!], [\![\mathsf{r}^d]\!]$.
- Emulate $\mathcal{F}_{\mathsf{MulPre}}$ to generate $[\cdot]$-shares of Λ_{ab}.

Online:

- If $P_{\mathsf{king}} \in \mathcal{C}$, send random value for ${}^{\mathcal{E}}\langle\zeta\rangle_i$ to $\mathcal{A}^{\mathsf{mal}}$ on behalf of the honest $P_i \in \mathcal{E}$.
- If $P_{\mathsf{king}} \notin \mathcal{C}$, send a random $\mathsf{z} - \mathsf{r}$ to $\mathcal{A}^{\mathsf{mal}}$.

Verification:

- Send $\mathsf{H}(\mathsf{z}_1 - \mathsf{r}_1 || \ldots || \mathsf{z}_m - \mathsf{r}_m)$ with respect to m multiplications, to $\mathcal{A}^{\mathsf{mal}}$ on behalf of the honest parties. If the hash values received from $\mathcal{A}^{\mathsf{mal}}$ are inconsistent, invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$.
- If $\mathcal{A}^{\mathsf{mal}}$ has sent incorrect $\mathsf{z} - \mathsf{r}$ for any multiplication ($\mathcal{S}^{\mathsf{mal}}$ can detect this since it knows all inputs and randomness that should be used by $\mathcal{A}^{\mathsf{mal}}$), generate random shares for Ω and simulate reconstruction steps of $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Rec}}$. Invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$.
- Else, if $\mathcal{A}^{\mathsf{mal}}$ has behaved honestly throughout, simulate reconstruction of $\Omega = 0$ using steps from $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Rec}}$. Invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$.

Figure 7.42: Malicious: Simulation for the multiplication protocol $\Pi^{\mathsf{M}}_{\mathsf{mult}}$.

Observe that since $\mathcal{A}^{\mathsf{mal}}$ sees random shares in both the real-world protocol and in the simulation, the indistinguishability of the simulation follows.

Other building blocks Simulations for the remaining building blocks can be obtained analogously using the steps for the underlying protocols.

Complete MPC protocol Simulation for the complete malicious MPC protocol $\Pi^{\mathsf{mal}}_{\mathsf{MPC}}$ (Fig. 7.24) appears in Fig. 7.43.

Simulator $\mathcal{S}^{\mathsf{mal}}_{\mathsf{MPC}}$

Preprocessing:

– Execute the preprocessing steps of simulators $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Sh}}$, $\mathcal{S}^{\mathsf{mal}}_{\mathsf{mult}}$ and $\mathcal{S}^{\mathsf{mal}}_{\mathsf{fairRec}}$ in the sequence of the gates in the circuit. For each addition gate with input wires a and b and output wire c, compute $[\alpha_{\mathsf{c}}]_{\mathcal{T}} = [\alpha_{\mathsf{a}}]_{\mathcal{T}} + [\alpha_{\mathsf{b}}]_{\mathcal{T}}$ for each $\mathcal{T}$ such that some honest $P_i \in \mathcal{T}$. If an inconsistency is identified in any of the steps, then invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$ and terminate.

Online:

– Execute the online steps of simulators $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Sh}}$. For each input wire x_s for which $P_s \in \mathcal{C}$ provides the input, using the $[\alpha_{\mathsf{x}_s}]$ shares of honest parties chosen during preprocessing and β_{x_s} received from P_s in the online, compute the value x_s.

– Execute the online and verification steps of $\mathcal{S}^{\mathsf{mal}}_{\mathsf{mult}}$ for all multiplication gates excluding those in the last layer.

– If an inconsistency is identified in any of the steps, then invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$ and terminate. Otherwise, invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ with $(\mathsf{Input}, \mathsf{x}_s)$ for each $P_s \in \mathcal{C}$. Receive from $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ the output of each $P_s \in \mathcal{C}$.

– Execute the steps of $\mathcal{S}^{\mathsf{mal}}_{\mathsf{fairRec}}$ for each output wire in the circuit.

Figure 7.43: Malicious: Simulation for the complete MPC protocol $\Pi^{\mathsf{mal}}_{\mathsf{MPC}}$.

Theorem 7.2 *Protocol* $\Pi^{\mathsf{mal}}_{\mathsf{MPC}}$ *(Fig. 7.24) realises* $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n\text{-}PC}}$ *(Fig. 7.18) with computational security against a malicious adversary* $\mathcal{A}^{\mathsf{mal}}$ *in the* $\{\mathcal{F}_{\mathsf{Broadcast}}, \mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{MulPre}}, \mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}\}$*-hybrid model.*

Proof: We prove that the adversary's view in the simulation is indistinguishable from its view in the real world via a sequence of hybrids.

Hybrid$_0$: Execution of protocol $\Pi^{\mathsf{mal}}_{\mathsf{MPC}}$ in the real world.

Hybrid$_1$: In this hybrid, the execution of $\Pi^{\mathsf{M}}_{\mathsf{Sh}}$ is replaced by the simulation of $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Sh}}$. Similar to the semi-honest protocol, the two hybrids differ only in the case of inputs x_s of each honest P_s. Note that for the input of an honest P_s, the adversary's view consists of $(\beta_{\mathsf{x}_s}, [\alpha_{\mathsf{x}_s}]_i)$ for each $P_i \in \mathcal{C}$. Of these, $[\alpha_{\mathsf{x}_s}]_i$ consists of random values selected using the shared key setup among parties and hence is indistinguishable in both the hybrids. Moreover, β_{x_s} remains a random value from the adversary's view in both the hybrids due to the share $[\alpha_{\mathsf{x}_s}]_{\mathcal{T}}$ where $\mathcal{T} \subseteq \mathcal{P} \setminus \mathcal{C}$ unknown to the adversary. Hence the distributions of **Hybrid$_0$** and **Hybrid$_1$** are indistinguishable.

