LEARNING ON LARGE-SCALE DATA
WITH SECURITY AND PRIVACY

by

Sahar Sadat Seyed Mazloom
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

Dr. S. Dov Gordon, Dissertation Director

Dr. Foteini Baldimtsi, Committee Member

Dr. Zoran Duric, Committee Member

Dr. Kris Gaj, Committee Member

Dr. David Rosenblum, Department Chair

Dr. Kenneth S. Ball, Dean, The Volgenau School of Engineering

Date: November 30th, 2020

Fall Semester 2020
George Mason University
Fairfax, VA

Learning on Large-Scale Data with Security and Privacy

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Sahar Sadat Seyed Mazloom
Master of Science
Azad University Qazvin Branch, 2009
Bachelor of Science
Azad University Tehran-North Branch, 2006

Director: Dr. S. Dov Gordon, Professor
Department of Computer Science

Fall Semester 2020
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my beloved parents, Shahrzad and Hossein, for their endless support, constant encouragement, and unconditional love.

Acknowledgments

I would like to thank my PhD advisor, Dr. Dov Gordon for his continuous support and inspirational guidance during my studies. He created a fine balance between supervising research and providing enough personal freedom to develop my own ideas. I enjoyed and learned a lot from him during our whiteboard research discussions, and always appreciate his admirable attitude for treating his students as his peers. It was a great honor to work with him and to be his first doctoral student.

I am grateful to all of my doctoral committee members, Dr. Foteini Baldimtsi, Dr. Zoran Duric and Dr. Kris Gaj, for taking the time to be part of my committee, whom their valuable feedback had a great impact on the development of this dissertation. I also would like to extend my sincere gratitude to Prof. Hakan Aydin, the Director of the Computer Science PhD program at GMU, for his great comments and suggestions on my dissertation. Also, I want to thank Ms. Michele Pieper and Mr. Ryan Lucas in CS admin office, for the kind and constant support they provide for the students.

My experience of Ph.D. was made a lot more delightful because of the brilliant people I was lucky enough to get to know and/or collaborate with. In particular, I would like to thank my great friend and bright collaborator, Phi Hung Le, as well as Dr. Feng-Hao Liu and Dr. Samuel Ranellucci, for all the long hours of interesting discussions to solve research problems. I also would like to thank my kind and supportive lab-mates, Panagiotis (Panos) Chatzigiannis, Ioanna Karantaidou and Mingyu Liang, for the great time we shared at Security Lab.

Finally, I would like to express my profound appreciation and gratitude to my family, especially my dearest parents, Shahrzad and Hossein, for believing in me and giving me the strength and freedom to reach for the stars and chase my dreams. Words can't express my admiration to my brother, soon to be Dr. Ali, who supported me in every step of the way during my graduate studies, and also my sister and my best friend, Shadi, whose encouragements carried me through the hard days of PhD life, spent far away from the family.

I would like to leave this note here for my future-self, to remind her of how she tried her best to overcome the obstacles on her way, to not let the discouragements affect her determination, and remember to always try to go that extra mile to pursue her dreams. Trying to graduate from PhD and landing a job during a global pandemic didn't make it easier, although you did it; and know that this is just the beginning!

Table of Contents

List of Tables

List of Figures

Abstract

LEARNING ON LARGE-SCALE DATA WITH SECURITY AND PRIVACY

Sahar Sadat Seyed Mazloom, PhD

George Mason University, 2020

Dissertation Director: Dr. S. Dov Gordon

Recent advancements in machine learning domain have been enabled by the ability to analyze massive volumes of data, and to extract and learn patterns within that data. However, large-scale data collection raises privacy concerns, as it can expose individual's sensitive data to actors with malicious intent. This lack of privacy can lead to potential data breaches, and consequently, can compromise the successful development of machine learning techniques. Secure Computation is a branch of modern cryptography that introduces promising solutions for processing data in a privacy-preserving manner. It enables computing any functionality on data while the data is "encrypted". This field has been the topic of extensive research in recent years and made remarkable progress. However, most results remained impractical for real applications and its deployment remained limited due to efficiency and scalability constraints.

The goal of this dissertation is to present novel protocol designs and development techniques to overcome these efficiency and scalability limitations. We demonstrate how to construct secure and privacy-preserving machine learning schemes that are practical for real-world applications, while dealing with large-scale data, and guaranteeing security against different types of adversaries. In the first part of this dissertation, we design and

develop privacy-preserving machine learning frameworks using secure computation techniques and explore the trade-off between security and efficiency on these frameworks. In order to improve the efficiency, we relax the security notion by allowing the adversary to learn some small information during the computation. Then, we use Differential Privacy mechanisms to provide a formal bound on the amount of leakage, and prove that what is learned by the adversary is deferentially private. We also leverage Parallel Computation techniques to improve the performance and running time of these novel algorithms. These frameworks follow a centralized computation architecture in which users send their private data to untrusted computation servers in order to perform some computations on them. In the second part, we design and develop secure and privacy-preserving machine learning algorithms in the distributed setting known as Federated Learning. In federated learning, users do not share their sensitive data with the computation severs, but instead they train a local model on their private data and only send their model parameters to the computation servers, which then aggregate those local parameters and construct a global model on all participants' data. Our secure and privacy-preserving federated learning protocols are designed to have low communication cost, as well as being robust to the users dropping out of the protocol at any point. We leverage secure computation and differential privacy techniques to preserve the privacy of user's data, as well as the trained model's parameters. All of our secure and privacy-preserving frameworks presented in this dissertation are designed to support two adversarial models, passive and active adversaries.

Chapter 1: Introduction

Everyday, users generate huge amount of data, which are mostly sensitive, such as medical or financial records. Large organizations and enterprises such as Apple and Google, look at this data in the clear to run some machine learning algorithms, in order to improve their user experience and provide some services for users, such as movie recommendations implemented by Netflix or targeted advertisements deployed by Amazon and Facebook. However, these services come at the cost of user privacy, and exposing their sensitive data to the risk of data breach. One of the promising solutions to protect this data, which has been around since world war II, is using cryptography. There exist many cryptographic solutions to protect our data while it is in transit, for example secure communication channels implemented by SSL or TLS protocols that we use for our banking transactions. Also there are many cryptographic techniques available to protect data while it is at rest, such as encrypting the sensitive files before storing them on the hard drive. But how do we protect the data at the time that it is being used? Can we keep our data encrypted during its whole life cycle and still be able to use it and compute on it?

Secure multi-party computation is a branch of cryptography, with the goal of creating methods for parties to jointly compute a function over their sensitive inputs, while keeping those inputs private. That function can be a machine learning algorithm that multiple parties are interested to compute on their encrypted data. The theory of secure computation has been the topic of extensive research since the 1980's [1], and there is an immense line of work in the literature trying to improve the efficiency of these solutions, reducing the costs introduced by providing security. However, most of these results have remained impractical for real applications that deal with large-scale data.

The main goal in secure computation is to guarantee that nothing about individual's

private input should be learned during the computation by other parties, except the output. However, there is another line of research that aims to preserver the privacy of the output of computation, which is called Differential Privacy and was introduced by [2] in 2006. Differential privacy is a separate orthogonal question from secure computation, since they are traditionally two different areas and are concerned with two different problems. Differential privacy is concerned with *what is learned from the output of the computation*, while what we care about in secure computation is *what is learned from the process of the computation itself*.

In this dissertation, we aimed to study the interplay between secure computation and differential privacy in order to build efficient privacy-aware machine learning techniques. Part of this research focused on leveraging differential privacy to provide an efficiency versus privacy tradeoff in secure computation. We identified a class of highly parallelizable computations that could benefit from this privacy/efficiency tradeoff we propose. In the second part, we focused on constructing secure solutions in the setting of federated learning, in which secure computation is being used to protect the privacy of data during computation by an untrusted server, while differential privacy is used to protect the privacy of individual's after the global model is released.

1.1 Research Objectives

The main, broad objective of this dissertation is to tackle this primary question: "Can we design and develop privacy-preserving machine learning techniques that can compute on encrypted data at large scale? And while they preserve the privacy of user's private data, can they be efficient enough to be deployed in real world applications?" Our attempt to address this goal and answer the entailed research questions, resulted into several theoretical and technical contributions, as follows:

1. We explore the trade-off between security and efficiency in secure computation techniques in order to design more efficient secure machine learning algorithms. We introduce

the idea of using differential privacy to relax the security notion in secure computation, and define a new security model in which the adversary is provided some leakage that is proven to preserve differential privacy.

2. We show that this leakage allows us to construct a more efficient protocol for a broad class of computations. We identify a class of parallel computation schemes that could benefit from this relaxation, and build a privacy-preserving machine learning framework that works in two-party setting and is secure against semi-honest adversary.

3. We demonstrate that by increasing the number of computation servers from two to four, and changing the computation design from Boolean circuit to arithmetic circuits, we can build a secure parallel computation technique that is much faster in running time as compare to the previous scheme, and is also secure against stronger adversary who tries to deviate from the protocol.

4. We also explore cryptographic solutions to introduce security for another class of machine learning techniques that work in the distributed fashion, called Federated Leaning. We design and develop secure federated learning frameworks, in which secure computation is used to protect the learning parameters shared by users with the computation servers, and differential privacy is used to preserve the privacy of the global model against potential model-inversion or membership-inference attacks. Since the computation servers are not trusted, the DP noise sampling is handled by MPC solutions. Secret sharing and fully-homomorphic encryption schemes are the cryptographic primitives that are utilized to design and develop these frameworks. They are designed to be secure against semi-honest as well as malicious adversaries.

1.2 Thesis Outline

In Chapter 2, we define some of the fundamental concepts that build the foundation of the designed solutions during this research, such as secure computation, differential privacy, parallel computation, and federate learning. This chapter also provides a literature

review on these concepts. Chapter 3 presents our secure parallel-computation framework that is secure against honest-but-curious adversary. In Chapter 4, we design a parallel-computation framework that has stronger security assumption and can tolerate an actively malicious adversary. Both of these frameworks are designed for centralized machine learning setting. Chapter 5 introduces two secure frameworks that are designed in Federated Learning setting. One of them is secure against semi-honest adversaries while the other one can tolerate arbitrary number of malicious adversaries up to a threshold value. Chapter 6 summarizes the results achieved while conducting this research. In chapter 7, few suggestions are made to carry out further research on design and development of other secure machine learning algorithms that are popular and wildly used in the ML community such as Deep Learning.

Chapter 2: Background

This chapter reviews concepts and tools that will be used throughout this dissertation, and provides a literature review on each topic.

2.1 Definitions and Preliminaries

We start by outlining some fundamental concepts such as secure computation, differential privacy, graph parallel computation, and federated learning. We also introduce two main adversarial models and their differences.

2.1.1 Secure Computation

Secure multiparty computation protocols (MPC) is a branch of cryptography that makes it possible for a set of participants (parties) to compute a function over their sensitive inputs without revealing anything about those inputs to other parties, except the output of the computation. MPC has received significant research attention since the 1980's, when [1, 3] tried to establish the feasibility of generic MPC protocols. In recent years, there have been great efforts on realizing these theoretical results and researcher aimed to build MPC protocols that are practical and applicable for real applications Researchers have also introduced tailored MPC protocols for specific computations. Some researchers developed frameworks that are more generic by allowing functionalities to be expressed in a high-level language [4]. [5] conducted a comprehensive survey on compilers and frameworks developed for both customized and generic MPC.

2.1.2 Adversarial Model in MPC

The security of an MPC protocol can be measured by the number of corrupted parties that it can tolerate, and how strong are the corrupted parties considering their behavior. There is a threshold parameter that can determine the maximum number of corrupted parties that it can tolerate and still stay secure. How the corrupted party behave in the protocol can be modeled as a passive or active adversary. Passive adversary that is also known as semi-honest or honest-but-curious, is an adversary that follows the protocol honestly, however it tries to learn about honest parties private input by analyzing the transcript of the protocol and communicated messages. Active or malicious adversaries are considered to be stronger and more realistic. They usually do not follow the protocol exactly and even try to deviate from the protocol instruction by sending or behaving arbitrarily. Malicious secure protocols are usually less efficient as compared to semi-honest ones, and sometimes they need to have a mechanism to protect the privacy of honest users from corrupted ones.

2.1.3 Differential Privacy

Differential privacy provides a strong mathematical guarantees user's privacy for algorithms processing user's data. The main idea is that any statistical functions running on the database, should not overly depend on the data of any one individual. We say a computation is differentially private if the probability of producing a given output does not depend very much on whether a particular data point is included in the input dataset or not. Differential privacy was initially introduced in the ground-breaking work by Dwork et al. [6] and it's formal definition is as follows:

Definition 1. **((ϵ, δ)-*differential privacy [6])*** *A randomized mechanism $\mathcal{M}$ satisfies (ϵ, δ) - differential privacy if for any pair of neighboring datasets $D_1, D_2 \in D$ s.t. $|D_1 - D_2| \leq 1$, and for any subset of outputs $T \subseteq Range(\mathcal{M})$:*

$$\Pr[\mathcal{M}(D_1) \in T] \leq e^{\epsilon} \cdot \Pr[\mathcal{M}(D_2) \in T] + \delta$$

where the probability is taken over the coin tosses of $\mathcal{M}$.

The trade-off between the utility and privacy leakage of the mechanism M is controlled by adjusting the privacy budget parameter ϵ. A smaller the privacy budget represents a less privacy leakage and a stronger privacy level. The additional variant δ, allows for the possibility that ϵ-differential privacy is broken with probability δ, which is preferably smaller than $1/|d|$. If δ is 0, we say the mechanism M is ϵ-differentially private.

2.1.4 Graph-parallel computation

The Graph-parallel abstraction as it is used in several frameworks such as MapReduce [7], GraphLab [8] and PowerGraph [9], consists of a sparse graph that encodes computation as *vertex-programs* that run in parallel, and interact along edges in the graph. These frameworks all follow the same computational model, called the GAS model, which includes three conceptual phases: Gather, Apply, and Scatter. The framework is quite general, and captures computations such as gradient descent, which is used in matrix factorization for recommendation systems, as well as histograms or counting operation, and many other computations. In Matrix Factorization, as an example, an edge $(u, v, data)$ indicates that user u reviewed item v, and the $data$ stored on the edge contains the value of the user's review. The computation proceeds in iterations, and in each iteration, every node gathers (copy) data from their incoming edges, applies some computation to the data, and then scatters (copy) the result to their outgoing edges. Viewing each vertex as a CPU or by assigning multiple vertices to each CPU, the apply phase which computes the main functionality, is easily parallelized. [10, 11] constructed frameworks for securely computing graph-parallel algorithms. They did this by designing a nicely parallelizable circuit for the gather and scatter phases.

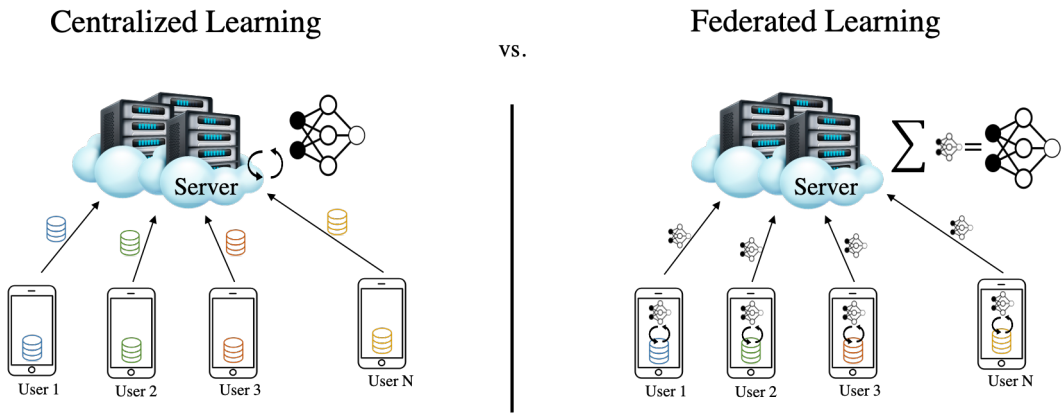


Figure 2.1: Centralized Machine Learning vs. Federated (Distributed) Learning

2.1.5 Federated (Collaborative) Learning

Federated Learning has recently emerged as an alternative to centralized ML algorithms. The concept of federated machine learning received significant attention after it first was coined by Google [12, 13]. It is a learning techniques that allows the training of a high quality centralized model over decentralized data which can be scattered across multiple edge devices. The procedure is that, each party trains a local model on its sensitive data and only shares the parameter updates with the server (curator). In the centralized setting, the model takes advantage of having higher accuracy by collecting a large amount of data at one point for training. However, it also brings high storage and computation load to the centralized server, and once any attacks happens on the server, or if it behaves dishonestly, all individual data will be at risk. Moreover, the privacy of the ML model and data are dependent of how the ML model is used, and also the extent of the adversary's access to the system hosting the model and data. In the collaborative setting, however, the input data is still in the possession of the individuals, but the adversary can observe the communication between the users and the curator who is only receiving the learning weights.

2.2 Related Works

Hiding access pattern in secure computation. One of the key challenges in secure computation is to prevent any leakage in the form of memory access pattern during the computation. Although the circuit model of computation can handle this problem, as they are data oblivious solutions, there are more promising ways to circumvent this issue when computing on large-scale data. Oblivious RAMs or so called ORAM provides more efficient solutions to hide access pattern during computation, and has its own extensive line of research [14–19]. However, in applications that deal with large-scale data, in most of the cases, both circuits and ORAM-based solutions are too slow for practical requirements, and creates a great demand to find better alternatives.

Allowing differentially private leakage. Wagh et al. [20] define and construct differentially private ORAM in which the server's views are 'similar' on two neighboring access patterns. They consider the client/server model, and don't consider using their construction in a secure computation. Recently, researchers have explored the idea of relaxing security to allow leakage in secure computation, coupled with a bound demonstrating that the leakage preserves differential privacy [21, 22]. Chan et al. study differential obliviousness in the client/server model [21]. They also show asymptotic improvement for several computations, together with lower bounds for fully secure variants of the same algorithms, demonstrating that this relaxation allows us to bypass impossibility results. Their results are purely theoretical, but raise the very interesting question of whether we can lower-bound the number of AND gates needed in fully secure graph parallel computation.

Parallelizing secure computation. One of the most relevant work to our research in this direction, is that by Nayak et al. [11], which generalizes the work of Nikolaenko et al. [10], computing graph parallel computations with full security in a circuit model of computation. Papadimitriou et al. [23] also build a system for the secure computation of graph-structured data, and ensure differential privacy of the output. They do not consider differentially private leakage in the access patterns. While our proposed security relaxation

using differential privacy is appealing, it is not immediately clear that it provides a natural way to improve efficiency. When computing on plaintext data, frameworks such as MapReduce, Pregel, GraphLab and PowerGraph have very successfully enabled developers to leverage large networks of parallelized CPUs [7–9, 24]. The latter three mentioned systems are specifically designed to support computations on data that resides in a graph, either at the nodes or edges. The computation proceeds by iteratively gathering data from incoming edges to the nodes, performing some simple computation at the node, and pushing the data back to the outgoing edges. This simple iterative procedure captures many important computational tasks, including histogram, gradient descent and page-rank, as well as Markov random field parameter learning, parallelized Gibbs samplers, and name entity resolution, to name a few more.

When federated learning meets secure computation and differential privacy. Federated Learning has recently emerged as an alternative to centralized ML algorithms. Google has conducted a comprehensive survey study on Federate Learning, its challenges and its open problems [25]. It allows multiple participants to jointly build a model on their own local training set. Each participant trains a local model on its own data and exchange these parameters with other participants (sometimes through a server or curator) to train a global model. Several architectures have been proposed for federated learning [26–30] with and without a central server. One of the main goals in developing these frameworks, is protecting the privacy of participants in the training process [13,31]. Because the training data never leave the participants' local device, federated learning can be a good candidate for the scenarios where the data is highly sensitive and cannot be shared with untrusted parties, for example foe some of the GDPR-aware applications. Even-though these parameter updates are ephemeral and very small as compared to the user's high-dimensional private data vectors, observing these parameters by the untrusted server or any other entity, or even in some scenarios the model itself may still leak important information about user input, hence they can lead to potential adversarial attacks such as the model inversion attacks [32] or membership inference

attacks [33].

One of the ways to mitigate these attacks, is to use cryptography techniques to securely share these updates with the server, and then using secure computation techniques to aggregate those parameters in order to update the global model. Secure aggregation protocols presented by [34–36] allow the untrusted central server to only learn the summation of the input vectors of many clients securely. Their protocol is robust against a fraction of users dropping out. [36] improved the efficiency of the previous secure aggregation protocols and constructs secure aggregation protocols that achieve polylogarithmic communication and computation per client. Their semi-honest construction handles billions of clients and semi-malicious construction supports tens of thousands of clients for the same per client cost. Their solutions have low-communication cost, but do not handle aggregating noisy learning parameters. Handling noise aggregation is trivial if we know how many people participate from the beginning, but much more subtle if clients frequently join and dropout during the protocol.

One of the recent works on the idea of collaborative learning with privacy protection is proposed by Shokri and Shmatikov [31]. In every iteration of training, each participant downloads the global model from the parameter server, locally computes gradient updates based on one batch of her training data, and sends the updates to the server. As a result, they can benefit from other participants who are concurrently learning similar models. The server waits for the gradient updates from all participants and then applies the aggregated updates to the global model, using stochastic gradient descent (SGD). But still this scheme does not protect the training parameters using cryptographic primitives, from the adversaries that are monitoring the communication channel and are observing the noisy values of the user's local parameters.

Chapter 3: Privacy-Preserving Parallel Machine Learning Computation in Semi-Honest Settings [37]

In this chapter, we present our secure and privacy-preserving machine learning framework based on secure computation techniques in semi-honest adversarial model. We explore a new security model for secure computation on large datasets. We assume that two servers have been employed to compute on private data that was collected from many users, and, in order to improve the efficiency of their computation, we establish a new trade-off with privacy. Specifically, instead of claiming that the servers learn nothing about the input values, we claim that what they do learn from the computation preserves the differential privacy of the input. Leveraging this relaxation of the security model allows us to build a protocol that leaks some information in the form of access patterns to memory, while also providing a formal bound on what is learned from the leakage. We then demonstrate that this leakage is useful in a broad class of computations. We show that computations such as histograms, PageRank and matrix factorization, which can be performed in common graph-parallel frameworks such as MapReduce or Pregel, benefit from our relaxation. We implement a protocol for securely executing graph-parallel computations, and evaluate the performance on the three examples just mentioned above. We demonstrate marked improvement over prior implementations for these computations.

3.1 Our Main Framework Intuition

To illustrate our main idea, we describe an algorithm that computes the data histogram (i.e. counting, or data frequency) with differentially private access patterns. Although this computation can be formalized in the context of our general framework, it is instructive to

demonstrate some of the main technical ideas with this simple example before considering how they generalize (which we do in Section 3.3.2). We defer a discussion about security until we present the more general protocol.

3.1.1 A Differentially Private Protocol for Computing Histograms

To illustrate our main idea, we describe an algorithm that computes the data histogram (i.e. counting, or data frequency) with differentially private access patterns. Although this computation can be formalized in the context of our general framework, it is instructive to demonstrate some of the main technical ideas with this simple example before considering how they generalize (which we do in Section 3.3.2). We defer a discussion about security until we present the more general protocol.

In this computation, we assume that each user in the system contributes a single input value, $x_i \in S$, where we call the set S the set of *types*. The computation servers (parties) each begin the computation with secret shares of the input array, denoted by $\langle \mathsf{real} \rangle$. The output is a secret share of $|S|$ counters, where the counter for each type contains the exact number of inputs of that type. The full protocol specification appears in Figure 3.1.

The protocol is in a hybrid model, where the parties have access to three ideal functionalities: $\mathsf{DumGen}_{p,\alpha}, \mathcal{F}_{\mathsf{shuffle}}, \mathcal{F}_{\mathsf{add}}$. The two parties begin by calling $\mathsf{DumGen}_{p,\alpha}$, which generates some number of dummy inputs. The ideal functionality for this is described in the left of Figure 3.2, and it is realized using a generic secure two-party computation. As part of this computation, the parties have to securely sample from the distribution $\mathbb{D}_{p,\alpha}$. In the next section, we define this distribution and describe our method for sampling it. We simply remark now that it has integer support, and is negative with only negligible probability (in δ). The output of $\mathsf{DumGen}_{p,\alpha}$ is a secret sharing of values in $S \cup \{\perp\}$: the size of the output is $2\alpha|S|$, where α is determined by the desired privacy values ϵ and δ (see Section 3.3.2). The number of dummy items of each type is random, and neither party should learn this value; shares of $\perp$ are used to pad the number of dummy items of each type until they

total 2α.

Each party locally concatenates their share of the real input array with their share of the dummy values. They also initialize shares of an array of flags, denoted as isReal, which will be used to keep track of which item is real and which is dummy. They then shuffle the real and dummy items together using an oblivious shuffle. This is presented as an ideal functionality, but in practice we implement this using two sequential, generic secure computations of the Waksman permutation network [38], where each party randomly choose one of the two permutations. The same permutations are used to shuffle the array isReal flags, ensuring that these flags are "moved around with" the items. We note that all secret shares are updated during the process of shuffling, so while the parties knew which items and flags were real and which were not before the shuffle, they have no way of knowing this after they receive fresh shares of the shuffled items and isReal flags.

The parties now open their shares of the data types, while leaving the flag values unknown. This is where our protocol leaks some information: revealing the data types allows the parties to see a noisy sum of the number inputs of each type. On the other hand, this is also where we gain in efficiency: the remainder of the protocol requires only a linear scan over the data array, with a small secure computation for each element in order to update the appropriate counter value. More specifically, the parties iterate through the shuffled array, opening each type. On data type i, they fetch their shares of the counter for type i from memory, and call the $\mathcal{F}_{\mathsf{add}}$ functionality. This functionality adds the (reconstructed) flag value to the (reconstructed) counter; if the item was a real item, the counter is incremented, while if it was a dummy item, the counter remains the same. The functionality returns fresh shares of the counter value. Neither party ever learns whether the counter was updated. In particular, they cannot know whether they fetched that counter from memory because of a real input value, or because of a dummy value. In our implementation, we instantiate $\mathcal{F}_{\mathsf{add}}$ with a garbled circuit.

Simple extensions: In Section 3.3.2 we show how to generalize this protocol to the wider

Differentially Private Histogram Protocol

Input: Each party, P_1 and P_2, receives a secret-share of real items denoted as $\langle \mathsf{real} \rangle$ (r stands for number of real items, and d for number of dummy ones)

Output: Secret share of counter values denoted as $\langle \mathsf{Counter} \rangle$, where the counter for each type contains the exact number of inputs of that type (S is the number of counter types)

Preprocessing:

$\langle \mathsf{Counter} \rangle_{1:|S|} \leftarrow 0$

Computation:

$\langle \mathsf{dummy} \rangle_{1:d} \leftarrow \mathsf{DumGen}_{p,\alpha}$

$\langle \mathsf{data} \rangle_{1:(r+d)} = \langle \mathsf{real} \rangle_{1:r} || \langle \mathsf{dummy} \rangle_{1:d}$

$\langle \mathsf{isReal} \rangle_{1:r} \leftarrow 1$, $\langle \mathsf{isReal} \rangle_{(r+1):(r+d)} \leftarrow 0$

$\langle \widehat{\mathsf{data}} \rangle \leftarrow \mathcal{F}_{\mathsf{shuffle}}(\langle \mathsf{data} \rangle, \langle \rho \rangle)$

$\langle \widehat{\mathsf{isReal}} \rangle \leftarrow \mathcal{F}_{\mathsf{shuffle}}(\langle \mathsf{isReal} \rangle, \langle \rho \rangle)$

$\widehat{\mathsf{data}} \leftarrow \mathsf{Open}(\langle \widehat{\mathsf{data}} \rangle)$

for $i = 1 \ldots (n + d)$

$\mathcal{F}_{\mathsf{add}}(\langle \mathsf{isReal} \rangle_i, \langle \mathsf{Counter} \rangle_t)$ where $t = \widehat{\mathsf{data}}_i$

Figure 3.1: A protocol for two parties to compute a histogram on secret-shared data with an access pattern that preserves differential privacy.

function class. However, we note that in this specific case, if we did want to add noise to the output, we could simply instruct the servers to count the number of times each counter is accessed. They would no longer have to update the counter values through a secure computation, so this would be a (slightly) faster protocol. The output would contain the one-sided noise, but they could simply subtract off α from each counter to get a more accurate estimate of the counts. We stress that in this modified protocol, the dummy items are still shuffled in with the real items, so the access pattern still preserves differential privacy for each user. The modification ensures that the (reconstructed) output preserves differential privacy as well.

We also note that the protocol in Figure 3.1 can be applied to other similar computations, such as taking averages or sums over r values of $|S|$ types (though, now again *without* adding noise to the output). For example, if each user contributed a salary value and a zip-code, we could use the above method for computing the average salary in each zip-code, while ensuring that the access patterns preserve user privacy. We simply need to modify the $\mathcal{F}_{\mathsf{add}}$ functionality: instead of incrementing the secret-shared counter by 1 when the input is a real item, the functionality would increment the counter by the value of the secret-shared salary. In this case, though, the noisy access pattern alone does not suffice for creating noisy output: the use of $\mathcal{F}_{\mathsf{add}}$ is essential. If we want to ensure that the reconstructed output preserves privacy, the noise would have to be generated independently, through a secure computation, and then added obliviously to the output.

3.2 Notations and Definitions

Secret-Shares: We let $\langle x \rangle$ denote a variable which is XOR secret-shared between parties. Arrays have a public length and are accessed via public indices; we use $\langle x \rangle_i$ to specify element i within a shared array, and $\langle x \rangle_{i:j}$ to indicate a specific portion of the array containing elements i through j, inclusive. When we write $\langle x \rangle \leftarrow c$, we mean that both users should fix their shares of x (using some agreed upon manner) to ensure that $x = c$. For example, one party might set his share to be c while the other sets his share to 0.

Multi-Sets: We represent multi-sets over a set V by a $|V|$ dimensional vector of natural numbers: $D \in \mathbb{N}^{|V|}$. We refer to the ith element of this vector by $D(i)$. We use $|D|$ in the natural way to mean $\sum_{i=1}^{|V|} D(i)$. We use $\mathcal{DB}_i$ to denote the set of all multi-sets over V of size i, and $\mathcal{DB} = \bigcup_i \mathcal{DB}_i$. We define a metric on these multi-sets in the natural way: $|D_1 - D_2| = \sum_{i=1}^{|V|} |D_1(i) - D_2(i)|$. We say two multi-sets are *neighboring* if they have distance at most 1: $|D_1 - D_2| \leq 1$.

Neighboring Graphs: In our main protocol of Section 3.3.2, the input is a data-augmented

directed graph, denoted by $G = (V, E)$, with user-defined data on each edge. We need to define a metric on these input graphs, in order to claim security for graphs of bounded distance. In section 3.1.1, the input to the computation is a multi-set of elements drawn from some set S, rather than a graph, so we use the simple distance metric described above to define the distance between inputs. For each $v \in V$, we let $\mathsf{in\text{-}deg}(v)$ and $\mathsf{out\text{-}deg}(v)$ denote the in-degree and out-degree of node v. We define the *in-degree profile* of a graph G as the multi-set $\mathsf{D_{in}}(G) = \{\mathsf{in\text{-}deg}(v_1), \ldots, \mathsf{in\text{-}deg}(v_n)\}$. Intuitively, this is a multi-set over the node identifiers from the input graph, with vertex identifier v appearing k times if $\mathsf{in\text{-}deg}(v) = k$. We define the *full-degree profile* of G as the pair of multi-sets: $\{\mathsf{D_{in}}(G), \mathsf{D_{out}}(G)\}$, where $\mathsf{D_{out}}(G) = \{\mathsf{out\text{-}deg}(v_1), \ldots, \mathsf{out\text{-}deg}(v_n)\}$. We now define two different metrics on graphs, using these degree profiles. Later in this section, we provide two different security definitions: we rely on the first distance metric below when claiming security as defined in Definition 9, and rely on the second metric below when claiming security as defined in Definition 7.

Definition 2. *We say two graphs G and G' have distance at most d if they have in-degree profiles of distance at most d: $|\mathsf{D_{in}}(G) - \mathsf{D_{in}}(G')| \leq d$. We say that G and G' are* neighboring *if they have distance 1.*

Definition 3. *We say two graphs G and G' have full-degree profiles of distance d if the sum of the distances in their in-degree profiles and their out-degree profiles is at most d: $|\mathsf{D_{in}}(G) - \mathsf{D_{in}}(G')| + |\mathsf{D_{out}}(G) - \mathsf{D_{out}}(G')| \leq d$. We say that G and G' have* neighboring full-degree profiles *if they have full-degree profiles of distance 2.*

Differential Privacy definition for Graph datasets: We use the definition that appears in [6], to define differential privacy for neighboring datasets in graphs and in multi-set.

Definition 4. *A randomized algorithm $\mathcal{L} : \mathcal{DB} \to \mathcal{R}_{\mathcal{L}}$, with an input domain $\mathcal{DB}$ that is the set of all multi-sets over some fixed set V, and output $\mathcal{R}_{\mathcal{L}} \subset \{0,1\}^*$, is (ϵ, δ)-differentially private if for*

all $T \subseteq \mathcal{R}_{\mathcal{L}}$ and $\forall D_1, D_2 \in \mathcal{DB}$ such that $|D_1 - D_2| \leq 1$:

$$\Pr[\mathcal{L}(D_1) \in T] \leq e^{\epsilon} \Pr[\mathcal{L}(D_2) \in T] + \delta$$

where the probability space is over the coin flips of the mechanism $\mathcal{L}$.

The above definition describes differential privacy for neighboring multi-sets. Letting $\mathcal{G}$ denote the set of all graphs, we define it for neighboring graphs as well:

Definition 5. *A randomized algorithm $\mathcal{L} : \mathcal{G} \rightarrow \mathcal{R}_{\mathcal{L}}$ is (ϵ, δ)-edge private if for all neighboring graphs, $G_1, G_2 \in \mathcal{G}$, we have:*

$$\Pr[\mathcal{L}(G_1) \in T] \leq e^{\epsilon} \Pr[\mathcal{L}(G_2) \in T] + \delta$$

Input model: We try to keep the definitions general, as we expect they will find application beyond the space of graph-structured data. However, we use notation that is suggestive of computation on graphs, in order to keep our notation consistent with the later sections. We assume that two computation servers have been entrusted to compute on behalf of a large set of users, $\mathcal{V}$, with $|\mathcal{V}| = n$, and having sequential identifiers, $1, \ldots, n$. Each user i contributes data v_i. They might each entrust their data to one of the two servers (we call this the *disjoint collection setting*), or they might each secret-share their input with the two-servers (*joint collection setting*). In the latter case, we note that both servers learn the size of each v_i but neither learns the input values; in the former case, each server learns a subset of the input values, but learns nothing about the remaining input values (other than the sum of their sizes). We note that the disjoint collection setting corresponds to the "standard" setting for secure computation where each computing party contributes one set of inputs. Just as in that setting, each of the two computing parties could pad their inputs to some maximum size, hiding even the sum of the user input sizes. In fact, we could have them pad their inputs using a randomized mechanism that preserves differential privacy, possibly leading to smaller padding sizes, depending on what the maximum and average

input sizes are. We don't explore this option further in this work. Below we will define two variant security notions that capture these two scenarios.

In all computations that we consider in our constructions, the input is represented by a graph. In every case, each user is represented as a node in this graph, and each user input is a set of weighted, directed edges that originate at their node. In some applications, the graph is bipartite, with user nodes on the left, and some distinct set of item nodes on the right: in this case, all edges go from user nodes to item nodes. In other applications, there are only user nodes, and every edge is from one user to another. In the joint collection setting, we can leak the out-degree of each node, which is the same as the user input size, but must hide (among other things) the in-degree of each node. In the disjoint collection setting, the protocol must hide both the in-degree and out-degree of each node. We note that in some applications, such as when we perform gradient descent, the graph is bipartite, and it is publicly known that the in-degree of every user is 0 (i.e. the movies don't review the viewers). In the joint collection setting, this knowledge allows for some improvement in efficiency that we will leverage in Section 4.8.

Defining secure computation with leakage: For simplicity, we start with a standard definition of semi-honest security, but make two important changes. We stress that our allowance of differentially private leakage brings gains in the *circuit construction*, so we could use *any* generic secure computation of Boolean circuits, including those that are maliciously secure, and benefit from the same gains. The first change is that we allow certain leakage in the ideal world, in order to reflect what is learned by the adversary in the real world through the observed access pattern on memory. The leakage function is a randomized function of the inputs. The second change is an additional requirement that this leakage function be proven to preserve the differential privacy for the users that contribute data. Our ideal world experiment is as follows. There are two parties, P_1 and P_2, and an adversary $\mathcal{S}$ that corrupts one of them. The parties are given input, as described above; we use V_1 and V_2 to

denote the inputs of the computing parties, regardless of whether we are in the joint collection setting or the disjoint collection setting, and we let $V = \{v_1, \ldots, v_n\}$ denote the user input. Technically, in the joint collection setting, $V = V_1 \oplus V_2$, while in the disjoint collection setting, $V = V_1 \cup V_2$. Each computing party submits their input to the ideal functionality, unchanged. The ideal functionality reconstructs the n user inputs, $v_1, \ldots, v_n$, either by taking the union of the inputs submitted by the computation servers in the disjoint collection setting, or by reconstructing the input set from the provided secret shares in the joint collection setting. The ideal functionality then outputs $f_1(v_1, \ldots, v_n)$ to P_1 and $f_2(v_1, \ldots, v_n)$ to P_2. These outputs might be correlated, and, in particular, in our own use-cases, each party receives a secret share of a single function evaluation: $\langle f(v_1, \ldots, v_n)\rangle_1, \langle f(v_1, \ldots, v_n)\rangle_2$. The ideal functionality also applies some leakage function to the data, $\mathcal{L}(V)$, and provides the output of $\mathcal{L}(V)$, along with $\sum_{i \in \mathcal{V}} |v_i|$ to $\mathcal{S}$. In the joint collection setting, the simulator can infer this value from the size of the input that was submitted to the ideal functionality. But it simplifies things to give it to him explicitly. Additionally, depending on the choice of security definition, the ideal functionality might or might not give the simulator, $\forall i \in \mathcal{V}$, $|v_i|$.