Hybrid$_2$: In this hybrid, the execution of $\Pi^{\mathsf{M}}_{\mathsf{mult}}$ is replaced with the simulation of $\mathcal{S}^{\mathsf{mal}}_{\mathsf{mult}}$ for all the multiplication gates. The adversary's view here may consist of the reconstructed value $\mathsf{z}-\mathsf{r}$. For multiplication gates in the last layer of the circuit, the simulation differs from the gates in

the other layers only for the reconstructed value $\mathsf{z} - \mathsf{r}$. Specifically, the simulator receives the output wire value of the multiplication gate by functionality $\mathcal{F}_f$ and adjusts $\mathsf{z} - \mathsf{r}$ (by adjusting shares held by honest parties) accordingly. However, note that it remains a random value from the adversary's view in both the hybrids due to the randomly chosen r. Moreover, r is unknown to the adversary due to the common share held by $n - t$ honest parties in $\mathcal{T} \subseteq \mathcal{P} \setminus \mathcal{C}$. Hence, the distributions of $\mathbf{Hybrid_1}$ and $\mathbf{Hybrid_2}$ are indistinguishable.

$\mathbf{Hybrid_3}$: In this hybrid, the fair reconstruction protocol $\Pi^{\mathsf{fair}}_{\mathsf{Rec}}$ is replaced with the simulation of $\mathcal{S}^{\mathsf{mal}}_{\mathsf{fairRec}}$. Note that this is exactly the execution in the ideal world. The transcript of a corrupt party for an output wire a consists of shares of $[\alpha_{\mathsf{a}}]$ and a. Assume without loss of generality that the output wire is the output of a multiplication gate. As described earlier, the simulator obtains the output wire value a from the functionality and adjusts the shares of β_{a} held only by the honest parties to ensure a sharing that is consistent with the output a. Since α_{a} is random and unknown to the adversary, β_{a} is also random. Therefore, the adversary's view is indistinguishable in both these executions.

Thus we conclude that the view of the adversary is indistinguishable in $\mathbf{Hybrid_0}$, which is the execution of the protocol in the real world and $\mathbf{Hybrid_3}$ corresponding to the execution in the ideal world. □

Chapter 8

Conclusion and Open Problems

The thesis identifies various real-world applications where privacy is of utmost importance and designs privacy-preserving solutions for the same. Elaborately, the applications considered were in the realm of secure graph computation—secure local clustering and secure graph neural network—, secure computation for financial applications such as dark pools, and secure allegation escrow systems. To design secure and efficient solutions for these, the desirable MPC setting was identified with a focus on a small number of parties. In the process of designing secure solutions for graph-based computations, the secure frameworks of SWIFT [136] and Tetrad [138] were enhanced by incorporating missing primitives such as division, prefix OR, shuffle, etc., which makes these a more comprehensive framework. Next, to address the drawbacks present in traditional security definition, the work of [5] proposed the friends-and-foes (FaF) security model. The thesis identified the need to depart from this traditional security notion, which may not be desirable for certain applications that deal with highly sensitive data, and designed concretely efficient (1, 1)-FaF secure protocols in the 5-party setting. Specifically, the applications considered were dark pools and an allegation escrow system. Finally, given that different applications may demand operating with a varying number of parties, a generalized MPC protocol was provided that allows operating with an arbitrary (constant) number of parties (n) and can tolerate up to $t < n/2$ corruptions. A wide range of building blocks in this multiparty setting were also designed that facilitate the secure realization of various applications, including but not limited to genome sequence matching, biometric matching, and even deep neural networks. All the designed solutions were benchmarked using Google Cloud instances, and their performance was analyzed using various metrics to showcase their practicality.

It is also worthwhile to note that the considered model of small-party honest majority for the various applications is indeed relevant in practice. This is because the small-party honest majority setting is gaining traction lately due to the simplicity and efficiency of the resulting

protocols. Elaborately, honest majority protocols are more efficient than their dishonest majority counterparts since the former allows the use of efficient information-theoretic tools as opposed to more expensive cryptographic methods in the latter. In general, MPC protocols are known to have an overhead in comparison to the cleartext solutions. Hence, it is of utmost importance to design protocols that have good efficiency. Another reason why the small-party honest majority setting is practically relevant is because of the following. When working in the secure outsourced computation (SOC) setting, powerful servers are hired to enact the role of parties in the MPC protocol, and these servers belong to reputed organizations. Any cheating or malicious behaviour by these servers will put the reputation of these organizations at stake. Hence, these servers do not have an incentive to cheat or collude with others. Assuming an honest majority among a small set of parties (3 or 4) translates to the fact that these parties do not collude among themselves. This non-collusion assumption aligns seamlessly with the SOC scenario and, hence, is a reasonable assumption to make in practice.

For the aforementioned reasons, several real-world deployments of MPC have indeed considered the 3-party honest-majority setting, thereby substantiating the practical usability of the considered model. We briefly describe a few of these below.

1. The *Danish sugar beet auction* [28] was carried out in 3-party honest-majority setting between representatives from Danisco (sugar beet processor), Danish sugar beet grower's association and the research group responsible for implementing the auction computation.
2. *Prime Match* [196] is a privacy-preserving inventory matching solution that is in production at J.P Morgan, a large US bank, that includes a 3-party honest majority solution to facilitate trading of goods between two clients where the bank acts as third party.
3. Another example is that of *interoperable private attribution (IPA)* [42], which is a privacy-preserving framework that is proposed by Meta and Mozilla for attribution measurement in digital advertising and is being implemented with a 3-party honest majority setting.