Our protocols are described in a *hybrid world*, in which the parties are given access to several secure, ideal functionalities. In our implementation, these are replaced using generic constructions of secure computation (i.e. garbled circuits). Relying on a classic result of Canetti [39], when proving security, it suffices to treat these as calls to a trusted functionality. In the definitions that follow, we let $\mathcal{G}$ denote an appropriate collection of ideal functionalities. As is conventionally done in the literature on secure computation, we let $\text{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}(V_1, V_2, \kappa)$ denote a joint distribution over the output of the honest party and, the view of the adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$, when the parties interact in the hybrid protocol $\pi^{\mathcal{G}}$ on inputs V_1 and V_2, each held by one of the two parties, and computational security parameter κ. We let $\text{IDEAL}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\forall i \in V:|v_i|)}(V_1, V_2, \kappa)$ denote the joint distribution over the output of the honest party, and the view output by the simulator $\mathcal{S}$

with auxiliary input $z \in \{0,1\}^*$, when the parties interact with an ideal functionality $\mathcal{F}$ on inputs V_1 and V_2, each submitted by one of the two parties, and security parameters κ. Letting $v = \sum_{i \in V} |v_i|$), we define the joint distribution $\text{IDEAL}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),v)}(V_1, V_2, \kappa)$ in a similar way, the only difference being that the simulator is given the sum of the input sizes and not the value of each input size.

Definition 6. *Let $\mathcal{F}$ be some functionality, and let π be a two-party protocol for computing $\mathcal{F}$, while making calls to an ideal functionality $\mathcal{G}$. π is said to* securely compute $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with $\mathcal{L}$ leakage, known input sizes, and $(\kappa, \epsilon, \delta)$-security *if $\mathcal{L}$ is (ϵ, δ)-differentially private, and, for every PPT, semi-honest, non-uniform adversary $\mathcal{A}$ corrupting a party in the $\mathcal{G}$-hybrid model, there exists a PPT, non-uniform adversary $\mathcal{S}$ corrupting the same party in the ideal model, such that, on any valid inputs V_1 and V_2*

$$\left\{\text{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}(V_1, V_2, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv}$$

$$\left\{\text{IDEAL}^{(1)}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\forall i\in V:|v_i|)}(V_1, V_2, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \tag{3.1}$$

The above definition is the one that we use in our implementations. However, in Section 3.3.2 we also describe a modified protocol that achieves the stronger security definition that follows, where the adversary does not learn the sizes of individual inputs. This property might be desirable (or maybe even essential) in the disjoint collection model, where users have not entrusted one of the two computing parties with their inputs, or even the sizes of their inputs. On the other hand, the previous definition is, in some sense, more "typical" of definitions in cryptography, where we assume that inputs sizes are leaked. It is only in this model where data is outsourced that we can hope to hide each individual input size among the other inputs.

Definition 7. *Let $\mathcal{F}$ be some functionality, and let π be a two-party protocol for computing $\mathcal{F}$, while making calls to an ideal functionality $\mathcal{G}$. π is said to* securely compute $\mathcal{F}$ in the $\mathcal{G}$-hybrid

model with $\mathcal{L}$ leakage, and $(\kappa, \epsilon, \delta)$-security *if* $\mathcal{L}$ *is* (ϵ, δ)*-differentially private, and, for every PPT, semi-honest, non-uniform adversary* $\mathcal{A}$ *corrupting a party in the* $\mathcal{G}$*-hybrid model, there exists a PPT, non-uniform adversary* $\mathcal{S}$ *corrupting the same party in the ideal model, such that, on any valid inputs* V_1 *and* V_2

$$\left\{\text{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}\left(V_1, V_2, \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv}$$

$$\left\{\text{IDEAL}^{(2)}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\sum_{i\in V}|v_i|)}(V_1, V_2, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \tag{3.2}$$

Differentially Private Output: As is typical in secure computation, we are concerned here with *how* to securely compute some agreed upon function, rather than *what* function ought to be computed. In other words, we view the question of what the output itself might reveal about the input to be beyond scope of our work. Our concern is only that the process of computing the output should not reveal too much. Nevertheless, one could ask that the output of all computations also be made to preserve differential privacy. Interestingly, for the specific case of histograms, which we present as an example in Section 3.1.1, adding differentially private noise to the output is substantially *more efficient* than preserving an exact count. This is not true for the general protocol, but the cost of adding noise for these cases has been studied elsewhere [23], and it would be minor compared to the rest of the protocol.

Nevertheless, we take a different approach. In all of our computations, the output of each server is a secret share of the desired output, and thus it is unconditionally secure. The question of where to deliver these shares is left to the user, though we can imagine several useful choices. Perhaps most obvious, the shares might never be reconstructed, but rather used later inside another secure computation that makes decisions driven by the output. Or, as Nikolaenko et al. suggest [10], when computing gradient descent to provide users with recommendations, the recommendation vectors can be sent to the user to store for

themselves. Regardless, since the aim of our work is to study the utility of our relaxation, this concern is orthogonal, and we mainly leave it alone.

Privacy versus efficiency: In the "standard" settings where differential privacy is employed, additional noise affects the accuracy of the result. Here, added noise has no impact on the output, which is always correct, and is protected by the secure computation. Instead, the tradeoff is with efficiency: using more noise helps to further hide the true memory accesses among the fake ones, but requires additional, costly oblivious computation.

Malicious security and multi-party computation: Extending these definitions to model malicious adversaries and/or multi-party computation is straightforward, so we omit redundant detail. Similarly, we stress that by leveraging the security relaxation defined above, we gain improvement *at the circuit level*, so we can easily extend our protocols to either (or both) of these two settings in a generic way. To make our protocol from Section 3.3.2 secure against a malicious adversary, the only subtlety to address is that our protocols make iterative use of multiple secure computations (i.e. the functionality we realize is reactive), so we would need to authenticate outputs and verify inputs in each of these computations. While this can be done generically, such authentication comes "for free" in many common protocols for secure computation (e.g. [40, 41]). To extend our protocols to a multiparty setting, the only subtlety is in constructing a multiparty oblivious shuffle. With a small number of parties, c, it is very efficient to implement c iterations of a permutation network, where in each iteration, a different party chooses the control bits that determine the permutation. As c grows, it becomes less clear what the best method is for implementing an oblivious shuffle. Interestingly, we note that there has been some recent work on parallelizing multi-party oblivious shuffle [42]. We do not explore this direction in our work; presenting our protocols in the two-party, semi-honest setting greatly simplifies the exposition, and suffices to demonstrate the advantages of our security relaxation. In our performance analysis, we primarily focus on counting the number of AND gates in our construction, which makes the analysis more general and allows for more accurate comparison with prior work (than,

say, comparing the timed performance of systems that use different frameworks for implementing secure computation).

3.3 Our Main Framework Construction

In this section, we describe our protocol for graph structured data, and the graph-parallel frameworks that support highly parallelized computation.

When considering how the protocol from the previous section might be generalized, it is helpful to recognize the essential property of the computation's access pattern that we were leveraging. When computing a histogram, the access pattern to memory exactly leaks a histogram of the input! This might sound like a trivial observation, but it is in fact fairly important, as histograms are the canonical example in the field of differential privacy, and finding other computations where the access pattern reveals a histogram of the input will allow us to broadly apply our techniques.

With that in mind, we extend our techniques to graph structured data, and the graph-parallel frameworks that support highly parallelized computation.

There are several frameworks of this type, including MapReduce, Pregel, GraphLab and others [7,8,24]. We describe the framework by Gonzalez et al. [9] called PowerGraph since it combines the best features from both Pregel and GraphLab. PowerGraph is a graph-parallel abstraction, consisting of a sparse graph that encodes computation as *vertex-programs* that run in parallel and interact along edges in the graph. While the implementation of vertex-programs in Pregel and GraphLab differ in how they collect and disseminate information, they share a common structure called the GAS model of graph computation. The GAS model represents three conceptual phases of a vertex-program: Gather, Apply, and Scatter. The computation proceeds in iterations, and in each iteration, every node gathers (copy) data from their incoming edges, applies some simple computation to the data, and then scatters (copy) the result to their outgoing edges. Viewing each node as a CPU (or by assigning multiple nodes to each CPU), the apply step, which constitutes the bulk of the

computational work, is easily parallelized.

3.3.1 A Connection to Differential Privacy

The memory access pattern induced by this computation is easily described: during the gather stage, each edge is touched when fetching the data, and the adjacent node is touched when copying the data. A similar pattern is revealed during the scatter phase. (The computation performed during the apply phase is typically very simple, and can be executed in a circuit, which is memory oblivious.) Let's consider what might be revealed by this access pattern in some concrete application. In our framework, each user is represented by a node in the graph, and provides the data on the edges adjacent to that node. For example, in a recommendation system, the graph is bipartite, each node on the left represents a user, each node on the right represents an item that users might review, and the edges are labeled with scores indicating the user's review of an item. The access pattern just described would reveal exactly which items every user reviewed!

Our first observation is that if we use a secure computation to obliviously shuffle all of the edges in between the gather and scatter phases, we break the correlation between the nodes. Now the only thing revealed to the computing parties is a *histogram* of how many times each node is accessed – i.e. a count of each node's in-degree and out-degree. When building a recommendation system, this would reveal how many items each user reviewed, as well as how many times each item was reviewed. Fortunately, histograms are the canonical problem for differential privacy. Our second observation is that we can shuffle in dummy edges to help obscure this information, and, by sampling the dummy edges from an appropriate distribution (which has to be done within a secure computation), we can claim that the degrees of each node remain differentially private.

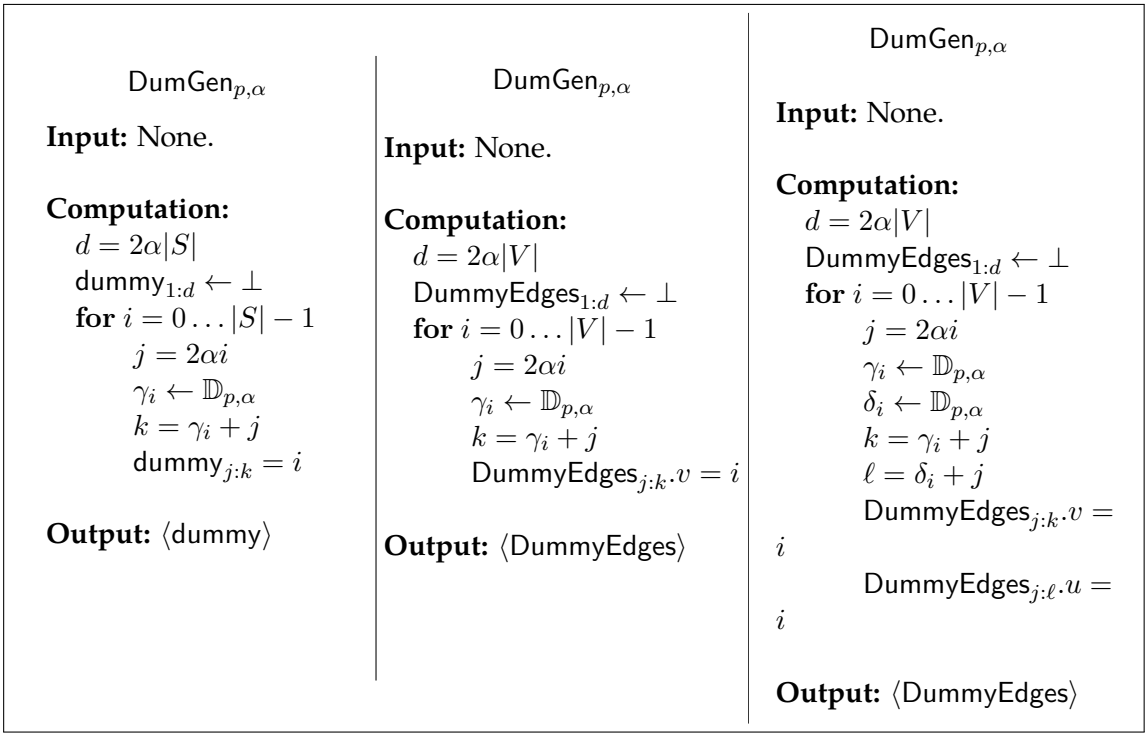


Figure 3.2: Three variations on the Ideal functionality, DumGen$_{p,\alpha}$. Each is parameterized by α, p. The leftmost functionality is used in the histogram protocol described in Section 3.1.1. The middle definition is the one used in our implementation, and suffices for satisfying security according to Definition 9. The right-most adds differential privacy to out-degrees, which is needed in the disjoint collection model (i.e. when hiding the input sizes for all users, in Definition 7).

3.3.2 The OblivGraph Protocol

When performing such computations securely, the data is secret-shared between the computing servers as it moves from edge to node and back, as well as during the Apply phase. The Apply phase is performed on these secret shares using any protocol for secure computation as a black-box. The main challenge is to hide the movement of the data during the Gather and Scatter phases, as these memory accesses reveal substantial information about the user data.

Take matrix factorization as an example: an edge (u, v, Data) indicates that user u reviewed item v, and the data stored on the edge indicates the value of the user's review.

Because the data is secret shared, the value of the review is never revealed. During the Gather phase, the right vertex of every edge is opened, and the data is moved to the corresponding vertex. After the Apply phase, the left vertex of every edge is open, and data is pulled back to the edge. If this data movement were performed in the clear, the memory access pattern would reveal the edges between nodes, exactly revealing which users reviewed which items. Our first observation is that, because we touch only the right node of every edge during the gather, and only the left node of every edge during the scatter, by adding an oblivious shuffle of the edges between these two phases, we can hide the connection between neighboring nodes. The leakage of the computation is then reduced to two histograms: the in-degrees of each node, and, after the shuffle, the out-degrees of each node!

Histograms are the canonical problem in differential privacy; we preserve privacy by adding noise to these two histograms, just as we do in Section 3.1.1. Details follow below, the formal protocol specification appears in Figure 3.4, and the ideal functionality for the PowerGraph framework appears in Figure 3.3.

We denote the data graph by $G = (V, E)$. The structure of each edge is comprised of $(\mathsf{u}, \mathsf{v}, \mathsf{uData}, \mathsf{vData}, \mathsf{isReal})$, where isReal indicates if an edge is "real" or "dummy". Each vertex is represented as $(\mathsf{x}, \mathsf{xData})$. The xData field is large enough to hold edge data from multiple adjacent edges. As in Section 3.1.1, our protocol is in a hybrid model where we assume we have access to three ideal functionalities: $\mathsf{DumGen}_{p,\alpha}$, $\mathcal{F}_{\mathsf{shuffle}}$, $\mathcal{F}_{\mathsf{func}}$. As compared to Section 3.1.1, here we have dropped an explicit specification of the permutation used in $\mathcal{F}_{\mathsf{shuffle}}$.

During the initialization phase, the $\mathsf{DumGen}_{p,\alpha}$ functionality is used to generate secret-shares of the dummy edges. These are placed alongside the real edges, and are then repeatedly shuffled in with the real edges during the iterative phases. We describe $\mathsf{DumGen}_{p,\alpha}$ in detail later in this section. Every call to $\mathcal{F}_{\mathsf{shuffle}}$ uses a new random permutation. (Since the dummy flags are now included inside the edge structure, we no longer need to specify that

$\mathcal{F}_{\mathsf{gas}}$

GAS Model Operations

Inputs: Secret share of edges denoted as $\langle \mathsf{Edges} \rangle$, each edge is $\mathsf{edge} : (\mathsf{u}, \mathsf{v}, \mathsf{uData}, \mathsf{vData}, \mathsf{isReal})$. Secret share of vertices denoted as $\langle \mathsf{Vertices} \rangle$, each vertex contains $\mathsf{vertex} : (\mathsf{x}, \mathsf{xData})$

Outputs: Updated $\langle \mathsf{Vertices} \rangle$

```
Gather(Edges)
    for each edge ∈ Edges
      for each vertex ∈ Vertices
        if edge.v == vertex.x
          vertex.xData ← copy(edge.uData)

Apply_f(Vertices)
    for each vertex ∈ Vertices
      vertex ← f(vertex)

Scatter (Edges)
    for each edge ∈ Edges
      for each vertex ∈ Vertices
        if edge.u == vertex.x
          edge.uData ← copy(vertex.xData)
```

Figure 3.3: Ideal functionality for a single iteration of the GAS model operations

they are shuffled using the same permutation as the data elements.)

Both the Gather and Scatter phases begin with calls to $\mathcal{F}_{\mathsf{shuffle}}$, which takes secret shares of the edge data from each party, and outputs fresh shares of the randomly permuted data. In practice we implement this using two sequential, generic secure computations of the Waksman permutation network [38], where each party randomly chooses one of the two permutations. Then, the parties iterate through the shuffled edge set, opening one side of each edge to reveal the neighboring vertex. Opening these vertices in the clear is where we leak information, and gain in efficiency. As we mentioned previously, this reveals a noisy histogram of the node degrees. In doing so, the parties can fetch the appropriate vertex from memory, without performing expensive oblivious sort operations, as in GraphSC,

π_{gas}

Secure Graph-Parallel Computation with Differentially Private Access Patterns

Inputs: Secret share of edges denoted as $\langle \mathsf{RealEdges} \rangle$, each edge is $\mathsf{edge} : (\mathsf{u}, \mathsf{v}, \mathsf{uData}, \mathsf{vData}, \mathsf{isReal})$. Secret share of vertices denoted as $\langle \mathsf{Vertices} \rangle$, each vertex contains $\mathsf{vertex} : (\mathsf{x}, \mathsf{xData})$. ($r$ stands for number of real items, and d for number of dummy ones)

Output: $\langle \mathsf{Edges} \rangle$, $\langle \mathsf{Vertices} \rangle$

Initialization:

$\langle \mathsf{DummyEdges} \rangle_{1:d} \leftarrow \mathsf{DumGen}_{p,\alpha}$
$\langle \mathsf{Edges} \rangle_{1:r} \leftarrow \langle \mathsf{RealEdges} \rangle_{1:r}$
$\langle \mathsf{Edges} \rangle_{r+1:r+d} \leftarrow \langle \mathsf{DummyEdges} \rangle_{1:d}$
$\langle \mathsf{Edges.isReal} \rangle_{1:r} \leftarrow \langle 1 \rangle$
$\langle \mathsf{Edges.isReal} \rangle_{r+1:r+d} \leftarrow \langle 0 \rangle$

Gather($\langle \mathsf{Edges} \rangle$)
  $\langle \mathsf{Edges} \rangle \leftarrow \mathcal{F}_{\mathsf{shuffle}}(\langle \mathsf{Edges} \rangle)$
  for each $\langle \mathsf{edge} \rangle \in \langle \mathsf{Edges} \rangle$
    $\mathsf{edge.v} \leftarrow \mathsf{Open}(\langle \mathsf{edge}.v \rangle)$
    for $\langle \mathsf{vertex} \rangle \in \langle \mathsf{Vertices} \rangle$
      if $\mathsf{edge.v} == \mathsf{vertex.x}$
        $\langle \mathsf{vertex.xData} \rangle \leftarrow \mathsf{copy}(\langle \mathsf{edge.uData} \rangle)$

Apply($\langle \mathsf{Vertices} \rangle$)
  for $\langle \mathsf{vertex} \rangle \in \langle \mathsf{Vertices} \rangle$
    $\langle \mathsf{vertex.xData} \rangle \leftarrow \mathcal{F}_{\mathsf{func}}(\langle \mathsf{vertex.xData} \rangle)$

Scatter($\langle \mathsf{Edges} \rangle$)
  $\langle \mathsf{Edges} \rangle \leftarrow \mathcal{F}_{\mathsf{shuffle}}(\langle \mathsf{Edges} \rangle)$
  for each $\langle \mathsf{edge} \rangle \in \langle \mathsf{Edges} \rangle$
    $\mathsf{edge.u} \leftarrow \mathsf{Open}(\langle \mathsf{edge}.u \rangle)$
    for $\langle \mathsf{vertex} \rangle \in \langle \mathsf{Vertices} \rangle$
      if $\mathsf{edge.u} == \mathsf{vertex.x}$
        $\langle \mathsf{edge.uData} \rangle \leftarrow \mathsf{copy}(\langle \mathsf{vertex.xData} \rangle)$

Figure 3.4: A protocol for two parties to compute a single iteration of the GAS model operation on secret-shared data. This protocol realizes the ideal functionality described in Figure 3.3.

and without resorting to ORAM. After fetching the appropriate node, the secret shared data is copied to/from the adjacent edge.

During Apply, the parties make a call to an ideal functionality, $\mathcal{F}_{\mathsf{func}}$. This functionality

takes secret shares of all vertices, reconstructs the data from the shares, applies the specified function to the real data at each vertex (while ignoring data from dummy edges), and returns fresh secret shares of the aggregated vertex data. In our implementation, we realize this ideal functionality using garbled circuits. We don't focus on the details here, as they have been described elsewhere (e.g. [10,11]).

$\mathsf{DumGen}_{p,\alpha}$ **in detail:** The ideal functionality for $\mathsf{DumGen}_{p,\alpha}$ appears in Figure 3.2. The role of $\mathsf{DumGen}_{p,\alpha}$ is to generate the dummy elements that create a "noisy" degree profile, $\widehat{\mathcal{D}}$. Starting with in-degree profile $\mathcal{D} = \mathsf{D}_{\mathsf{in}}(G)$, for each $i \in V$, $\widehat{\mathcal{D}}(i) = \mathcal{D}(i) + \gamma_i$, where each γ_i is drawn independently from a shifted geometric distribution, parameterized by a "stopping" probability p, and "shift" of α: we denote the distribution by $\mathbb{D}_{p,\alpha}$, and define it more precisely below. The shift ensures that negative values are negligible likely to occur. This is necessary because the noisy set determines our access pattern to memory, and we cannot accommodate a negative number of accesses (or, more accurately, we do not want to omit any accesses needed for the real data). More specifically, we will define below a "shift function" $\alpha : \mathbb{R} \times \mathbb{R} \to \mathbb{N}$ that maps every (ϵ, δ) pair to a natural number. (When ϵ and δ are fixed, we will simply use α to denote $\alpha(\epsilon, \delta)$.)

The functionality iterates through each vertex identifier $i \in V$, sampling a random number $\gamma_i \leftarrow \mathbb{D}_{p,\alpha}$, and creating γ_i edges of the form $(\perp, i)$. The remainder of the array contains "blank" edges, $(\perp, \perp)$, which can be tossed away as they are discovered later in the protocol, after the dummy edges have all been shuffled [1] $\mathsf{DumGen}_{p,\alpha}$ returns secret shares of the dummy edges, $\langle \mathsf{DummyEdges} \rangle$. The only difference between the functionality described in the middle column, and the one in the left portion of the figure (which was used in Section 3.1.1), is that our "types" are now node identifiers, and they are stored within edge structures. However, the reader should note that only the right node in each edge is assigned a dummy value, while the left nodes all remain $\perp$. This design choice is for efficiency, and

[1] Revealing these blank edges before shuffling would reveal how many dummy edges there are of the form $(*, i)$, which would break privacy. After all the edges are shuffled, revealing the number of blank edges only reveals the *total* number of dummy edges, which is fine.

comes at the cost of leaking the exact histogram defined by the out-degrees of the graph nodes when executing $\mathsf{Open}(\mathsf{Edges_i.u})$ in the Scatter operation. As an example of how this impacts privacy, when computing gradient descent for matrix factorization, this reveals the number of reviews written by each user, while ensuring that the number of reviews received by each item remains differentially private. This hides whether any given user reviewed any specific item, which suffices for achieving security with known input sizes, as defined in Definition 9. This is the protocol that we use in our implementation, but we briefly discuss what is needed to achieve Definition 7 below.

In some computations, the graph is known to be bipartite, with all edges starting in the left vertex set and ending in the right vertex set (again, recommendation systems are a natural example). In this case, since it is known that all nodes in the left vertex set have in-degree 0, we do not need to add dummy edges containing these nodes. This cuts down on the number of dummies required, and we take advantage of this when implementing matrix factorization.

Implementing $\mathsf{DumGen}_{p,\alpha}$: Intuitively, we sample γ_i by flipping a biased coin p until it comes up heads. We flip one more unbiased coin to determine the sign of the noise, and then add the result to α. We will determine p based on ϵ and δ. Formally, γ_i is sampled as follows:

$$\Pr[\gamma_i = \alpha] = \frac{p}{2}$$

$$\forall k \in \mathbb{N}, k \neq 0 : \Pr[\gamma_i = \alpha + k] = \frac{1}{2}(1 - \frac{p}{2})p(1-p)^{|k|-1}.$$

As just previously described, we view p as the stopping probability. However, in the first coin flip, we stop with probability $p/2$. We note that this is a slight modification to the normalized 2-sided geometric distribution, which would typically be written as $\Pr[\gamma_i = \alpha + k] = \frac{1}{2-p}p(1-p)^{|k|}$. The advantage of the distribution as it is written above is that it is very easy to sample in a garbled circuit, so long as p is an inverse power of 2; normalizing by $\frac{1}{2-p}$

introduces problems of finite precision and greatly complicates the sampling circuit. We note that Dwork et al. [43] suggest using the geometric distribution with $p = 2^{\ell}$, precisely because it is easy to sample in a garbled circuit. However, they describe a 1-sided geometric distribution, which is not immediately useful for preserving differential privacy, and did not seem to consider that, after normalizing, the 2-sided distribution cannot be sampled as cleanly.

We note that with some probability that is dependent on the choice of α, $\exists i \in V$ s.t. $\widehat{\mathcal{D}}(i) < 0$, which leaves us with a bad representation of a multiset. We therefore modify the definition of $\mathcal{F}$ to output $\emptyset$ whenever this occurs, and we always choose α so that this occurs with probability bound by δ. In our implementation, we set $\delta = 2^{-40}$.

To securely sample $\mathbb{D}_{p,\alpha}$, each party inputs a random string, and we let the XOR of these strings define the random tape for flipping the biased coins. If the first ℓ bits of the random tape are 1, the first coin is set to heads, and otherwise it set to tails: this is computed with a single ℓ-input AND gate. We iterate through the random tape, ℓ bits at a time, determining the value of each coin, and setting the dummy elements appropriately. We use one bit from the random tape to determine the sign of our coin flips, and we add α dummies to the result. Recall that the output length is fixed, regardless of this random tape, so after we set the appropriate number of dummy items based on our coin flips, the remaining output values are set to $\perp$.

The cost of this implementation of $\mathsf{DumGen}_{p,\alpha}$ is $O(V)$, though this hides a dependence on ϵ and δ: an exact accounting for various values can be found in Section 4.8. This cost is small relative to the cost of the oblivious shuffle, but we did first consider a much simpler protocol for $\mathsf{DumGen}_{p,\alpha}$ that is worth describing. Instead of performing a coin flip inside a secure computation, by choosing a different distribution, we can implement $\mathsf{DumGen}_{p,\alpha}$ without any interaction at all! To do this, we have each party choose d random values from $\{1, \ldots, |V|\}$, and view them as additive shares (modulo $|V|$) of each dummy item. Note

that this distribution is already one-sided, so we do not need to worry about α, and it already has fixed length output, so we do not need to worry about padding the dummy array with $\perp$ values. Intuitively, this can be viewed as $|V|$ correlated samples from the binomial distribution, where the bias of the coin is $1/|V|$. Unfortunately, the binomial distribution performs far worse than the geometric distribution, and in concrete terms, for the same values of ϵ and δ, this protocol resulted in 250X more dummy items. The savings from avoiding the secure computation of $\mathsf{DumGen}_{p,\alpha}$ were easily washed away by the cost of shuffling so many additional items.

3.4 Security Analysis

We begin by describing the leakage function $\mathcal{L}(G)$. Intuitively, we leak a noisy degree profile. As we mentioned previously, we analyze the simpler $\mathsf{DumGen}_{p,\alpha}$ algorithm, and prove that the mechanism provides differential privacy for graphs that have neighboring in-degree profiles. Then, we proceed afterwards to show that this leakage function suffices for simulating the protocol, achieving security in the joint-collection model, corresponding to Definition 9. (Extending the proof to meet Definition 7 is not much harder to do: we would use the $\mathsf{DumGen}_{p,\alpha}$ algorithm defined for the disjoint collection model, and prove that differential privacy holds for graphs that have neighboring full-degree profiles.)

We remind the reader that we use the following distribution, $\mathbb{D}_{p,\alpha}$ for sampling noise:

$$\Pr[\gamma_i = \alpha] = \frac{p}{2}$$

$$\forall k \in \mathbb{N}, k \neq 0 : \Pr[\gamma_i = \alpha + k] = \frac{1}{2}(1 - \frac{p}{2})p(1-p)^{|k|-1}.$$

We define a randomized algorithm, $\mathcal{F}_{\epsilon,\delta} : \ D \to \widehat{\mathcal{D}}$, whose input and output are multi-sets over V: $\forall i \in \{1, \ldots, |V|\}$, $\widehat{\mathcal{D}}(i) = \mathcal{D}(i) + \gamma_i$, where $\gamma_i \leftarrow \mathbb{D}_{p,\alpha}$.

Definition 8. *The leakage function is*

$\mathcal{L}(G) = (\mathcal{F}_{\epsilon,\delta}(\mathsf{D_{in}}(G)), \mathsf{D_{out}}G)$ *where* $\mathsf{D_{in}}(G)$ *denotes the in-degree profile of graph* G*, and* $\mathsf{D_{out}}(G)$ *denotes the out-degree profile.*

Theorem 1. *The randomized algorithm* $\mathcal{L}$ *is* (ϵ, δ)*-approximate differentially private, as defined in Definition 5.*

We note that $\mathsf{D_{out}}(G)$ can be modeled as auxiliary information about $\mathsf{D_{in}}(G)$, so the proof that $\mathcal{L}$ preserves differential privacy follows from the fact that the algorithm $\mathcal{F}_{\epsilon,\delta}$ is differentially private for graphs with neighboring in-degree profiles. It is well known that similar noise mechanisms preserve differential privacy, but, for completeness, we prove it below for our modified distribution, which is much simpler to execute in a garbled circuit.

Proof: To simplify notation, we use $\mathcal{F}$ to denote $\mathcal{F}_{\epsilon,\delta}$. Consider any two neighboring graphs, and let D_1, D_2 denote their neighboring in-degree profiles. Let $\mathcal{F}_R$ denote the range of $\mathcal{F}$, and let $\widehat{\mathcal{D}}$ be a multi-set in $\mathcal{F}_R$. We say that $\widehat{\mathcal{D}} \in \mathsf{Bad}$ if $\exists i \in \{1, \ldots, V\}, \widehat{\mathcal{D}}(i) < 0$, and assume for now that $\widehat{\mathcal{D}} \notin \mathsf{Bad}$. Let $\widehat{\mathcal{D}}_1 = \mathcal{F}(D_1)$, let $\widehat{\mathcal{D}}_2 = \mathcal{F}(D_2)$, and (without loss of generality) let i be the value for which $D_1(i) = D_2(i) + 1$. By the definition of $\mathcal{F}$, for $j \neq i$, $\Pr[\widehat{\mathcal{D}}_1(j) = \widehat{\mathcal{D}}(j)] = \Pr[\widehat{\mathcal{D}}_2(j) = \widehat{\mathcal{D}}(j)]$. Furthermore, for $k \neq j, k \neq i$, $b \in \{1, 2\}$, $\widehat{\mathcal{D}}_b(k)$ and $\widehat{\mathcal{D}}_b(j)$ are sampled independently. Therefore,

$$\frac{\Pr[\widehat{\mathcal{D}}_1 = \widehat{\mathcal{D}}]}{\Pr[\widehat{\mathcal{D}}_2 = \widehat{\mathcal{D}}]} = \frac{\Pr[\widehat{\mathcal{D}}_1(i) = \widehat{\mathcal{D}}(i)]}{\Pr[\widehat{\mathcal{D}}_2(i) = \widehat{\mathcal{D}}(i)]} \leq \frac{1}{(1-p)}$$

(Note that the case $|\widehat{\mathcal{D}}(i)| = |\widehat{\mathcal{D}}_1(i)|$ – i.e. where there is no noise of type i added to the first dataset – $\frac{\Pr[\widehat{\mathcal{D}}_1 = \widehat{\mathcal{D}}]}{\Pr[\widehat{\mathcal{D}}_2 = \widehat{\mathcal{D}}]} \leq \frac{1}{1-p/2} < \frac{1}{1-p}$.) By choosing $1 - p = e^{-\epsilon}$, we achieve the desired

bound. Then, for any $T_{\mathsf{g}} \subseteq \mathcal{F}_R \setminus \mathsf{Bad}$,

$$\begin{aligned}
\Pr[\mathcal{F}(D_1) \in T_{\mathsf{g}}] &= \sum_{D \in T_{\mathsf{g}}} \Pr[\mathcal{F}(D_1) = D] \\
&\leq \sum_{D \in T_{\mathsf{g}}} e^{\epsilon} \Pr[\mathcal{F}(D_2) = D] \\
&= e^{\epsilon} \Pr[\mathcal{F}(D_2) \in T_{\mathsf{g}}]
\end{aligned}$$

We now consider the probability that $\mathcal{F}(D) \in \mathsf{Bad}$. Recall, this is exactly the probability that for some $i \in V$, $\gamma_i < 0$, which grows as a negligible function in α. We choose α such that this probability is δ. (We will derive the exact function below, and demonstrate some sample parameters.) Then, for any $T \subseteq \mathcal{F}_R$, letting $T_{\mathsf{g}} = T \setminus \mathsf{Bad}$,

$$\begin{aligned}
\Pr[\mathcal{F}(D_1) \in T] &= \Pr[\mathcal{F}(D_1) \in T_{\mathsf{g}}] + \Pr[\mathcal{F}(D_1) \in \mathsf{Bad}] \\
&\leq e^{\epsilon} \Pr[\mathcal{F}(D_2) \in T_{\mathsf{g}}] + \delta \\
&\leq e^{\epsilon} \Pr[\mathcal{F}(D_2) \in T] + \delta
\end{aligned}$$

■

Setting the parameters Note that the sensitivity of the distance metric defined in Definition 2 is 1. Although our proof here is for neighboring graphs, we can use standard composition theorems to claim differential privacy for graphs of distance d, at the cost of scaling ϵ by a factor of d. We also note that $e^{\epsilon} = 1/(1-p)$, where p is the stopping probability defined in our noise distribution.

We set $\delta = 2^{-40}$, and show how to calculate α; this allows us to give the expected size of $\widehat{\mathcal{D}}$ as a function of ϵ and δ. We first fix some $i \in V$ and calculate $\Pr[\gamma_i < 0]$, and then we

take a union bound over $|V|$.

$$\begin{aligned}
\Pr[\gamma_i < 0] &= \sum_{k=\alpha+1}^{\infty} \frac{1}{2}(1-\frac{p}{2})p(1-p)^{k-1} \\
&= \frac{p}{2}(1-\frac{p}{2})\sum_{k=0}^{\infty}(1-p)^{\alpha}(1-p)^{k} \\
&= \frac{p}{2}(1-\frac{p}{2})(1-p)^{\alpha}\frac{1}{1-(1-p)} \\
&= \frac{1}{2}(1-\frac{p}{2})(1-p)^{\alpha}
\end{aligned}$$

After taking a union bound over $|V|$, we have $\Pr[\mathcal{F}(D) \in \mathsf{Bad}] \leq 2^{-40}$ when $\alpha > \frac{-40-\log(\frac{1}{2}-\frac{p}{4})-\log(|V|)}{\log(1-p)}$. Recall that $(1-p) = e^{-\epsilon}$. So, as an example, setting $\epsilon = .3$ and $|V| = 2^{12}$, we have $\alpha = 118$, and $\mathbb{E}(|\mathcal{F}(D)|) = 118|V| + |D|$. That is, for these privacy parameters, we expect to add 118 dummy edges for each node in the graph.

Theorem 2. *The protocol π_{gas} defined in Figure 3.4 securely computes $\mathcal{F}_{\mathsf{gas}}$ with $\mathcal{L}$ leakage in the $(\mathcal{F}_{\mathsf{func}}, \mathcal{F}_{\mathsf{shuffle}}, \mathsf{DumGen}_{p,\alpha})$-hybrid model according to Definition 9 (respectively Definition 7) when using the second (resp. third) variant of $\mathsf{DumGen}_{p,\alpha}$.*

Proof: (sketch.) We only prove the first Theorem statement, and omit the proof that we can meet the stronger security definition. At the end of this section, we give some intuition for what would change in such a proof.

Recall that the leakage functionality contains $(\mathcal{F}(\mathcal{DB}_R), \mathsf{out\text{-}deg}(V))$. In particular, then, we assume that $\mathsf{out\text{-}deg}(V)$ is public knowledge and given to the simulator, which holds in the joint collection model of Definition 9. Note that $|V|$ and $|E|$ are both determined by $\mathsf{out\text{-}deg}(V)$, and these values will be used by the simulator as well.

We construct a simulator for a semi-honest P_1. For all three ideal functionalities, the

output is simply an XOR secret sharing of some computed value. The output of all calls to these functionalities can be perfectly simulated using random binary strings of the appropriate length. Let $\mathsf{simEdges}_1$ denote the random string used to simulate the output of $\mathcal{F}_{\mathsf{shuffle}}$ the first time the functionality is called, and let $\mathsf{simEdges}_2$ denote the random string used to simulate the output on the second call. Let $\mathsf{simEdges}_1.u$ denote the restriction of $\mathsf{simEdges}_1$ to the bits that make up the sharings of $\mathsf{Edges}.u$, and let $\mathsf{simEdges}_2.v$ be defined similarly.