In this way, for the applications considered in the thesis, such as secure local clustering and anonymous broadcast, to name a few, we believe that our choice of a 3-party honest majority setting will facilitate attaining efficient protocols in practice. Further, we would like to note that while 3-party solutions are being adopted in practice, the 4-party honest majority has also been shown to be more efficient than its 3-party counterpart. We believe that the adoption of 4-party solutions in practice is imminent. Hence, for compute-intensive applications such as GNNs, we believe the 4-party honest majority setting is justified. Even for applications such as dark pools, the Security Exchange Commission (SEC), which is responsible for ensuring the integrity of the dark pool, can be modelled as the semi-honest party, while the dark pool

operator can be modelled as the malicious one. Given this, the (1, 1)-FaF solution fits the bill, allowing us to achieve the desired security as well as efficiency. Finally, note that the choice of the number of parties is dependent on the application scenario. Hence, we also design in the thesis generic MPC solutions that cater to an arbitrary number of parties.

Open problems While improving the efficiency of the designed solutions is a direct follow-up question for further exploration, other questions that are left open are as follows.

1. *Applications.* The 3PC and 4PC frameworks of SWIFT [136] and Tetrad [138] were enhanced to include missing primitives, which makes it a more comprehensive framework. Although these frameworks were designed keeping PPML applications in mind, the addition of shuffle opens up new avenues for realizing various other applications. Extending these enhanced frameworks to realize other applications efficiently, such as secure auctions, private heavy hitters, and privacy-preserving contextual bandits, to name a few, is an interesting direction to pursue. This may require designing new primitives such as secure sort, compaction, etc. Moreover, the thesis provides efficient realizations for a secure shuffle protocol in the 3PC and 4PC settings. Given that shuffle is an important primitive that finds use in a plethora of applications, it will be interesting to design secure shuffle protocols in the FaF-model as well as the general multiparty settings considered in the thesis. This will open up a wide avenue of shuffle-based applications that can be realized securely.

2. *Security notion.* All the protocols designed in the thesis for the 3PC, 4PC, and 5PC settings allow attaining the strongest security of guaranteed output delivery in the presence of a malicious adversary. However, the generalized protocols for an arbitrary number of parties can tolerate up to $t < n/2$ corruptions in two different corruption models—semi-honest and malicious. In the malicious model, these protocols provide security with fairness. Uplifting this security to guaranteed output delivery without hampering the protocol efficiency is left as an interesting open question.

3. *Adversarial setting.* The protocols in the thesis were designed primarily in the honest-majority setting. It is an interesting question to explore the dishonest majority setting as well, which allows tolerating up to $n - 1$ corruptions in the n-party setting. Further, as opposed to the synchronous network model with static corruptions considered in the thesis, designing protocols in the asynchronous network model and against a stronger adaptive adversary is left as an open question.

Bibliography

[1] Me too movement. https://metoomvmt.org/. Accessed: 2021-02-01. 10

[2] 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 *TCC*, 2019. 107, 239

[3] Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings. In *ACNS*, 2019. 238, 257, 261, 262

[4] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013. 101

[5] Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. MPC with Friends and Foes. In *CRYPTO*, 2020. 7, 8, 14, 19, 146, 148, 203, 293

[6] Abdelrahaman Aly and Nigel P Smart. Benchmarking privacy preserving scientific operations. In *ACNS*, 2019. 42, 100, 101, 111

[7] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P Smart, and Tim Wood. Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE. In *ACM WAHC@CCS*, 2019. 238

[8] Reid Andersen and Yuval Peres. Finding sparse cuts locally using evolving sets. In *STOC*, 2009. 41

[9] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *FOCS*, 2006. 41

[10] Igor V Anikin and Rinat M Gazimov. Privacy preserving dbscan clustering algorithm for vertically partitioned data in distributed systems. In *SIBCON*, 2017. 41, 42

[11] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS*, 2016. 21, 109, 113, 151, 225, 238, 241, 275

[12] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *IEEE S&P*, 2017. 21, 113, 151, 225, 238

[13] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure Graph Analysis at Scale. In *ACM SIGSAC*, 2021. x, xv, xx, 5, 6, 13, 14, 16, 28, 29, 30, 31, 33, 34, 37, 39, 43, 44, 50, 52, 58, 60, 77, 78, 79, 80, 81, 98, 102, 121, 122

[14] Venkat Arun, Aniket Kate, Deepak Garg, Peter Druschel, and Bobby Bhattacharjee. Finding Safety in Numbers with Secure Allegation Escrows. In *NDSS*, 2020. xii, 11, 14, 205, 206, 208, 209, 210, 211, 214, 224, 226, 227

[15] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. Privacy-preserving search of similar patients in genomic data. *PETS*, 2018. 237, 273, 281, 282

[16] Gilad Asharov, Tucker Hybinette Balch, Antigoni Polychroniadou, and Manuela Veloso. Privacy-Preserving Dark Pools. In *AAMAS*, 2020. 9

[17] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In *ASIACRYPT*, 2020. 5

[18] Alessandro Baccarini, Marina Blanton, and Chen Yuan. Multi-Party Replicated Secret Sharing over a Ring with Applications to Privacy-Preserving Machine Learning. *PETS*, 2023. 238, 239

[19] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure MPC: Laziness Leads to GOD. In *ASIACRYPT*, 2020. 203

[20] Yikun Ban and Jingrui He. Local clustering in contextual multi-armed bandits. In *The Web Conference*, 2021. 4

[21] Carsten Baum, Ivan Damgård, Tomas Toft, and Rasmus Winther Zakarias. Better Preprocessing for Secure Multiparty Computation. In *ACNS*, 2016. 3

[22] Aner Ben-Efraim and Eran Omri. Concrete efficiency improvements for multiparty garbling with an honest majority. In *LATINCRYPT*, 2017. 238

[23] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *ACM CCS*, 2016. 238

[24] Azer Bestavros, Andrei Lapets, and Mayank Varia. User-centric distributed solutions for privacy-preserving analytics. *Communications of ACM*, 2017. 2

[25] Marina Blanton, Ahreum Kang, and Chen Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. In *ACNS*, 2020. 238, 239

[26] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*, pages 192–206, 2008. 238

[27] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the Estonian Tax and Customs Board Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation. In *FC*, 2015. 2

[28] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *FC*, 2009. 2, 294

[29] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *CRYPTO*, 2019. 15, 158, 162, 165, 225, 234, 238, 239, 242, 262, 276

[30] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *ACM CCS*, 2019. 7, 21, 22, 56, 107, 148, 165, 225, 262, 267

[31] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs. In *ASIACRYPT*, 2020. 7, 148, 158, 160, 161, 165, 171, 172, 174, 177, 179, 180, 181, 238, 239, 242, 257, 259, 267, 269

[32] Beyza Bozdemir, Sébastien Canard, Orhan Ermis, Helen Möllering, Melek Önen, and Thomas Schneider. Privacy-preserving density-based clustering. In *AsiaCCS*, 2021. 41, 42

[33] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:SGX cache attacks are practical. In *WOOT@USENIX*, 2017. 208

[34] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION–A Framework for Mixed-Protocol Multi-Party Computation. *ACM Transactions on Privacy and Security*, 2022. 238

[35] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014. 99

[36] Megha Byali, Arun Joseph, Arpita Patra, and Divya Ravi. Fast Secure Computation for Small Population over the Internet. In *ACM CCS*, 2018. 7, 22, 56, 225

[37] Megha Byali, Carmit Hazay, Arpita Patra, and Swati Singla. Fast actively secure five-party computation with security beyond abort. In *ACM CCS*, 2019. 148

[38] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. *PETS*, 2020. 2, 16, 21, 147, 148, 151, 179, 206, 225, 238

[39] Sergiu Carpov, Kevin Deforth, Nicolas Gama, Mariya Georgieva, Dimitar Jetchev, Jonathan Katz, Iraklis Leontiadis, M. Mohammadi, Abson Sae-Tang, and Marius Vuille. Manticore: Efficient Framework for Scalable Secure Multiparty Computation Protocols. *ePrint Archive*, 2021. https://eprint.iacr.org/2021/200. 238

[40] John Cartlidge, Nigel P Smart, and Younes Talibi Alaoui. MPC joins the dark side. In *AsiaCCS*, 2019. xvii, 9, 14, 148, 181, 182, 183, 184, 185, 186, 187

[41] John Cartlidge, Nigel P Smart, and Younes Talibi Alaoui. Multi-party computation mechanism for anonymous equity block trading: A secure implementation of turquoise plato uncross. *Intell. Syst. Account. Finance Manag.*, 2021. 9

[42] Benjamin Case, Richa Jain, Alex Koshelev, Andy Leiserson, Daniel Masny, Thurston Sandberg, Ben Savage, Erik Taubeneck, Martin Thomson, and Taiki Yamaguchi. Interoperable private attribution: A distributed attribution and aggregation protocol. *Cryptology ePrint Archive*, 2023. 294

[43] Octavian Catrina. Round-efficient protocols for secure multiparty fixed-point arithmetic. In *IEEE COMM*, 2018. 42, 45, 47, 48, 98, 100, 101, 113

[44] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN*, 2010. 42, 45, 48, 100, 250

[45] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *FC*, 2010. 42, 48, 100, 101, 250

[46] Pak K Chan, Martine DF Schlag, and Jason Y Zien. Spectral k-way ratio-cut partitioning and clustering. *IEEE TCAD*, 1994. 4

[47] T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *ASIACRYPT*, 2018. 5

[48] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *ASIACRYPT*, 2020. 43

[49] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*, 2019. 2, 16, 21, 100, 147, 206, 225, 238, 247, 275

[50] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*, 2020. 2, 3, 6, 16, 21, 42, 100, 101, 147, 151, 206, 225, 238, 271, 275

[51] David Chaum. The Spymasters Double-Agent Problem: Multiparty Computations Secure Unconditionally from Minorities and Cryptographically from Majorities. In *CRYPTO*, 1989. 203

[52] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO*, 2018. 238, 261

[53] Fan Chung. The heat kernel as the pagerank of a graph. *PNAS*, 2007. 41, 60

[54] Fan Chung and Olivia Simpson. Computing heat kernel pagerank and a local clustering algorithm. *European Journal of Combinatorics*, 2018. xx, 37, 38, 41, 44, 60, 66, 67, 74, 75, 76

[55] Richard Cleve. Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract). In *STOC*, 1986. 3, 225

[56] Daniele Cozzo, Nigel P Smart, and Younes Talibi Alaoui. Secure fast evaluation of iterative methods: with an application to secure pagerank. In *Cryptographers' Track at the RSA Conference*, 2021. 42

[57] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, 2005. 242

[58] 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 *CRYPTO*, 2018. 3

[59] Cryptography and Privacy Engineering Group at TU Darmstadt. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils. 178, 274

[60] Mariana Botelho da Gama, John Cartlidge, Antigoni Polychroniadou, Nigel P Smart, and Younes Talibi Alaoui. Kicking-the-Bucket: Fast Privacy-Preserving Trading Using Buckets. In *FC*, 2022. 9, 181