There are only two remaining messages to simulate:
$\mathsf{Open}(\mathsf{edge.u})$, and $\mathsf{Open}(\mathsf{edge.v})$. Recall that there are $|E| + 2\alpha|V|$ edges in the Edges array: the original $|E|$ real edges, and the $2\alpha|V|$ dummy edges generated in $\mathsf{DumGen}_{p,\alpha}$. To simulate the message sent when opening $\mathsf{Edges}.u$, the simulator uses the values $|V|$ and $\mathsf{out\text{-}deg}(V)$ to create a bit string representing a random shuffling of the following array of size $|E| + 2\alpha|V|$. For each $u \in V$, the array contains the identifier of u exactly $\mathsf{out\text{-}deg}(u)$ times. This accounts for $|E| = \sum_u \mathsf{out\text{-}deg}(u)$ positions of the array; the remaining $2\alpha|V|$ positions are set to $\perp$, consistent with the left nodes output by $\mathsf{DumGen}_{p,\alpha}$. Letting r denote the resulting bit-string, the simulator sends $r \oplus \mathsf{simEdges}_1.u$ to the adversary.

To simulate $\mathsf{simEdges}_2.v$, the simulator creates another bit-string representing a random shuffling of the following array, again of size $|E| + 2\alpha|V|$. Letting $\widehat{\mathcal{D}} = \mathcal{F}(\mathcal{DB}_R)$ denote the first element output by the leakage $\mathcal{L}$, the simulator adds the node identifiers in $\widehat{\mathcal{D}}$ to the array. In the remaining $|E| + 2\alpha|V| - |\widehat{\mathcal{D}}|$ positions of the array, he adds $\perp$. Letting r denote the resulting bit-string, the simulator sends $r \oplus \mathsf{simEdges}_2.v$ to the adversary.

So far, this results in a perfect simulation of the adversary's view. However, note that the outputs of the two parties should be correlated. To ensure that the joint distribution over the adversary's view and the honest party's output is correct, the simulator has to submit the adversary's input, $\langle \mathsf{Vertices} \rangle$, to the trusted party. He receives back a new sharing of Vertices, and has to "plant" this value in his simulation. Specifically, in the final iteration of the protocol, when simulating the output of $\mathcal{F}_{\mathsf{func}}$ for the last time, the simulator uses

$\langle$Vertices$\rangle$, as received from the trusted party, as the simulated output of this function call.

■

Hiding the out-degree of each node. We include another variant of $\mathsf{DumGen}_{p,\alpha}$ on the right side of Figure 3.2. In that variant, separate noise is added to the left node of each edge as well as to the right, which provides security according to Definition 7. We do not implement or analyze the security of this variant. Intuitively, though, for a graph $G = (E, V)$, it is helpful to think of the edge set as defining two databases of elements over V: for each (directed) edge (u, v), we will view u as an element in database E_L and v as an element in database E_R. Because the oblivious shuffle hides the edges between these two databases, the access pattern can be fully simulated from two noisy histograms (one for each database). This doubles the "sensitivity" of the "query", and, because differential privacy composes, the added noisy information has the affect of cutting ϵ in half. Since our analysis includes multiple values of ϵ, the reader can easily extrapolate to get a sense of how we perform under our stronger security notion.

Hiding a user's full edge set. The leakage function described above provide edge privacy to each contributing party. That is, we have defined two databases to be neighboring when they differ in a single edge. To understand the distinction, consider the application of building a movie recommendation system through matrix factorization. If we guarantee edge privacy, then nobody can learn whether a particular user reviewed a particular movie, but we cannot rule out the possibility that an adversary could learn something about the *set* of movies they have reviewed, perhaps, say, the genre that they enjoy. We could also define two neighboring databases as differing in a single node. Using the same example, this would guarantee that nothing can be learned about any individual user's reviews, at all. It would require more noise: if the maximum degree of any node is d, ensuring node privacy would have the affect of scaling ϵ by d. In our experiments, we have included some smaller values of ϵ to help the reader evaluate how this additional noise would impact performance. However, we note that if the maximum degree in the graph is large, achieving node privacy

might be difficult. We defer investigating other possible notions of neighboring graphs to future work.

Sequential composition. The standard security definition for secure computation composes sequentially, allowing the servers to perform repeated computations on the same data without impacting security. With our relaxation, if we later use the same user data in a new computation, the leakage does compound. The standard composition theorems from the literature on differential privacy do apply, and we do not address here how privacy ought to be budgeted across multiple computations. The reader should note that in our iterative protocol, there is no additional leakage beyond the first iteration, because we do not regenerate the dummy items: the leakage in each iteration is the exactly the same noisy degree profile that was leaked in all prior iterations.

3.5 Differentially Private Graph Computation with $O(|E|)$ complexity

The construction in Section 3.3.2 requires $O((|E|+\alpha|V|)\log(|E|+\alpha|V|))$ garbled AND gates. In comparison, the implementation of Nayak et al. [11] uses $O(|E|+|V|)\log^2(|E|+|V|)$ garbled gates. As we found in the previous section, $\alpha = O(\frac{\log\delta - \log|V|}{\epsilon})$. When $|E| = O(\alpha|V|)$, this amounts to an asymptotic improvement of $O(\log(|E|))$. This improvement stems from our ability to replace several oblivious sorting circuits with oblivious shuffle circuits, which we are able to do only because of our security relaxation. However, while less practical, Nayak et al. could instead rely on an asymptotically better algorithm for oblivious sort, reducing their runtime to $O((|E|+|V|)\log(|E|+|V|))$. We therefore find it interesting to ask whether our security relaxation admits *asymptotic* improvement for this class of computations, in addition to the practical improvements described in the previous section. Indeed, we show that we can remove the need for an oblivious shuffle altogether by allowing one party to shuffle the data locally. As long as the party that knows the shuffling permutation

does not see the access pattern to V during the Scatter and Gather phases, the protocol remains secure. The reason this protocol is less practical then the protocol of Section 3.3.2 is because $\mathcal{F}_{\mathsf{func}}$ now has to perform decryption and encryption, which would require large garbled circuits.

The construction we present here requires $O(|E| + \alpha|V|)$ garbled AND gates, demonstrating asymptotic improvement over the best known construction for this class of computations, whenever $|E| = O(\alpha|V|)$.

We assume that the two computation servers hold key pairs, $(\mathsf{sk}_{\mathsf{Alice}}, \mathsf{pk}_{\mathsf{Alice}})$ and $(\mathsf{sk}_{\mathsf{Bob}}, \mathsf{pk}_{\mathsf{Bob}})$. When data owners upload their data, they encrypt the data under Alice's key, encrypt the resulting ciphertext under Bob's key, and send the result to Bob (obviously this second encryption is unnecessary, but it simplifies the exposition to assume Bob receives the input in this form).[2] Recall that edge data contains $(\mathsf{u}, \mathsf{v}, \mathsf{uData}, \mathsf{vData}, \mathsf{isReal})$, and vertex data contains $(\mathsf{x}, \mathsf{xData})$. We assume each of these elements are encrypted independently, so that we can decrypt portions of edges when needed. We also assume that these encryption schemes are publicly re- randomizable: anyone can take an encryption of x under pk, and re-randomize the ciphertext to give an encryption of x, with fresh randomness, under the same pk. We assume that re-randomized ciphertexts and "fresh" ciphertexts are equivalently distributed. Throughout this protocol, we use $[\![x]\!]_y$ to denote the encryption of x using y's public key.

The protocol follows the same outline as the one in Section 3.3.2, but here we separate the tasks of shuffling and data copying. Bob locally shuffles the edges, $[\![[\![\mathsf{Edges}]\!]_{\mathsf{Alice}}]\!]_{\mathsf{Bob}}$ according to a permutation of his choice. He sends the encrypted, shuffled arrays to Alice. For each edge, he also partially decrypts the node identifier for the right node, recovering $[\![\mathsf{Edges.v}]\!]_{\mathsf{Alice}}$. He re-randomizes the resulting ciphertext, and sends it to Alice. Alice can

[2] The data could instead be uploaded as in the previous section, and the servers could perform a linear scan on the data to encrypt it as described here. This wouldn't impact the asymptotic claim; we chose the simpler presentation.

now find the right vertex of every edge. She executes the Gather operation locally by performing a linear scan over the edge data, opening the right vertex of edge, and copying data from edge to vertex.

The two parties then execute the Apply operation together, performing a linear scan over the vertices, and calling a two-party functionality at vertex.[3] Alice supplies the functionality, $\mathcal{F}_{\mathsf{func}}$, with the encrypted data at each vertex, and both parties provide their decryption key. The functionality decrypts, performs the Apply function to all real data, and re-encrypts. The updated, encrypted vertex data is output to Alice.

Bob now reshuffles all the edges and dummy flags, just as before, re-randomizing Alice's ciphertexts. He sends $[\![[\![\mathsf{Edges}]\!]_{\mathsf{Alice}}]\!]_{\mathsf{Bob}}$ to Alice, who now performs the Scatter operation, as with Gather. That is, for each edge, she receives the re-randomized encryption of the left vertex id, $[\![\mathsf{Edges.u}]\!]_{\mathsf{Alice}}$, recovers the vertex identifier, and copies the vertex data from u back to the appropriate edge. She re-randomizes all ciphertexts, and sends the edge data back to Bob.

The proof of security is not substantially different than in the previous section, so we only give an intuition here. Instead of using random strings to simulate secret shares, we now rely on the semantic security of the encryption scheme. When simulating Alice's view, for each $u \in \mathsf{Vertices}$, the leakage function is used to determine how many times the identifier for u should be encrypted. The rest of the ciphertexts can be simulated with encryptions of 0 strings. The rest of the simulation is straightforward.

When simulating Bob's view, an interesting subtlety arises. Even though Bob does not get to see the access pattern to the vertices during the Gather and Scatter operations, he does in fact still learn $\mathcal{F}(\mathcal{DB}_R)$. This is because the instantiation of $\mathcal{F}_{\mathsf{func}}$ with a secure computation will leak the input size of Alice (assuming we use a generic two-party computation for realizing the functionality). This reveals the number of data items that were moved to that vertex during Gather.[4] These input sizes can be exactly simulated using the leakage

[3] As before, we can replace this functionality with a two-party computation.

[4] If Bob knew how many dummy edges have the form $(*, v)$, he could immediately deduce $\mathsf{in\text{-}deg}(v)$; this

function.

3.6 Implementation and Evaluation

In this section, we describe and evaluate the implementation of our proposed framework. We implement OblivGraph using FlexSC, a Java-based garbled circuit framework. We measure the performance of our framework on a set of benchmark algorithms in order to evaluate our design. These benchmarks consist of histogram, PageRank and matrix factorization problems which are commonly used for evaluating highly-parallelizable frameworks. In all scenarios, we assume that the data is secret-shared across two non-colluding cloud providers, as motivated in Section 1. For comparison, we compare our results with the closest large-scale secure parallel graph computation, called GraphSC [11].

3.6.1 Implementation

Using the OblivGraph framework, the histogram and matrix factorization problems can be represented as directed bipartite graphs, and PageRank as a directed non-bipartite graph. When we are computing on bipartite graphs, if we consider Definition 9 where we aim to hide the in-degree of the nodes (nodes on the left have in-degree 0), the growth rate of dummy edges is linear in the number of nodes on the right and it is independent of the real edges or users. If we consider the stronger Definition 7, the growth rate of dummy edges is linear with max(users, items).

Histogram: In histogram, left vertices represent data elements, right vertices are the counters for each type of data element, and existence of an edge indicates that the data element on the left has the type on the right.

Matrix Factorization: In matrix factorization, left vertices represent the users, right vertices are items (e.g. movies in movie recommendation systems), an edge indicates that a user ranked an item, and the weight of the edge represents the rating value.

is why $\mathsf{DumGen}_{p,\alpha}$ is still executed by an ideal functionality, and not entrusted to Bob.

PageRank: In PageRank, each vertex corresponds to a webpage and each edge is a link between two webpages. The vertex data comprises of two real values, one for the PageRank of the vertex and the other for the number of its outgoing edges. Edge data is a real value corresponding to the weighted contribution of the source vertex to the PageRank of the sink vertex.

Vertex and Edge representation: In all scenarios, vertices are identified using 16-bit integers and 1 bit is used to indicate if the edge is real or dummy. For Histograms, we use an additional 20 bits to represent the counter values. In PageRank, we represent the PageRank value using a 40-bit fixed-point representation, with 20-bits for the fractional part. In our matrix factorization experiments, we factorized the matrix to user and movie feature vectors; each vector has dimension 10, and each value is represented as 40-bit fixed-point number, with 20-bits for the fractional part. We chose these values to be consistent with GraphSC representation.

System setting: We conduct experiments on both a lab testbed, and on a real-world scale Amazon AWS deployment. Our lab testbed comprises 8 virtual machines each with dedicated (reserved) hardware of 4 CPU cores (2.4 GHz) and 16 GB RAM. These VMs were deployed on a vSphere Cluster of 3 physical servers and they were interconnected with 1Gbps virtual interfaces. We run our experiments on $p \in \{1, 2, 4, 8, 16, 32\}$ pairs of these processors, where in each pair, one processor works as the garbler, and the other as the evaluator. Each processor can be implemented by a core in a multi-core VM, or can be a VM in our compute cluster.

3.6.2 Evaluation

We use two metrics in evaluating the impact of our security relaxation: circuit complexity (e.g. # of AND gates), and runtime. Counting AND gates provides a "normalized" comparison with other frameworks, since circuit size is independent of the hardware configuration and of the chosen secure computation implementation. However, it is also nice

to have a sense of concrete runtime, so we provide this evaluation as well. Of course, runtime is highly affected by the choice of hardware, and ours can be improved by using more processors or dedicated hardware (e.g. AES-NI).

Evaluation setting: For the LAN setup, we use synthesized data and run all the benchmarks with the similar set of parameters that have been used in the GraphSC framework. In our histogram and matrix factorization experiments, we run the experiments for 2048 users and 128 items. The number of nodes in our PageRank experiment is set to be 2048.

For real world experiment using AWS, we run matrix factorization using gradient descent on the real-world MovieLens dataset that contains 1 million ratings provided by 6040 users to 3883 movies [44] on 2 m4.16xlarge AWS instances on the Northern Virginia Datacenter.

Circuit Complexity: The results presented in Figures 3.5, 3.6 and 3.7 are for execution on a single processor, to show the performance of our design without leveraging the desired effect of parallelization.

Histogram: Figure 3.5 demonstrates the number of AND gates for computing histogram in both the GraphSC and OblivGraph frameworks. With 2048 data elements and 128 data types, we always do better than GraphSC when $\epsilon >= 0.3$. When $\epsilon = 0.1$, we start outperforming GraphSC when there are at least 3400 edges.

Matrix Factorization: In Figure 3.6, we use the (batch) gradient decent method for generating the recommendation model, as in [10, 11]. With 2048 users, 128 items, and $\epsilon = 0.3$, we outperform GraphSC once there are at least 15000 edges. When $\epsilon = 0.1$, we start outperforming them on 54000 edges. We always do better than GraphSC when the $\epsilon = 1$ or higher.

PageRank: Figure 3.7 provides the result of running PageRank in our framework with 2048 nodes and different values of ϵ. With $\epsilon = 0.3$, we outperform GraphSC when the number of edges are about 400000, and with $\epsilon = 1$ we outperform them on just 130000 edges. In both cases, the graph is quite sparse, compared to a complete graph of 2 million edges.

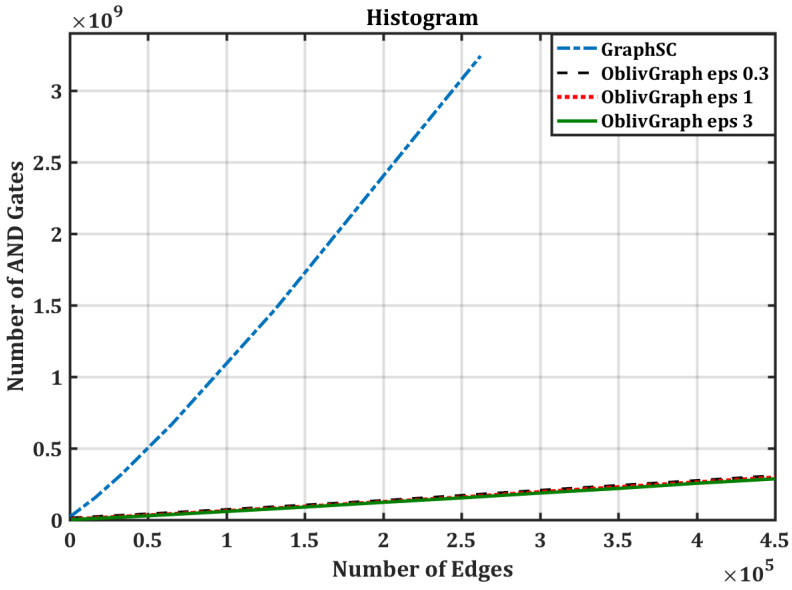


Figure 3.5: Histogram with 2048 users, 128 counters, and varying ϵ

Note, though, that our comparison is slightly less favorable for this computation. Recall, the number of dummy edges grow with the number of nodes in the graph, and, when hiding only in-degree in a bipartite graph, this amounts to growing only with the number of nodes on the right. In contrast, the runtime of GraphSC grows equivalently with any increase in users, items, or edges, because their protocol hides any distinction between these data types. We therefore compare best with them when there are more users than items. When looking at a non-bipartite graph, such as PageRank, our protocol grows with any increase in the size of the singular set of nodes, just as theirs does. If we increase the number of items in matrix factorization to 2048, or decrease the number of nodes in PageRank to 128, the comparison to GraphSC in the resulting experiments would look similar. We let the reader extrapolate, and avoid the redundancy of adding such experiments.

Large scale experiments on Amazon AWS: OblivGraph factorizes the MovieLens recommendation matrix consist of 1 million ratings provided by 6040 users to 3883 movies, in almost 2 hours while GraphSC does it in 13 hours. We provide results of computing matrix factorization problem for different values of ϵ and different numbers of ratings in Table 3.2.

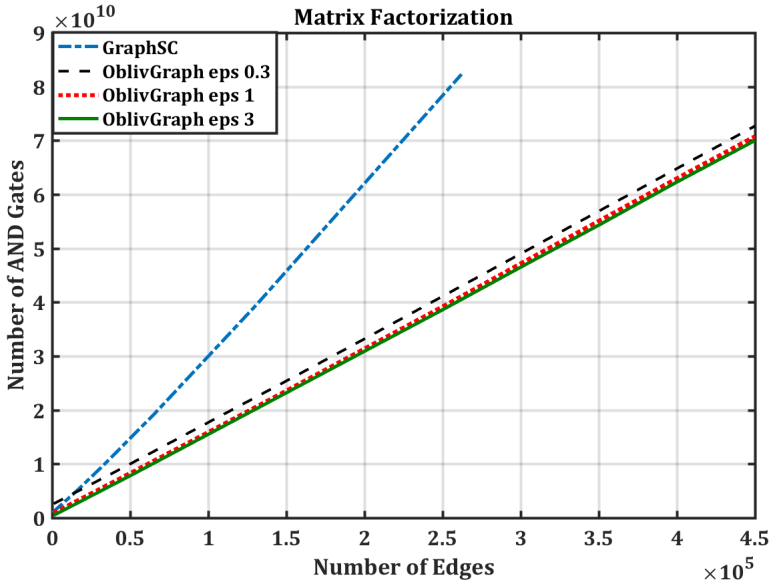


Figure 3.6: Matrix Factorization with 2048 users, 128 movies, and varying ϵ

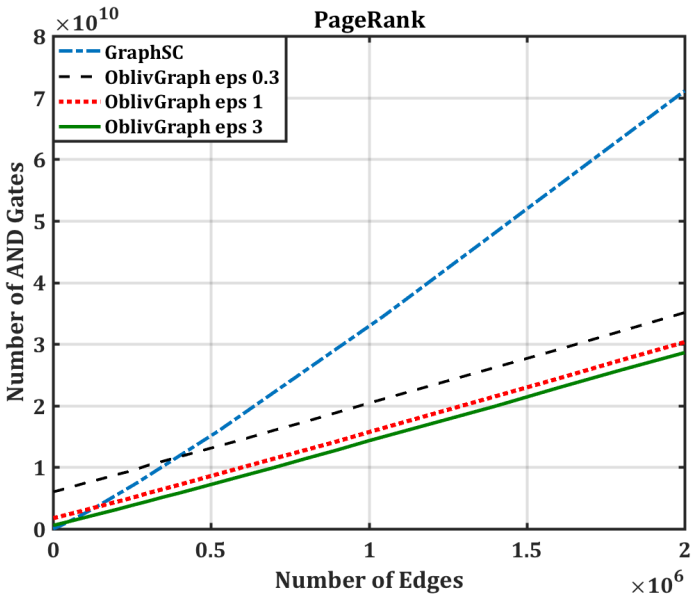


Figure 3.7: PageRank with 2048 webpages, and varying ϵ

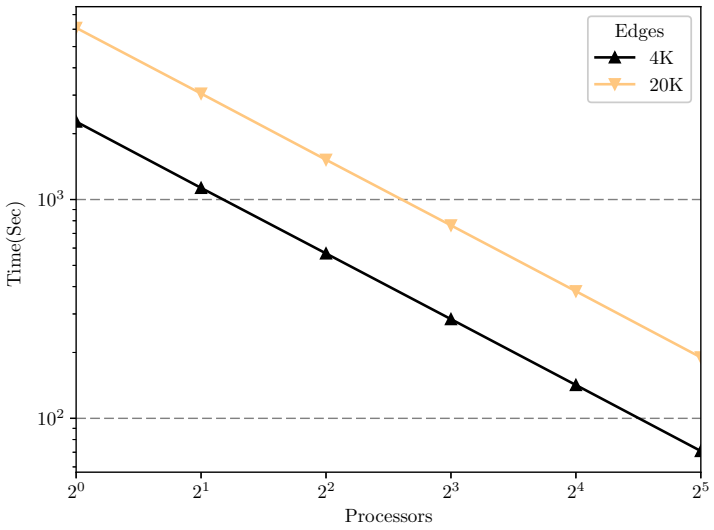


Figure 3.8: Effect of parallelization on Matrix Factorization computation time

We outperform the best result achieved by GraphSC, using 128 processors and 1M ratings.

Effect of Parallelization: Figure 3.8 illustrates that the execution time can be significantly reduced through parallelization. We achieve nearly a linear speedup in the computation time. The lines corresponds to two different numbers of edges for 2048 users and 128 movies. Since in our these problems, the computation is the bottleneck, parallelization can significantly speed up the computation process. Table 3.1 shows the effect of parallelization in our framework as compared to GraphSC in terms of number of AND gates. As shown in the Table 3.1, adding more processors in the GraphSC framework increases the total number of AND Gates by some small amount. In contrast, the size of the circuit generated in our framework is constant in the number of processors: parallelization does not affect total number of AND gates in the OblivGraph GAS operations, or in DumGen.

Optimization using Compaction: It is important to note that the measured circuit sizes in our OblivGraph experiments correspond to the worst-case scenario in which the number of dummy edges are equal to $d = 2\alpha|V|$, which is the maximum number of dummies per

Table 3.1: Cost of Parallelization on OblivGraph vs. GraphSC in computing Matrix Factorization

Processors	**GraphSC [11]**		**OblivGraph**	
	$\|E\| = 8192$	$\|E\| = 24576$	$\|E\| = 8192$	$\|E\| = 24576$
1	$4.047E+09$	$1.035E+10$	$2.018E+09$	$4.480E+09$
2	$4.055E+09$	$1.039E+10$	$2.018E+09$	$4.480E+09$
4	$4.070E+09$	$1.046E+10$	$2.018E+09$	$4.480E+09$
8	$4.092E+09$	$1.057E+10$	$2.018E+09$	$4.480E+09$

type. Consequently the time for OblivShuffle is its maximum value. However, looking at the geometric distribution used in the DumGen procedure, the expected number of dummy edges is $\alpha|V|$, so half of the dummy items are unnecessary. Removing these extra dummy items during DumGen is non-trivial, because, while it is safe to reveal the total number of dummy items in the system, revealing the number of dummy items of each type would violate differential privacy. After the first iteration of the computation, once the dummy items are shuffled in with the real items, an extra flag marking the excessive dummy items can be used to safely remove them from the system; this optimization can significantly reduce the shuffling time (roughly by half) in the following iterations. However, our graphs are showing only the first iteration of the algorithm and they do not reflect this simple optimization.

Comparison with a Cleartext Baseline GraphSC [11] compared their execution time with GraphLab [8], a state-of-the-art framework for running graph-parallel algorithms on clear text. They ran Matrix Factorization using gradient descent with input length of 32K in both frameworks and demonstrated that GraphSC is about 200K - 500K times slower than GraphLab when run on 2 to 16 processors. Considering our improvements over GraphSC, we estimate our secure computation to be about 16K-32K times slower than insecure baseline computation (GraphLab), running the same experiments.

Oblivious Shuffle: We use an Oblivious Shuffle in our OblivGraph framework, which has

a factor of $\log(n)$ less overhead than the Bitonic sort used in GraphSC. We designed the Oblivious Shuffle operation based on the Waksman network [45]. The cost of shuffling is approximately $BW(n)$ using a Waksman network, where $W(n) = n \log n - n + 1$ is the number of oblivious swaps required to permute n input elements, and B indicates the size of the elements being shuffled. In the original Waksman switching network, the size of the input, n, is assumed to be a power of two. However, in order to have an Oblivious Shuffle for arbitrary sized input, we must use an improved version of the Waksman network proposed in [38] which is called AS-Waksman (Arbitrary-Sized Waksman). In our current set of experiments, we have only implemented the original version of the Waksman network and have not implemented AS-Waksman. We interpolate precisely to determine the size of arbitrary AS-Waksman when using arbitrary sized input.

Cost of each operation in OblivGraph framework: In order to understand how expensive the DumGen and OblivShuffle procedures are, as compared to other GAS model operations, we show the number of AND gates for each of these procedures in Figure 3.9. The figure corresponds to Matrix Factorization problem, with 2048 users, 128 movies and 20K ratings, with epsilon 0.5. The cost of a single iteration in the OblivGraph framework is first dominated by the Apply operation which computes the gradient descent and second by Oblivious Shuffle. Figure shows the effect of parallelization on decreasing the circuit size of each operation. See the full version to compare the cost of DumGen procedure in different protocols.

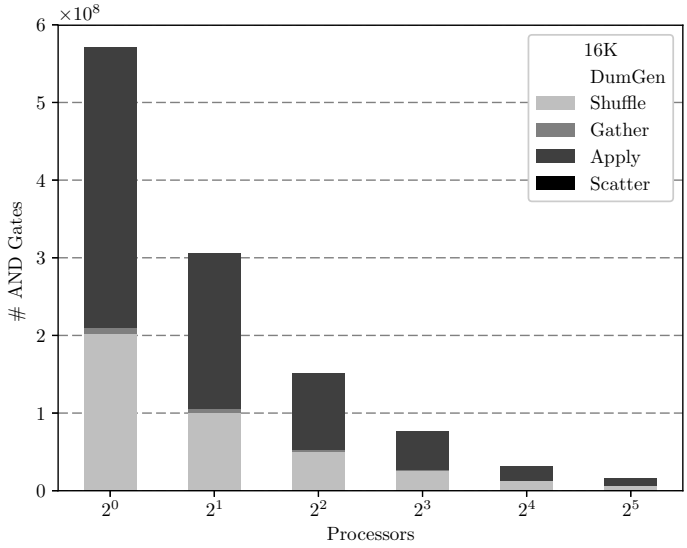


Figure 3.9: Cost of each operation in OblivGraph for Matrix Factorization

Table 3.2: Runtime of a single iteration of OblivGraph vs. GraphSC to solve matrix factorization problem in scale, with real-world dataset, MovieLens with 6040 users ranked 3883 movies

	OblivGraph			**GraphSC[11]**
	ϵ=0.3	**ϵ=0.5**	**ϵ=1**	
# Real Edges	$1.2M$	$1.5M$	$1.8M$	$1M$
Time(hours)	2.2	2.3	2.4	13

Chapter 4: Privacy-Preserving Parallel Machine Learning Computation in Malicious Settings [46]

In this chapter, we revisit secure computation of graph parallel algorithms, simultaneously leveraging all three of the advances just described: we assume four computation servers (with an honest majority, and one malicious corruption), allow differentially private leakage during computation, and, exploiting the parallelism that this affords, we construct an MPC protocol that can perform at *national scales*. Concretely, we compute histograms on 300 million inputs in 4.18 minutes, and we perform sparse matrix factorization, which is used in recommendation systems, on 20 million inputs in under 6 minutes. These problems have broad, real-world applications, and, at this scale, we could imagine supporting the Census Bureau, or a large company such as Amazon. For comparison, the largest experiments in GraphSC [11] and OblivGraph [47] had 1M inputs, and required 13 hours and 2 hours of runtime, respectively, while using 4 times the number of processors that we employ. End-to-end, our construction is $320X$ faster than OblivGraph.

Technical contributions. Merging the four-party protocol of Gordon et al. [48] with the construction of Mazloom and Gordon [47] raises several challenges and opportunities.

4.1 Background

4.1.1 MPC with differentially private leakage

The security definition for secure computation is built around the notion of protocol simulation in an *ideal world* execution [49]. In the ideal world, a trusted functionality takes the inputs, performs the agreed upon computation, and returns the result. We say the protocol

is secure if a simulator can simulate the adversary's protocol view in this ideal world, drawing from a distribution that is indistinguishable from the adversary's view in the real world execution. The simulator can interact with the adversary, but is otherwise given nothing but the output computed by the ideal functionality.[1]

In prior work, Mazloom and Gordon [47] proposed a relaxation to this definition in which the simulator is additionally given the output of some leakage function, $\mathcal{L}$, applied to all inputs, but $\mathcal{L}$ is proven to preserve differential privacy of the input. They define several varying security models. Here we focus on one variant, which supports more efficient protocol design. We assume that thousands of clients have secret shared their inputs with 4 computation servers, and we use E to denote the full set of inputs. We denote the set of secret shares received by server i as E_i. We denote the input of party j as e_j. Note that the servers learn the input size of each client. Formally, the security definition is as follows.

Definition 9. *[47] Let $\mathcal{F}$ be some functionality, and let π be an interactive protocol for computing $\mathcal{F}$, while making calls to an ideal functionality $\mathcal{G}$. π is said to* securely compute $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with $\mathcal{L}$ leakage, known input sizes, and $(\kappa, \epsilon, \delta)$-security *if $\mathcal{L}$ is (ϵ, δ)-differentially private, and, for every PPT, malicious, non-uniform adversary $\mathcal{A}$ corrupting a party in the $\mathcal{G}$-hybrid model, there exists a PPT, non-uniform adversary $\mathcal{S}$ corrupting the same party in the ideal model, such that, on any valid input shares, E_1, E_2, E_3, E_4*

$$\left\{\textsc{hybrid}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}\left(E_1, E_2, E_3, E_4, \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv}$$

$$\left\{\textsc{ideal}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\forall j:|e_j|)}(E_1, E_2, E_3, E_4, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \tag{4.1}$$

Mazloom and Gordon construct a protocol for securely performing graph-parallel computations with differentially private leakage. In their protocol, the data is secret shared

[1] This brushes over some of the important technical details, but we refer the reader to a formal treatment of security in Goldreich's book [49].

throughout each iteration: when the Apply phase is executed at each graph node, it is computed securely on secret shared data, with both input and output in the form of secret shares. The leakage is purely in the form of access patterns to memory: as data moves from edge to neighboring node and back again, during the Gather and Scatter phases, the protocol allows some information to leak about the structure of the graph. To minimize and bound this leakage, two additional actions are taken: 1) The edges are obliviously shuffled in between when the data is gathered at the left vertex, and when it is gathered at the right vertex. This breaks the connections between the left and right neighboring nodes, and reduces the graph structure leakage to a simple degree count of each node. 2) "Dummy" edges are created at the beginning of the protocol, and shuffled in with the real edges. These dummy edges ensure that the degree counts are *noisy*. When the dummy edges are sampled from an appropriate distribution, the leakage can be shown to preserve differential privacy. Note that when the input size of each party is known, the degree count of certain nodes may not need to be hidden, allowing for better performance. For example, if the data elements owned by user u are weighted edges of the form $(u, v, data)$, it is essential that the degree of node v remain private, as its degree leaks the edge structure of the graph, but the degree of node u is implied by the input size of user u. The implications of this are discussed more fully in their work.

4.1.2 Securely outsourcing computation.

These advances have introduced an opportunity for several applications of secure computation in which user data from thousands of parties are secret shared among a few servers (usually three) to perform a secure computation on their behalf. Multiple variants of this application have now been deployed. In some cases, users have already entrusted their data, in the clear, to a single entity, which then wishes to safeguard against data breach; secret sharing the data among several servers, each with a unique software stack, helps diversify the risk of exposure. In other cases, users were unwilling, or were even forbidden by

law, to entrust their data to any single entity, and the use of secure computation was essential to gaining their participation in the computation. In many of these cases, the servers executing the secure computation are owned and operated by a single entity that is trusted for the time being, but may be corrupted by an outside party. In other cases, some data were entrusted to one entity, while other data, from another set of users, were entrusted to a second entity, and these two distrusting parties wish to join in a shared computation.

The common denominator in all of these variants is that the computation servers are distinct from the data owners. In this context, the relaxation allowing these servers to learn some small, statistical information about the data may be quite reasonable, as long as the impact to any individual data contributor can be bounded. For example, when computing a histogram of the populations in each U.S. zip code, the servers see only a noisy count for each zip code, gaining little information about the place of residence of any individual data contributor. In the context of securely performing matrix factorization for use in a recommendation system, we allow the servers to learn a noisy count of the number of items that each contributing user has reviewed. Even when combined with arbitrary external data, this limits the servers from gaining any certainty about the existence of a link between any given user and any given item in the system.

Our reliance on a fourth server in the computation introduces a tradeoff between security and efficiency, when compared with the more common reliance on three servers.[2] It is almost certainly easier for an adversary to corrupt two out of four servers than it is to corrupt two out of three. However, as our results demonstrate, the use of a fourth server enables far faster computation, which, for large-scale applications, might make the use of secure computation far more feasible than it was previously.

[2]From a purely logistical standpoint, we do not envision that this requirement will add much complexity. The additional server(s) can simply be run in one or more public clouds. In some cases, as already mentioned, all servers are anyway run by a single entity, so adding a fourth server may be trivial.

4.1.3 Fixed point arithmetic.

There are few results in the MPC literature that support fixed point computation with malicious security. In part, this is because most malicious secure MPC relies on authentication of field elements, but field elements are not easily "truncated" to handle the rounding of fixed-point values. The most efficient that we know of is the work by Mohassel and Rindal, which uses replicated sharing in the three party, honest majority setting [50], modifying the protocol of Furakawa et al. [51]. Their construction requires each party to send 11 ring elements for each multiplication (it is possible that this can be improved to 8 ring elements, using the more recent result of Araki et al.[52]).

With a bit of care, we show that we can extend the four-party protocol of Gordon et al. [48] to handle fixed point arithmetic, without any additional overhead, requiring each party to send just 1.5 ring elements for each multiplication. This provides about a 5X improvement in communication over Mohassel and Rindal. The protocol of Gordon et al. proceeds through a duel execution of masked circuit evaluation: for circuit wire i carrying value w_i, one pair of parties holds $w_i + \lambda_i$, while the other holds $w_i + \lambda'_i$, where λ_i, λ'_i are random mask values known to the opposite pair. To ensure that nobody has cheated in the execution, the two pairs of parties compute and compare $w_i + \lambda_i + \lambda'_i$. This already supports computation over an arbitrary ring, with malicious security. However, if w_i is a fractional value, the two random masks may result in different rounded values, causing the comparisons to fail. We show how to handle rounding errors securely, allowing us to leverage the efficiency of this protocol for fixed point computation.

4.1.4 Four party, linear-time, oblivious shuffle.

The experimental results of Mazloom and Gordon have complexity $O(V\alpha + E)\log(V\alpha + E)$, where $\alpha = \alpha(\epsilon, \delta)$ is a function of the desired privacy parameters, E is the number of edges in the graph, and V is the number of nodes. The authors also show how to improve the

asymptotic complexity to $O(V\alpha + E)$, removing the $\log$ factor by replacing a circuit for performing an oblivious shuffle of the data with a linear-time oblivious shuffle. They don't leverage this improvement in their experimental results, because it seems to require encrypting and decrypting the data inside a secure computation. (Additionally, for malicious security, it would require expensive zero-knowledge proofs.) When comparing our four-party oblivious shuffling protocol with their semi-honest construction, they require 540X more AES calls and 140X communication.

Operating in the 4-party setting allows us to construct a highly efficient, linear-time protocol for oblivious shuffle. One of the challenges we face in constructing this shuffle protocol is that we have to authenticate the values before shuffling, and verify correctness of the values after shuffling, and because we are committed to computing over elements from $\mathbb{Z}_{2^k}$, we need to authenticate ring values. Recently, Cramer et al. [53] proposed a mechanism for supporting arithmetic circuits over finite rings by constructing authentication in an "extension ring": to compute in $\mathbb{Z}_{2^k}$, they sample $\alpha \leftarrow \mathbb{Z}_{2^s}$, and use a secret-sharing of $\alpha x \in \mathbb{Z}_{2^{k+s}}$ for authentication. We adopt their construction in our shuffle protocol to ensure the integrity of the data during shuffling.