[61] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. In *USENIX Security*, 2021. 2, 16, 22, 58, 100, 148, 238, 274

[62] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007. 15, 174, 234, 236, 238, 239, 267, 268

[63] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, 2006. 101, 111

[64] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*, 2012. 3

[65] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS*, 2013. 3

[66] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. In *CRYPTO*, 2018. 3, 238

[67] 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. *IEEE S&P*, 2019. 238, 251, 252, 266

[68] Priyanka Das and Asit Kumar Das. Behavioural analysis of crime against women using a graph based clustering approach. In *ICCCI*, 2017. 4

[69] Ipsa De and Animesh Tripathy. A secure two party hierarchical clustering approach for vertically partitioned data set with accuracy measure. In *ISI*, 2014. 42

[70] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NeurIPS*, 2016. 99, 100

[71] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*, 2015. 238

[72] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society: series B (methodological)*, 1977. 42

[73] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *PKC*. Springer, 2005. 213, 214

[74] Danny Dolev and H. Raymond Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.*, 1983. 240

[75] Danny Dolev, Cynthia Dwork, Orli Waarts, and Moti Yung. Perfectly Secure Message Transmission. *J. ACM*, 1993. 203

[76] Jennifer A Dunne, Richard J Williams, and Neo D Martinez. Food-web structure and network theory: the role of connectance and size. *PNAS*, 2002. 4

[77] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-Preserving Face Recognition. *PETS*, 2009. 284

[78] Daniel Escudero and Anders Dalskov. Honest Majority MPC with Abort with Minimal Online Communication. In *LATINCRYPT*, 2021. 234, 236, 238, 239, 249, 257, 264

[79] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In *CRYPTO*, 2020. 111

[80] Saba Eskandarian and Dan Boneh. Clarion: Anonymous Communication from Multiparty Shuffling Protocols. In *NDSS*, 2022. 5, 6, 13, 39, 40, 43, 50, 52, 58, 60, 77, 78, 81, 82, 83, 84, 98

[81] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996. 42

[82] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading Correctness for Privacy in Unconditional Multi-Party Computation (Extended Abstract). In *CRYPTO*, 1998. 203

[83] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 2007. 41

[84] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT*, 2017. 109, 225, 238

[85] Daniel Genkin, Yuval Ishai, Manoj M Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *STOC*, 2014. 236, 238

[86] Shayan Oveis Gharan and Luca Trevisan. Approximating the Expansion Profile and Almost Optimal Local Graph Clustering. In *FOCS*, 2012. 3, 4

[87] Hossein Ghodosi and Josef Pieprzyk. Multi-Party Computation with Omnipresent Adversary. In *PKC*, 2009. 203

[88] Oded Goldreich. *Foundations of cryptography: volume 1, basic tools*. Cambridge university press, 2007. 17, 18

[89] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009. 2

[90] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP*, 2011. 42, 43

[91] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure Computation with Low Communication from Cross-Checking. In *ASIACRYPT*, 2018. 148

[92] S Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The More The Merrier: Reducing the Cost of Large Scale MPC. In *EUROCRYPT*, 2021. 238

[93] Vipul Goyal and Yifan Song. Malicious Security Comes Free in Honest-Majority MPC. In *CRYPTO*, 2020. 238, 239, 267

[94] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In *CRYPTO*, 2019. 236, 239, 257, 263, 264

[95] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and Scalable MPC in the Honest Majority Setting. In *CRYPTO*, 2021. 236, 237, 238, 255, 256

[96] The Guardian. Cambridge analytica scandal: what you need to know, 2018. URL https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election. 1

[97] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, LANL, 2008. 72

[98] Thomas Haines, Rajeev Goré, and Bhavesh Sharma. Did you mix me? formally verifying verifiable mix nets in electronic voting. In *IEEE SP*, 2021. 5

[99] Mona Hamidi, Mina Sheikhalishahi, and Fabio Martinelli. Privacy preserving expectation maximization (em) clustering construction. In *Distributed Computing and Artificial Intelligence, 15th International Conference 15*, 2019. 41, 42

[100] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *NeurIPS*, 2017. 99

[101] Danny Harnik, Paula Ta-Shma, and Eliad Tsfadia. It takes two to# metoo-using enclaves to build autonomous trusted systems. *arXiv preprint arXiv:1808.02708*, 2018. 208

[102] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 2020. 72

[103] Aditya Hegde, Helen Möllering, Thomas Schneider, and Hossein Yalame. Sok: Efficient privacy-preserving clustering. *PETS*, 2021. 42

[104] Aditya Hegde, Nishat Koti, Varsha Bhat Kukkala, Shravani Patil, Arpita Patra, and Protik Paul. Attaining GOD Beyond Honest Majority with Friends and Foes. In *ASIACRYPT*, 2022. viii

[105] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015. 99

[106] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *AsiaCCS*, 2013. 284

[107] Martin Hirt and Marta Mularczyk. Efficient MPC with a mixed adversary. In *ITC*, 2020. 203

[108] Martin Hirt, Ueli M. Maurer, and Vassilis Zikas. MPC vs. SFE : Unconditional and Computational Security. In *ASIACRYPT*, 2008. 203

[109] T Ryan Hoens, Marina Blanton, and Nitesh V Chawla. A private and reliable recommendation system for social networks. In *IEEE*. IEEE, 2010. 2

[110] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure {two-party} deep neural network inference. In *USENIX Security*, 2022. 100

[111] Ali Inan, Selim V Kaya, Yücel Saygın, Erkay Savaş, Ayça A Hintoğlu, and Albert Levi. Privacy preserving clustering on horizontally partitioned data. *Data & Knowledge Engineering*, 2007. 41, 42