4.1.5 Computation over a ring.

Both the work of Nayak et al. [11] and Mazloom and Gordon [47] use Boolean circuits throughout the computation. Boolean circuits are a sensible choice when using sorting and shuffling circuits, which require bit comparisons. Additionally, as just discussed, Boolean circuits provide immediate support for fixed point computation, removing one further barrier. However, for the apply phase, where, for example, we compute vector gradients, computation in a ring (or field) is far more efficient. With the introduction of our four-party shuffle, which is not circuit-based, and after modifying Gordon et al. [48] to support fixed-point computation, there is no longer any reason to support computation on Boolean values. We construct a method for securely converting the shared, and authenticated values

used in our shuffle protocol into the "masked" ring values required for our four-party computation of the Apply phase. For the problem of Matrix Factorization on dataset of 1 million ratings, the Apply phase of Mazloom and Gordon [47] requires 550X more AES calls and 370X more bandwidth than ours.

4.2 Our Main Framework Intuition

We use the secure computation protocol by Gordon et al. for four parties, tolerating one malicious corruption [48]. We provide an overview of the construction here. The four parties are split into two groups, and each group will perform an evaluation of the circuit to be computed. The invariant throughout each evaluation is that both evaluating parties hold $x+\lambda_x$ and $y+\lambda_y$, where x and y are inputs to a circuit gate, and λ_x, λ_y are random mask values from the ring. After communicating, both parties hold $z + \lambda_z$, where z is the result of evaluating the gate on x and y, and λ_z is another uniformly chosen mask. To maintain this invariant, the evaluating parties need secret shares of $\lambda_x, \lambda_y, \lambda_x\lambda_y$ and λ_z. Securely generating these shares in the face of malicious behavior is typically quite expensive, but, relying on the assumption that only one party is corrupt, it becomes quite simple. Each pair of parties generates the shares for the other pair, and, to ensure that the shares are correctly formed, the pair sends duplicates to each recipient: if any party does not receive identical copies of their shares, they simply abort the protocol.

During the evaluation of the circuit, it is possible for a cheating party to perform an incorrect multiplication, violating the invariant. To prevent this, the two pairs securely compare their evaluations against one another. For wire value z, one pair should hold $z + \lambda_z$, and the other should hold $z + \lambda'_z$. Since the first pair knows λ'_z and the second pair knows λ_z, each pair can compute $z + \lambda_z + \lambda'_z$. They compare these values with the other pair, verifying equality. Some subtleties arise in reducing the communication in this comparison; we allow the interested reader to read the original result.

4.3 Notations

XOR Shares: We let $[X]$ denote a variable which is XOR secret-shared between parties. Arrays have a public length and are accessed via public indices; we use $[X]_i$ to specify element i within a shared array, and $[X]_{i:j}$ to indicate a specific portion of the array containing elements i through j, inclusive. When we write $[x] \leftarrow c$, we mean that both users should fix their shares of x (using some agreed upon manner) to ensure that $x = c$. For example, one party might set his share to be c while the other sets his share to 0.

Additive Shares: We denote the 2-out-of-2 additive shares of a value x between two parties P_1 and P_2 to be $[x]_1$ and $[x]_2$, and between two parties P_3 and P_4 to be $[x]_3$ and $[x]_4$ ($x = [x]_1 + [x]_2 = [x]_3 + [x]_4$). When it is clear, we use $[x]$ instead of $[x]_i$ to denote the share of x held by the i^{th} party. Additive secret shares are used in all steps of the graph computation model except for the Apply phase. In Apply phase, data is converted from additive secret shares to masked value and back.

Masked Values For a value $x \in Z_{2^k}$, its masked value is defined as $m_x \equiv x + \lambda_x$, where $\lambda_x \in Z_{2^{k+s}}$ is sampled uniformly at random. In our four party computation model, for a value x, P_1 and P_2 hold the same masked value $x + \lambda_x$ and P_3 and P_4 hold the same $x + \lambda'_x$. λ_x is provided by P_3 and P_4 while P_1 and P_2 hold shares of λ_x. Similarly, λ'_x is provided by P_1 and P_2 while P_3 and P_4 hold shares of λ'_x

Doubly Masked Values Four players can locally compute the same doubly masked value for x from their shares as $d_x \equiv x + \lambda_x + \lambda'_x = m_x + \lambda'_x = m'_x + \lambda_x$.

Share or Masked Value of a Vector When X is a vector of data, i.e, $X = \{x_1, ..., x_n\}$, we define $[X] \equiv \{[x_1], ..., [x_n]\}$, $\lambda_X \equiv \{\lambda_{x_1}, ..., \lambda_{x_n}\}$, $m_X \equiv \{m_{x_1}, ..., m_{x_n}\}$ and $d_X \equiv \{d_{x_1}, ..., d_{x_n}\}$.

Fixed Point Representation All inputs, intermediate values, and outputs are k-bit fixed-point numbers, in which the least d significant bits are used for the fraction part. We represent a fixed-point number x by using a ring element in $Z_{2^{k+s}}$, where s denotes our statistical security parameter.

MAC Representation We adapt the technique used in SPDZ2k [53] for our MAC over rings

scheme. For a value $x \in Z_{2^k}$ and for a MAC key α, $\alpha \in Z_{2^s}$. The MAC value of x is defined as $\mathsf{MAC}_\alpha(x) \equiv \alpha x \in Z_{2^{k+s}}$. In our framework, $\mathsf{MAC}_\alpha(x)$ always exist in the form of additive secret shares.

We denote that in our four-party framework, presented in section 4, all the values, or the additive shares, or masked values (data, MAC key, MAC_α(x)) are represented as elements in the ring $Z_{2^{k+s}}$. However, the range of the data is in Z_{2^k}, and the MAC key is in Z_{2^s}.

Multi-Sets: We represent multi-sets over a set V by a $|V|$ dimensional vector of natural numbers: $D \in \mathbb{N}^{|V|}$. We refer to the ith element of this vector by $D(i)$. We use $|D|$ in the natural way to mean $\sum_{i=1}^{|V|} D(i)$. We use $\mathcal{DB}_i$ to denote the set of all multi-sets over V of size i, and $\mathcal{DB} = \bigcup_i \mathcal{DB}_i$. We define a metric on these multi-sets in the natural way: $|D_1 - D_2| = \sum_{i=1}^{|V|} |D_1(i) - D_2(i)|$. We say two multi-sets are *neighboring* if they have distance at most 1: $|D_1 - D_2| \leq 1$.

Neighboring Graphs: In our main protocol of Section 3.3.2, the input is a data-augmented directed graph, denoted by $G = (V, E)$, with user-defined data on each edge. We need to define a metric on these input graphs, in order to claim security for graphs of bounded distance.[3] For each $v \in V$, we let $\mathsf{in\text{-}deg}(v)$ and $\mathsf{out\text{-}deg}(v)$ denote the in-degree and out-degree of node v. We define the *in-degree profile* of a graph G as the multi-set $\mathsf{D_{in}}(G) = \{\mathsf{in\text{-}deg}(v_1), \ldots, \mathsf{in\text{-}deg}(v_n)\}$. Intuitively, this is a multi-set over the node identifiers from the input graph, with vertex identifier v appearing k times if $\mathsf{in\text{-}deg}(v) = k$. We define the *full-degree profile* of G as the pair of multi-sets: $\{\mathsf{D_{in}}(G), \mathsf{D_{out}}(G)\}$, where $\mathsf{D_{out}}(G) = \{\mathsf{out\text{-}deg}(v_1), \ldots, \mathsf{out\text{-}deg}(v_n)\}$. We now define two different metrics on graphs, using these degree profiles. Later in this section, we provide two different security definitions: we rely on the first distance metric below when claiming security as defined in Definition 9, and rely on the second metric below when claiming security as defined in Definition 7.

[3] In Section 3.1.1, the input to the computation is a multi-set of elements drawn from some set S, rather than a graph, so we use the simple distance metric described above to define the distance between inputs.

4.4 Overview of Our Main Framework

Our construction follows the graph-parallel computation model in which the computation is done in parallel using three main operations; Gather, Apply and Scatter. We partition the players into 2 groups, and in each group, there are two players. For ease of explanation, we name the parties in the first group, Alice and Bob (P_1, P_2), and parties in the second group, Charlie and David (P_3, P_4). These parties collaboratively compute a functionality, Matrix Factorization as an example, and each group is responsible for performing an operation that will then be verified by the other group. For example, one group securely shuffles the data, and the other group verifies that the data was unmodified, then the latter group performs the operation that accesses the data (e.g., gather), and the former group verifies the correctness of that operation. As described previously, each data access operation, Gather or Scatter, is always followed by a Shuffle operation in order to hide the graph edge structure. As long as the group that accesses the data does not know the permutation of the shuffle, our scheme remains secure. In our explanation of the construction, we assume Alice and Bob are responsible to access the data, and Charlie and David handle the shuffling. Before each group (operation group) does their operation, 4 parties securely compute the MAC, then the verification group verifies the MAC after the operation in order to prevent the malicious adversary from modifying the data.

4.5 Construction

The overview of the framework is shown in Figure 4.1. For illustration purposes, the steps 1-4 are only conducted on the left vertex of each edge. In order to perform a complete iteration of the Matrix Factorization, which is handled by a bipartite graph, these steps should be done twice, once on the left vertices, followed by the right vertices.

Data Structure: In our framework, the data is represented in a graph structure $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, in which vertices contain user and item profiles, and edges represent the relation between

Four-Party Secure Graph Parallel Computation Framework

Input: User input is a directed graph, $\mathsf{G(E,V)}$, secret shared between the parties:
$Alice, Bob$ hold a secret share of E, such that $[E]_A + [E]_B = E \bmod 2^{k+s}$.
$Charlie, David$ hold a secret share of E, such that $[E]_C + [E]_D = E \bmod 2^{k+s}$.
($[E]_A, [E]_B, [E]_C, [E]_D \in \mathbb{Z}_{2^{k+s}}$, and $E \in \mathbb{Z}_{2^k}$).

Protocol:
Note: The following steps are conducted on the left vertex of each edge (for example in computing Histogram). In order to perform one single iteration of Matrix Factorization, these steps should be done twice, once on the left vertices, then on the right vertices.

1. **Oblivious Shuffle** Four players make a call to $\mathcal{F}_{\mathsf{shuffle}}([E])$ to shuffle their shares. They receive shares of shuffled edges, $[E^{(1)}] \leftarrow [\pi(E)]$.

2. **Oblivious Gather** Four players make a call to $\mathcal{F}_{\mathsf{gather}}([E^{(1)}])$ to aggregate edge data into vertices. $Alice, Bob$ receive $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\}, ..., \{W_{n_1}..W_{n_j}\}]$, $[\beta]$, where V is the vector of gathered vertices, and $W \equiv \beta V$ is V's MAC. Charlie and David receive MAC key β
Note: Gather leaks the noisy degree of the vertices, however, this leakage preserves differential privacy.

3. **Oblivious Apply** Four players make a call to $\mathcal{F}_{\mathsf{apply}}$ $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\}, ..., \{W_{n_1}..W_{n_j}\}]$, $[\beta]$) to compute the function of interest on the vertex data. Four players receive updated values of shares of vertices $[\{V^{(1)}_{1_1}..V^{(1)}_{1_i}\}, ..., \{V^{(1)}_{n_1}..V^{(1)}_{n_j}\}]$.

4. **Oblivious Scatter** Four players call $\mathcal{F}_{\mathsf{scatter}}$ $([\{V^{(1)}_{1_1}..V^{(1)}_{1_i}\}, ..., \{V^{(1)}_{n_1}..V^{(1)}_{n_j}\}])$ to update the edges and receive $[E^{(2)}]$.
Each group re-randomizes the edges before enter the next round of computation (Step 1).

Output: Trained model parameters (user and items profiles)

Figure 4.1: Four parties collaborate to securely compute a functionality on their private data in parallel fashion, using graph model of computation.

connected vertices. Each edge, represented as E, has five main elements, ($\mathsf{E.I_{id}}$, $\mathsf{E.r_{id}}$, $\mathsf{E.I_{data}}$, $\mathsf{E.r_{data}}$, $\mathsf{E.isReal}$), where isReal indicates if an edge is "real" or "dummy". Each vertex, V, contains two main elements, ($\mathsf{V_{id}}, \mathsf{V_{data}}$). The $\mathsf{V_{data}}$ storage is large enough to hold edge data from multiple adjacent edges during the gather operation.

Dummy Generation: Before the main protocol begins, a number of dummy edges will be generated according to an appropriate distribution, and concatenated to the list of real edges, in order to provide (ϵ, δ)-Differential Privacy. Therefore, the input to the framework is a concatenated list of real and dummy edges, and list of vertices. The circuit for generating these dummies, together with the noise distribution, is taken directly from the work of Mazloom and Gordon, so we do not describe it again here. The cost of this execution is very small relative to the rest of the protocol. And it is only done once at the beginning of the any computation, regardless of how many iterations the computation has (both histogram and matrix factorization computation have only one dummy generation operation). These dummy edges are marked with a (secret shared) flag isReal, indicating that they should not influence the computation during the Apply phase, but they still contain node identifiers, so they contribute to the number of memory accesses to these nodes during the Gather and Scatter phases.

Step 0. Input preparation: We assume the input data is additive secret-shared between the parties in each group, so that parties in each group together can reconstruct the data. For example, Alice and Bob receive 2-out-of-2 secret shares of E, such that $[E]_A + [E]_B = E \bmod 2^{k+s}$, as shown in Figure 4.2.

Step 1. Oblivious Shuffle: In this step, Charlie and David shuffle the edges. Shuffling edges between the gathering of data at the left nodes and the gathering of data at the right nodes ensures that the graph edge structure remains hidden. Alice and Bob are responsible to verify that the shuffle operation has been done correctly. To facilitate that, before the shuffle begins, they need to compute a MAC tag for each edge. To compute the MACs, first Alice and Bob agree on a random value α, then all parties call a functionality, $\mathcal{F}_{\mathsf{MAC}}$, to securely compute shares of MAC tags, which we call α-MACs. To perform the shuffle, as described in Figure 4.3 Charlie and David agree on a random permutation π, then each locally shuffles its shares of the edges E along with its shares of the corresponding MAC tags, according to permutation π. At the the end of this step, Alice and Bob receive the

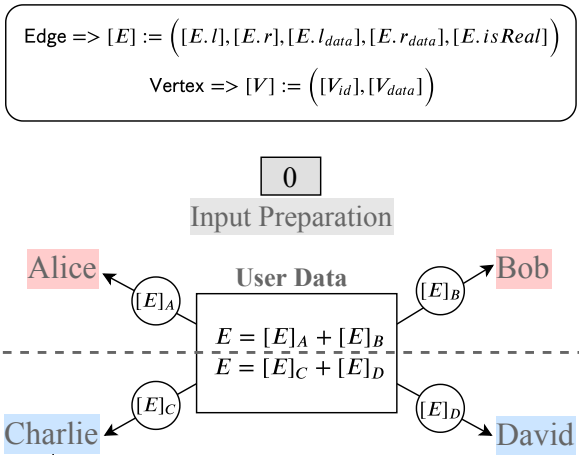


Figure 4.2: Input preparation phase: input data is secret-shared between both groups of parties

shuffled edges from the other group, and call the verification function, $\mathcal{F}_{\mathsf{CheckZero}}$. If the verification fails, it means one of the parties in the shuffling group, either Charlie or David, has cheated and modified the edge data, and the protocol aborts; otherwise they continue to the next phase.

Step 2. Oblivious Gather: The next operation after Shuffle is the Gather operation, which requires access to the node identifiers, and will be handled by Alice and Bob. In turn, Charlie and David should be able to verify the correctness of the Gather operation. Therefore, before the Gather operation, Charlie and David agree on a random value β, and all parties make a sequence of calls to the $\mathcal{F}_{\mathsf{MAC}}$ functionality, generating a new MAC tag for each data element of each edge. That is, they construct three tags per edge: one tag for each of the two vertex ids, and one tag for the edge data. The Gather operation is performed on only one side of each edge at a time; in one iteration of the protocol, data is gathered at all of the left vertices, and in the next iteration, it is gathered at all of the right vertices. Gather for the left vertices is described in Figure 4.21: for each edge, Alice and Bob first reconstruct the id of the left vertex $E.l_{id}$, locate the corresponding vertex, and then append the data of the other

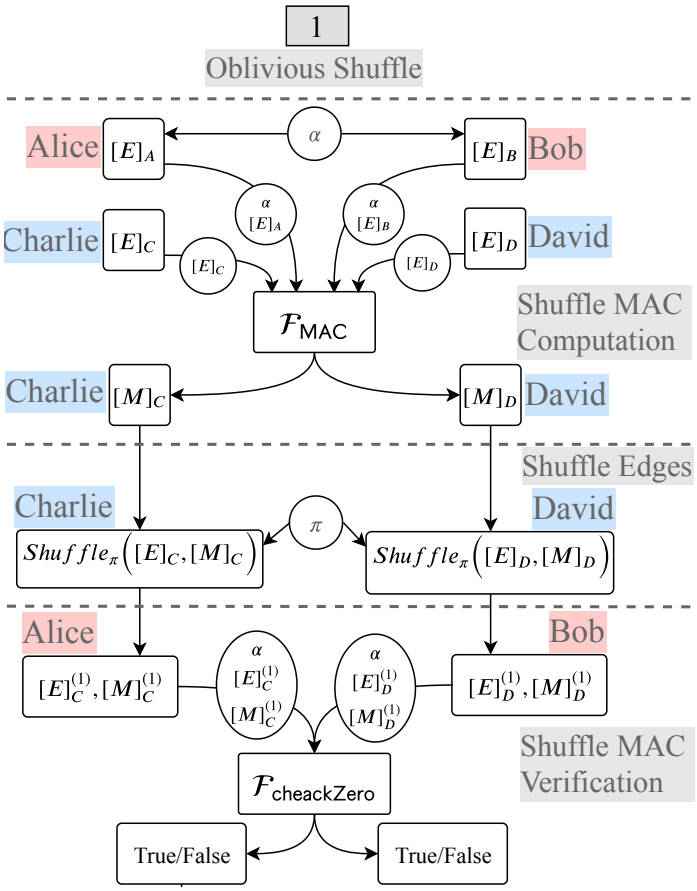


Figure 4.3: Oblivious Shuffle operation with MAC computation and verification

end of the edge, i.e. the data of the right vertex, $[\mathsf{E.r_{data}}]$ with its MAC tags, to the left vertex data storage. They do the same for all the incoming edges to that vertex. Note that in the next iteration of the algorithm they follow the same procedure for the right vertex, if applicable. When Alice and Bob access one side of each edge, left vertex for example, they learn the number of times each vertex is being accessed which leaks the degree of each vertex in the graph, however, due to dummy edges we shuffled with real ones, what they learn is the noisy degree of each vertex which preserve deferential privacy. At the end of this phase, Charlie and David verify the correctness of Gather operation by calling $\mathcal{F}_{\text{verifyMAC}}$, verifying that vertex ID's were opened correctly, and that the data shares were copied correctly. They abort if the verification fails. We note that, in addition to modifying data, a malicious

adversary might try to move data to the wrong vertex. From a security standpoint, this is equivalent to the case that the adversary moves data to the correct vertex during Gather, but modifies the shares of the authenticated identifier. To simplify the analysis, we assume that the adversary moves data to the correct vertex.