[112] Alphabet Inc. Yahoo announces data breach of approximately 3 billion accounts, 2017. URL `https://www.alphabet.com/news/yahoo-announces-data-breach-approximately-3-billion-accounts`. 1

[113] Alicia Iriberri, Gondy Leroy, and Nathan Garrett. Reporting on-campus crime online: User intention to use. In *HICSS*, 2006. 10

[114] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 1989. 17, 239

[115] Geetha Jagannathan, Krishnan Pillaipakkamnatt, Rebecca N Wright, and Daryl Umano. Communication-efficient privacy-preserving clustering. *Trans. Data Priv.*, 2010. 41, 42

[116] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 1999. 42

[117] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. *IEEE Transactions on Dependable and Secure Computing*, 2023. https://eprint.iacr.org/2022/1561. viii

[118] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 2015. 101

[119] Hiroki Kanezashi, Toyotaro Suzumura, Xin Liu, and Takahiro Hirofuchi. Ethereum Fraud Detection with Heterogeneous Graph Neural Networks. *arXiv preprint arXiv:2203.12363*, 2022. 6

[120] Ravi Kannan, Santosh Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *JACM*, 2004. 4

[121] Banashri Karmakar, Nishat Koti, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. Asterisk: Super-fast mpc with a friend. *Cryptology ePrint Archive*, 2023. viii

[122] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 1953. 39

[123] Mahimna Kelkar, Phi Hung Le, Mariana Raykova, and Karn Seth. Secure poisson regression. In *USENIX Security*, 2022. 101

[124] Hannah Keller, Helen Möllering, Thomas Schneider, and Hossein Yalame. Balancing quality and efficiency in private clustering with affinity propagation. In *SECRYPT*, 2021. 41

[125] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. In *ASIACRYPT*, 2014. 42, 43

[126] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *ICML*, 2022. 42, 98, 100, 101, 111, 112, 113, 131

[127] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *ACM CCS*, 2013. 3

[128] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *ACM CCS*, 2016. 3

[129] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ Great Again. In *EUROCRYPT*, 2018. 3

[130] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing mpc for robust and scalable integer and floating-point arithmetic. In *FC*, 2016. 101

[131] Pan-Jun Kim, Dong-Yup Lee, and Hawoong Jeong. Centralized modularity of N-linked glycosylation pathways in mammalian cells. *PloS one*, 2009. 4

[132] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR (poster)*, 2015. 42, 100, 101, 109

[133] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017. 3, 97, 99, 100, 102, 103, 104, 105, 109, 131, 132

[134] Kyle Kloster and David F Gleich. Heat kernel based community detection. In *ACM SIGKDD*, 2014. 4, 39, 41, 44, 74

[135] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *NeurIPS*, 2021. 98

[136] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*, 2021. viii, xv, xx, 2, 3, 5, 6, 7, 13, 16, 20, 21, 22, 24, 25, 26, 37, 38, 39, 40, 42, 44, 45, 46, 51, 56, 58, 84, 85, 100, 101, 106, 135, 147, 148, 149, 151, 173, 174, 179, 206, 237, 238, 241, 250, 257, 266, 267, 271, 273, 274, 275, 278, 293, 295

[137] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. PentaGOD: Stepping beyond Traditional GOD with Five Parties. In *ACM CCS*, 2022. vii, 146

[138] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *NDSS*, 2022. viii, xv, xx, xxi, 2, 3, 6, 14, 16, 22, 25, 26, 27, 42, 56, 97, 98, 99, 100, 101, 102, 109, 115, 121, 131, 133, 137, 138, 148, 173, 174, 176, 178, 236, 238, 247, 250, 253, 269, 270, 271, 272, 275, 293, 295

[139] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Shield: Secure Allegation Escrow System with Stronger Guarantees. In *The Web Conference*, 2023. vii, 205

[140] Nishat Koti, Shravani Patil, Arpita Patra, and Ajith Suresh. MPClan: Protocol Suite for Privacy-Conscious Computations. *Journal of Cryptology*, 2023. https://eprint.iacr.org/2022/675. vii, 234

[141] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. 2014. https://www.cs.toronto.edu/~kriz/cifar.html. 188, 279

[142] Varsha Bhat Kukkala, Jaspal Singh Saini, and SRS Iyengar. Privacy preserving network analysis of distributed social networks. In *ICISS 2016*, 2016. 42

[143] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *IEEE S&P*, 2020. 100

[144] Benjamin Kuykendall, Hugo Krawczyk, and Tal Rabin. Cryptography for# metoo. *PETS*, 2019. 11, 206, 208

[145] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In *ISC*, 2011. xv, 28, 29, 42, 43, 52, 118

[146] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. http://yann.lecun.com/exdb/mnist/. 188, 279

[147] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998. 148, 188, 237, 278

[148] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017. 208

[149] Yanhua Li, Zhi-Li Zhang, and Jie Bao. Mutual or unrequited love: Identifying stable clusters in social networks with uni-and bi-directional links. In *WAW*, 2012. 4

[150] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *ICLR*, 2016. 99

[151] Chung-Shou Liao, Kanghao Lu, Michael Baym, Rohit Singh, and Bonnie Berger. IsoRankN: spectral methods for global alignment of multiple protein networks. *BMC Bioinformatics*, 2009. 4

[152] Manuel Liedel. Secure distributed computation of the square root and applications. In *ISPEC*, 2012. 101

[153] Frank Lin and William W Cohen. Power iteration clustering. In *ICML*, 2010. 4

[154] Xiaodong Lin, Chris Clifton, and Michael Zhu. Privacy-preserving clustering with distributed em mixture modeling. *Knowledge and information systems*, 2005. 41, 42

[155] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*. 2017. 17, 18, 284

[156] Yujia Liu, Kang Zeng, Haiyang Wang, Xin Song, and Bin Zhou. Content matters: a GNN-based model combined with text semantics for social network cascade prediction. In *PAKDD*, 2021. 6

[157] Zheng Liu, Xiaohan Li, Hao Peng, Lifang He, and S Yu Philip. Heterogeneous similarity graph neural network on electronic health records. In *Big Data*, 2020. 6

[158] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. Heterogeneous graph neural networks for malicious account detection. In *CIKM*, 2018. 97, 104, 135, 136, 137

[159] Peter Lofgren and Ashish Goel. Personalized pagerank to a target node. *preprint arXiv:1304.4658*, 2013. 39

[160] László Lovász and Miklós Simonovits. The mixing rate of Markov chains, an isoperimetric inequality, and computing the volume. In *FOCS*, 1990. 3, 4

[161] Donghang Lu and Aniket Kate. RPM: Robust Anonymity at Scale. *PETS*, 2022. 107

[162] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *ACM CCS*, 2019. 43

[163] Wen-jie Lu, Yixuan Fang, Zhicong Huang, Cheng Hong, Chaochao Chen, Hunter Qu, Yajin Zhou, and Kui Ren. Faster secure multiparty computation of adaptive gradient

descent. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, 2020. 101, 113

[164] Zhenqi Lu, Johan Wahlström, and Arye Nehorai. Local clustering via approximate heat kernel pagerank with subgraph sampling. *Scientific Reports*, 2021. 41

[165] David Lusseau, Karsten Schneider, Oliver J Boisseau, Patti Haase, Elisabeth Slooten, and Steve M Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 2003. 4

[166] Jun Ma, Danqing Zhang, Yun Wang, Yan Zhang, and Alexey Pozdnoukhov. GraphRAD: a graph-based risky account detection system. In *ACM SIGKDD*, 2018. 4

[167] Dhaneshwar Mardi, Surbhi Tanwar, and Jaydeep Howlader. Multiparty protocol that usually shuffles. *Security and Privacy*, 2021. 42, 43

[168] Peter Markstein. Software division and square root using Goldschmidt's algorithms. In *RNC*, 2004. 42, 47, 48

[169] Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S Dov Gordon. Secure parallel computation on national scale volumes of data. In *USENIX Security*, 2020. 3, 250

[170] Ekaterina Merkurjev, Andrea L Bertozzi, and Fan Chung. A semi-supervised heat kernel pagerank mbo algorithm for data classification. *Communications in Mathematical Sciences*, 2018. 39

[171] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-Sided Malicious Security for Private Intersection-Sum with Cardinality. In *CRYPTO*, 2020. 275

[172] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC*, 2007. 71, 72

[173] Payman Mohassel and Peter Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*, 2018. 2, 3, 6, 16, 21, 100, 101, 147, 151, 173, 175, 179, 188, 206, 237, 241, 250, 271, 273, 274, 275, 278

[174] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017. 2, 42, 100, 147, 175, 188, 206, 273, 278, 284

[175] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *ACM CCS*, 2015. 225

[176] Payman Mohassel, Mike Rosulek, and Ni Trieu. Practical privacy-preserving k-means clustering. *PETS*, 2020. 41

[177] Ben Morris. The mixing time of the thorp shuffle. *SIAM Journal on Computing*, 2008. 42, 43

[178] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Secure multi-party shuffling. In *SIROCCO*, 2015. 42

[179] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *IEEE S&P*, 2015. x, 5, 6, 16, 31, 32, 33, 34, 37, 38, 39, 40, 42, 44, 61, 68, 98, 102, 127

[180] C Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *ACM CCS*, 2001. 5

[181] Mark EJ Newman and Elizabeth A Leicht. Mixture models and exploratory analysis in networks. *PNAS*, 2007. 4

[182] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *NeurIPS*, 2001. 4

[183] Peter Sebastian Nordholt and Meilof Veeningen. Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification. In *ACNS*, 2018. 225

[184] United States of America before the Securities and Exchange Commission. SEC institutes enforcement action against 20 former New York Stock Exchange specialists alleging pervasive course of fraudulent trading. Press Release. 2005. https://www.sec.gov/news/press/2005-54.htm. 9

[185] United States of America before the Securities and Exchange Commission. In the Matter of Pipeline Trading Systems LLC, et al., Securities Exchange Act of 1934 Release No. 65609. 2011. https://www.sec.gov/litigation/admin/2011/33-9271.pdf. 9

[186] United States of America before the Securities and Exchange Commission. In the Matter of eBX, LLC Securities Exchange Act of 1934 Release No. 67979. 2012. https://www.sec.gov/litigation/admin/2012/34-67969.pdf. 9

[187] United States of America before the Securities and Exchange Commission. In the Matter of LavaFlow, Inc. Securities Exchange Act of 1934 Release No. 72673. 2014. https://www.sec.gov/litigation/admin/2014/34-72673.pdf. 9

[188] United States of America before the Securities and Exchange Commission. In the Matter of Liquidnet, Inc., Securities Exchange Act of 1934 Release No. 72339. 2014. https://www.sec.gov/litigation/admin/2014/33-9596.pdf. 9

[189] United States of America before the Securities and Exchange Commission. In the Matter of Credit Suisse Securities (USA) LLC, Securities Exchange Act of 1934 Release No. 77002. 2016. https://www.sec.gov/litigation/admin/2016/33-10013.pdf. 9

[190] United States of America before the Securities and Exchange Commission. In the Matter of ITG Inc. and Alternet Securities, Inc., Securities Exchange Act of 1934 Release No. 84548. 2018. https://www.sec.gov/litigation/admin/2018/33-10572.pdf. 9

[191] Satsuya Ohata and Koji Nuida. Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application, 2020. 174, 236, 253

[192] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999. 39

[193] Arpita Patra and Ajith Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*, 2020. 2, 3, 16, 21, 100, 101, 147, 151, 174, 179, 206, 225, 237, 238, 241, 242, 257, 258, 271, 278

[194] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*, 2021. 100, 101, 115, 116, 117, 173, 174, 177, 178, 179, 235, 236, 237, 253, 269, 271, 272, 273, 274, 284

[195] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place, comparison-based sorting with CUDA: A study with bitonic sort. *Concurrency and Computation: Practice and Experience*, 2011. 32, 75

[196] Antigoni Polychroniadou, Gilad Asharov, Benjamin E. Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime match: A privacy-preserving inventory matching system. In *USENIX Security*, 2023. 294

[197] Mohammad Shahriar Rahman, Anirban Basu, and Shinsaku Kiyomoto. Towards outsourced privacy-preserving multiparty dbscan. In *PRDC*, 2017. 41, 42

[198] Anjana Rajan, Lucy Qin, David W Archer, Dan Boneh, Tancrede Lepoint, and Mayank Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *ACM SIGCAS*, 2018. 10, 11, 206, 224

[199] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow2: Practical 2-party secure inference. In *ACM CCS*, 2020. 100

[200] 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 *AsiaCCS*, pages 707–721, 2018. 3

[201] Corban G Rivera, Rachit Vakil, and Joel S Bader. Nemo: network module identification in cytoscape. *BMC bioinformatics*, 2010. 4

[202] Dragos Rotaru and Tim Wood. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *INDOCRYPT*, 2019. 238

[203] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. Secure multiparty pagerank algorithm for collaborative fraud detection. In *FC*, 2019. 42

[204] Thomas Schneider and Oleksandr Tkachenko. EPISODE: efficient privacy-preserving similar sequence queries on outsourced genomic databases. In *AsiaCCS*, 2019. 237, 273, 281, 282

[205] Google Cloud Computing Services. Google Cloud Platform, 2008. Network costs - `https://cloud.google.com/vpc/network-pricing`, Computation costs - `https://cloud.google.com/compute/vm-instance-pricing`. 80, 180, 275

[206] Adi Shamir. How to Share a Secret. *Commun. ACM*, pages 612–613, 1979. 238

[207] Shreya Sharma, Chaoping Xing, and Yang Liu. Privacy-Preserving Deep Learning with SPDZ. 2019. 238

[208] Liyan Shen, Xiaojun Chen, Jinqiao Shi, Ye Dong, and Binxing Fang. An efficient 3-party framework for privacy-preserving neural network inference. In *ESORICS*, 2020. 6, 100, 131, 273

[209] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE PAMI*, 2000. 4

[210] Erez Shmueli and Tamir Tassa. Secure multi-party protocols for item-based collaborative filtering. In *ACM RecSys*, 2017. 2

[211] Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Find Thy Neighbourhood: Privacy-Preserving Local Clustering. *PETS*, 2023. vii, 37

[212] Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-Party Shuffle Protocols. *PETS*, 2023. vii, 37

[213] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015. 148, 188, 237, 278

[214] Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *STOC*, 2004. 3, 4, 41

[215] Daniel A Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM Journal on computing*, 2013. 3, 4, 41

[216] Stanford. CS231n: Convolutional Neural Networks for Visual Recognition. URL `https://cs231n.github.io/convolutional-networks/`. 174

[217] Hugo Steinhaus et al. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci*, 1956. 41

[218] Steven H Strogatz. Exploring complex networks. *Nature*, 2001. 3

[219] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *ICLR*, 2018. 99

[220] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PETS*, 2021. 2, 3, 16, 42, 45, 100, 188, 238, 273, 278, 279

[221] Robert A Wagner and Michael J Fischer. The String-to-String Correction Problem. *J. ACM*, 1974. 281

[222] Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. Approximate Graph Propagation. In *ACM SIGKDD*, 2021. 37, 39, 41, 44, 61, 62, 63, 65

[223] Jianyu Wang, Rui Wen, Chunming Wu, Yu Huang, and Jian Xion. Fdgars: Fraudster detection via graph convolutional networks in online app review system. In *Companion@WWW*, 2019. 97, 135, 136, 137

[224] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *ACM SIGSAC*, 2017. 208

[225] Yuchung J Wang and George Y Wong. Stochastic blockmodels for directed graphs. *Journal of the American Statistical Association*, 1987. 4

[226] Wei Wu, Jian Liu, Huimei Wang, Jialu Hao, and Ming Xian. Secure and efficient outsourced k-means clustering using fully homomorphic encryption with ciphertext packing technique. *IEEE Transactions on Knowledge and Data Engineering*, 2020. 41

[227] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020. 99

[228] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2015. 41, 42

[229] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2018. 99

[230] Renchi Yang, Xiaokui Xiao, Zhewei Wei, Sourav S Bhowmick, Jun Zhao, and Rong-Hua Li. Efficient estimation of heat kernel pagerank for local clustering. In *ACM SIGMOD*, 2019. 41, 44, 60, 74

[231] Andrew Chi-Chih Yao. Protocols for Secure Computations (Extended Abstract). In *FOCS*, 1982. 32

[232] Guangming Zhu, Lu Yang, Liang Zhang, Peiyi Shen, and Juan Song. Recurrent graph convolutional networks for skeleton-based action recognition. In *ICPR*, 2021. 99