Step 3. Oblivious Apply: This operation consists of three sub operations, first, secret shares of data should be converted to masked values, then the main functionality (e.g. gradient descent) is applied on the masked values, and finally the masked values are converted back to secret shared data to be used in the other steps of the framework.

Step 3.1. Secure Share-Mask Conversion: All the parties participate in the Apply phase, providing their shares as input to the Arithmetic Circuit that computes the intended functionality. However, in order to prepare the private data for the Apply operation, the secret-share values need to be transformed into Mask values. Then they call the $\mathcal{F}_{\mathsf{ShareConv}}$ functionality and collaboratively transform the share values $[V]$ to masked values $V + \lambda$.

Step 3.2. Computing the function of interest on input data: The Apply operation compute the function of interest on the input data: for example it could performing addition for Histogram problem, or computing gradient descent for Matrix Factorization. The parties execute the four-party protocol described in Figure 4.11 to evaluate the relevant circuit.

Step 3.3. Secure Mask-Share Conversion: After the Apply phase, we need to convert the data back the additive secret sharing. This step can be done locally without any interaction between the players, as in Figure 4.9.

Step 4. Oblivious Scatter: Now the result of each computation resides inside the corresponding vertex. We need to update the data on the edges with the freshly computed data. In this step, all players copy the updated data from the vertex to the incoming (or outgoing) edges. The players refer to the list of opened ID's obtained during Gather to decide how to update each edge. Recall, edges are held as additive secret shares. The update of the edge data can be done locally. Finally, they re-randomize all the shares.

This explanation and accompanying diagrams only show the graph operations applied on

the outgoing edges. To complete one round of the graph computation, we need to repeat the steps 1-6 for incoming edges as well.

4.6 Building Blocks

In this section, we explain the details of each small component and building block in graph operations, present their real vs. ideal world functionalities, and provide the security proofs for each of them, under a single malicious corruption. We partition the 4 partied into 2 groups, first group consist of P_1 and P_2, we also call them Alice and Bob, and the second group is P_3, P_4 that are named Charlie and David, respectively. For ease of explanation, we name the parties in the first group, Alice and Bob, and parties in the second group, Charlie and David.

4.6.1 MAC Computation and Verification: Authentication For Additive Shares Over A Ring

One of the main challenges we face in constructing malicious secure version of the Graph operations is that we have to authenticate the values before each operation, and then verify correctness of the values after the operation is done. We adapt the MAC computation and Verification technique proposed in SPDZ2k [53]. In this part, we describe the ideal functionality and the real world protocol to generate MAC values for additive secret shares over a ring and we provide the security proof with simulation.

Theorem 3. *The MAC computation protocol* Π_{MAC} *(Figure 4.5) securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{MAC}}$ *(Figure 4.4) with abort, under a single malicious corruption.*

We provide a simulation for P_1 and P_3. The simulation for P_2 and P_4 is identical to that of P_1 and P_3 respectively.

First, a simulation for P_1:

FUNCTIONALITY $\mathcal{F}_{\mathsf{MAC}}$

Inputs: P_1, P_2 provide shares $[X] = \{[x_1], \ldots, [x_n]\}$ and MAC key α. P_3, P_4 only provide shares $[X]$.
Functionality:

- Verify that $X = [X]_1 + [X]_2 = [X]_3 + [X]_4$. If the check does not pass, send abort to all parties.
- If P_1 and P_2 submit different values of α, send abort to all parties.
- Compute $Y = \alpha X$.

Output: P_3 and P_4 receive $[Y]$.

Figure 4.4: MAC computation ideal functionality

PROTOCOL Π_{MAC}

Inputs: P_1, P_2 have shares $[X]$ and MAC key α. P_3, P_4 only have shares $[X]$. F is a PRF.
Protocol:

1. P_1 and P_2 sample a random PRF key k, by making a call to $\mathcal{F}_{\mathsf{coin}}$.
2. P_1 sends $[Y] = \{\alpha[X_i] + F_k(i) | i = 1, ..., n\}$ to P_3.
3. P_2 sends $[Y] = \{\alpha[X_i] - F_k(i) | i = 1, ..., n\}$ to P_4.
4. Four parties make a call to $\mathcal{F}_{\mathsf{mult}}(\alpha, \alpha, [X]_3, [X]_4)$. P_3 and P_4 receive shares $[Y^{(1)}] \leftarrow [\alpha X]$
5. P_3 and P_4 compute $[Z] = [Y - Y^{(1)}]$ and verify $Z = 0$ by making a call to $\mathcal{F}_{\mathsf{checkZero}}([Z])$. If the functionality returns false, they send abort to P_1 and P_2 and terminate.

Output: P_3 and P_4 output $[Y^{(1)}]$.

Figure 4.5: MAC computation protocol

- $\widetilde{k}$: S samples a random PRF key $\widetilde{k}$ and hands it to P_1 to simulate the output P_1 receives from $\mathcal{F}_{\mathsf{coin}}$. S receives P_1's input $[X]_1$ and α from the distinguisher and places it in the input tape of P_1. S computes the message $[Y]$ locally and observes the message

that P_1 sends to P_3. If P_1 sends $[Y']$ in Step 2 such that $[Y' - Y] \neq 0 \mod 2^k$, S aborts and outputs the partial view.

- S observes the message that P_1 sends to $\mathcal{F}_{\mathsf{mult}}$: if P_1 modifies α before sending it to the functionality, S aborts and outputs the partial view. Else, S submits P_1's inputs $(\alpha, [X])$ to the functionality $\mathcal{F}_{\mathsf{MAC}}$ and outputs whatever P_1 outputs.

Claim 1. *For the simulator S corrupting party P_1 as described above and interacting with the functionality $\mathcal{F}_{\mathsf{MAC}}$,*

$$\left\{\text{HYBRID}_{\pi_{\mathsf{MAC}},\mathcal{A}(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad \stackrel{c}{\equiv} \quad \left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{MAC}},S(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Case 0: If P_1 follow the protocol honestly,

$\left\{\text{HYBRID}_{\pi_{\mathsf{MAC}},\mathcal{A}(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{k, o_1, o_2, o_3, o_4\}$

$\left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{MAC}},S(z)}(X, Y, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{k}, \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3, \widetilde{o}_4\}$

The joint distributions are identical in both worlds as all the messages $k, \widetilde{k}$ are distributed uniformly at random, and the outputs are identically distributed.

Case 1: If P_1 cheats by sending the wrong shares for $[Y]$ as in Step 2, or sending a different α to $\mathcal{F}_{\mathsf{mult}}$ in Step 4, S will abort in the ideal world and the joint distribution is $\{\widetilde{k}, \perp\}$. In the hybrid world, this is equivalent to P_1 sending the wrong α or sending shares $[Y + D]$ ($D \neq 0 \mod 2^k$ is the vector of additive terms) instead of $[Y]$, P_3 and P_4 will send $[Y + D - \alpha X] = [D]$ to $\mathcal{F}_{\mathsf{checkZero}}$, which will output abort and the joint distribution in the hybrid world is $\{k, \perp\}$. Similar to case 0, the joint distributions in both worlds are also identical.

In conclusion, the joint distributions in both worlds are identical.

Now, we provide a simulation for P_3:

- S receives P_3's input $[X]$ from the distinguisher and puts it in the input tape of P_3.

- $[\widetilde{Y}]$: S samples a random number as shares $[\widetilde{Y}]$, then it hands $[\widetilde{Y}]$ to P_3 to simulate the message $[Y]$ P_3 receives from P_1 in Step 2.

- $[\widetilde{Y}^{(1)}]$: S observes the message that P_3 sends to $\mathcal{F}_{\mathsf{mult}}$, if it is not $[X]$, S sets $\mathsf{abort}_0 = 1$ and hands P_3 random numbers as shares $[\widetilde{Y}'^{(1)}]$. Else, S queries the ideal functionality with input $[X]$ and receives $[\widetilde{Y}^{(1)}]$. S hands $[\widetilde{Y}^{(1)}]$ to P_3 as the output of $\mathcal{F}_{\mathsf{mult}}$.

- $\widetilde{b}$: S observes the messages that P_3 sends to $\mathcal{F}_{\mathsf{checkZero}}$. If P_3 does not send the intended messages (P_3 sends $[Z']$ such that $[Z'] - [Z] \neq 0 \mod 2^k$), S sets $\mathsf{abort}_1 = 1$ (S can compute the messages that P_3 has to send to $\mathcal{F}_{\mathsf{CheckZero}}$). If $\mathsf{abort}_0 = 1$ or $\mathsf{abort}_1 = 1$, S hands $\widetilde{b} = \mathsf{false}$ to P_3 and aborts. Else, S hands $\widetilde{b} = \mathsf{true}$ and outputs whatever P_3 outputs

Claim 2. *For the simulator S corrupting party P_3 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{MAC}}$,*

$$\left\{\textsc{hybrid}_{\pi_{\mathsf{MAC}},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad \stackrel{c}{\equiv} \quad \left\{\textsc{ideal}_{\mathcal{F}_{\mathsf{MAC}},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Case 0: If P_3 follows the protocol honestly,

$$\left\{\textsc{hybrid}_{\pi_{\mathsf{MAC}},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad = \quad \{[Y],[Y^{(1)}],b,o_1,o_2,o_3,o_4\}$$

$$\left\{\textsc{ideal}_{\mathcal{F}_{\mathsf{MAC}},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad = \quad \{[\widetilde{Y}],[\widetilde{Y}^{(1)}],\widetilde{b},\widetilde{o}_1,\widetilde{o}_2,\widetilde{o}_3,\widetilde{o}_4\}$$

The joint distributions are identical in both worlds as $[Y],[\widetilde{Y}]$ are distributed uniformly at random, $[Y^{(1)}]$ and $[\widetilde{Y}^{(1)}]$ are identical, $\widetilde{b} = b = \mathsf{true}$.

Case 1: If P_3 submits the vector of additive terms D such that $D \neq 0 \mod 2^k$ to $\mathcal{F}_{\mathsf{mult}}$. In

the ideal world, S hands P_3 random values $[\widetilde{Y}'^{(1)}]$ as outputs of $\mathcal{F}_{\mathsf{mult}}$, and $\widetilde{b} = \mathsf{false}$ as output to $\mathcal{F}_{\mathsf{CheckZero}}$ and aborts. The joint distribution in the ideal world is $\{[\widetilde{Y}], [\widetilde{Y}'^{(1)}], \widetilde{b} = \mathsf{false}, \perp\}$. In the hybrid world, P_3 receives $[Y'^{(1)}]$ from $\mathcal{F}_{\mathsf{mult}}$. When submit shares $[Z]$ to $\mathcal{F}_{\mathsf{CheckZero}}$, P_3 can also cheat by providing additive vector E such that $E \in Z_{2^{k+s}}$: $[Z'] = [Z + E] = [Y'^{(1)} - Y + E] = [\alpha(X + D) - \alpha X + E] = [\alpha D + E]$. $\mathcal{F}_{\mathsf{CheckZero}}$ will output true if P_3 happens to choose D, E such that $D \neq 0 \mod 2^k, E \in Z_{2^{k+s}}$ and $\alpha D + E = 0 \mod 2^{k+s}$.

We claim that the chance for this to happen is at most 2^{-s}. As $D \neq 0 \mod 2^k$, there exists at least one $d_i \in D$ such that $d_i \neq 0 \mod 2^k$. Let $d_i = 2^t u$ where $t < k$ and u is odd. $Pr[\alpha d_i + e_i = 0 \mod 2^{k+s}] = Pr[e_i = -\alpha d_i \mod 2^{k+s}] = Pr[\frac{e_i}{2^t} = -\alpha u \mod 2^{k+s-t}] = 2^{-s}$ as α is distributed uniformly at random. So, $Pr[\alpha D + E = 0 \mod 2^{k+s}] \leq 2^{-s}$.

The joint distribution in the hybrid world is $\{[Y], [Y'^{(1)}], b = \mathsf{true}, o_1, o_2, o_3, o_4\}$ with probability at most 2^{-s} and $\{[Y], [Y'^{(1)}], b = \mathsf{false}, \perp\}$ with probability at least $1 - 2^{-s}$. The later and the joint distribution in the ideal world are identically distributed. So, the joint distributions in both worlds are statistically close.

Case 2: If P_3 only cheats at Step 5, abort happens in both worlds, and the joint distributions between both worlds are identical. In conclusion, the joint distributions in both worlds are statistically close.

4.6.2 Share-Mask Conversion

We construct a method for securely converting the shared, and authenticated values used in our Shuffle and Gather into the "masked" ring values required for our four-party computation of the Apply phase. At the end of the Apply phase, the result of the 4-party computation from masked values will be converted back into secret shares, because the following steps such as Scatter performs on secret shares.

Theorem 4. *The share-mask conversion protocol $\Pi_{[x]\to m_x}$ (Figure 4.7) securely realizes the ideal functionality $\mathcal{F}_{[x]\to m_x}$ (Figure 4.6) with abort, under a single malicious corruption.*

FUNCTIONALITY $\mathcal{F}_{[x]\to m_x}$

Inputs: P_1 and P_2 provide shares $[\beta]$, $[X]$ and $[Y](Y \equiv \beta X)$. P_3 and P_4 provide β.
Functionality:

- Wait for inputs from P_1 and P_2 and reconstruct β, X, and Y. Wait for inputs from P_3 and P_4. Verify that P_3 and P_4 send the same β.
- Verify that $Y = \beta X$. If the check fails, send abort to all parties.
- Sample shares $[\lambda_X]_1$, $[\lambda_X]_2$, $[\lambda'_X]_1$, $[\lambda'_X]_2$ uniformly at random, then reconstruct λ_X and λ'_X.
- Compute $m_X = X + \lambda_X$ and $m'_X = X + \lambda'_X$.

Output: Send $(m_X, \lambda'_X, [\lambda_X]_1)$ to P_1, $(m_X, \lambda'_X, [\lambda_X]_2)$ to P_2, $(m'_X, \lambda_X, [\lambda'_X]_1)$ to P_3 and $(m'_X, \lambda_X, [\lambda'_X]_2)$ to P_4.

Figure 4.6: Ideal Functionality to convert additive shares to masked values

We provide a simulation for P_1 and P_3. The simulation for P_2 and P_4 is similar to that of P_1 and P_3 respectively.

First, a simulation for P_1:

- S receives P_1's inputs from the distinguisher and puts them in the input tape of P_1. S submits the inputs to the ideal functionality and receives ($\widetilde{m}_X$, $\widetilde{\lambda}'_X$, $[\widetilde{\lambda_X}]_1$).
- $[\widetilde{\lambda}'_X]_1$: S hands P_1 random ring elements as shares $[\widetilde{\lambda'_X}]_1$ to simulate the messages P_1 receives from $\mathcal{F}_{\mathsf{coin}}$ in the hybrid world in Step 1.
- $[\widetilde{\lambda}'_X]_2$: S computes shares $[\widetilde{\lambda'_X}]_2$ from λ'_X and the previous message, and sends $[\widetilde{\lambda'_X}]_2$ to P_1 to simulate the messages P_1 receives from $\mathcal{F}_{\mathsf{coin}}$ in the hybrid world in Step 2.
- $[\widetilde{\lambda}_X]_1$: S hands P_1 $[\widetilde{\lambda_X}]_1$, which was obtained from the functionality, to simulate the messages P_1 receives from $\mathcal{F}_{\mathsf{coin}}$ in the hybrid world in Step 3.
- $[\widetilde{m}_X]_2$: S computes $[m_X]_1$, $[m'_X]_1$, and $[Y']_1$ to mirror P_1's actions in Step 5. S sends P_1 the message $[m_X]_2 = m_X - [m_X]_1$ to P_1 to simulate the message P_1 receives from

PROTOCOL $\Pi_{[x]\to m_x}$

Inputs: P_1 and P_2 have shares $[\beta]$, $[X]$ and $[Y \equiv \beta X]$ P_3 and P_4 have β.
Functionality:

1. P_1, P_2, and P_3 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda'_X]_1$
2. P_1, P_2, and P_4 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda'_X]_2$
3. P_1, P_3, and P_4 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda_X]_1$
4. P_2, P_3, and P_4 make calls to $\mathcal{F}_{\mathsf{coin}}$ to sample $[\lambda_X]_2$
5. P_1 and P_2 compute $[m_X] = [X] + [\lambda_X]$, $[m'_X] = [X] + [\lambda'_X]$, $[Y'] \leftarrow [Y] + [\beta]\lambda'_X$ (where $\lambda'_X = [\lambda'_X]_1 + [\lambda'_X]_2$).
6. P_1 and P_2 reconstruct $m_X \leftarrow \mathsf{open}([m_X])$.
7. P_1 sends his shares $[m'_X]$, $[Y']$ to P_3. P_2 sends his shares $[m'_X]$, $[Y']$ to P_4.
8. P_3 and P_4 computes $[Z] = \beta[m'_X] - [Y']$ and make a call to $\mathcal{F}_{\mathsf{CheckZero}}([Z])$. If the functionality outputs $b = \mathsf{false}$, they call abort. Else, if $b = \mathsf{true}$, they open $m'_X \leftarrow \mathsf{open}([m'_X])$.
9. All parties compute $d_X = m_X + \lambda_X = m'_X + \lambda'_X$, P_1 and P_3 compare $h_1 = H(d_X)$ with each other, while P_2 and P_4 compare $h_2 = H(d_X)$ with each other. If any group sees a mismatch, they call abort.

Output: P_1 and P_2 output m_X, λ'_X, $[\lambda_X]$. P_3 and P_4 output m'_X, λ_X, $[\lambda'_X]$.

Figure 4.7: Real-world protocol to convert additive shares to masked values

P_2 in Step 6. S also observes the message P_1 send to P_2, if P_1 sends $[m_X]_1 + D$ where $D \neq 0 \mod 2^k$, S sets $\mathsf{abort}_1 = 1$.

- In Step 7, if P_1 sends $[m'_X]_1 + D'$, $[Y']_1 + E$, where either $D' \neq 0 \mod 2^k$ or $E \neq 0 \mod 2^{k+s}$, S aborts and outputs the partial view.

- $\widetilde{h}_1$: S computes $d_X = m_X + \lambda'_X$ and sends $\widetilde{h}_1 = H(d_X)$ to P_1 to simulate the message P_1 receives from P_3 in Step 9. If $\mathsf{abort}_1 = 1$, S aborts and outputs the partial view. If $\mathsf{abort}_1 = 0$, S observes the message $\hat{h}_1$ that P_1 sends to P_3. If $\hat{h}_1 \neq \widetilde{h}_1$, S also aborts and outputs the partial view. Else, S outputs whatever P_1 outputs.

Claim 3. *For the simulator S corrupting party P_1 as described above, and interacting with the functionality $\mathcal{F}_{[x]\to m_x}$,*

$$\left\{\text{HYBRID}_{\pi_{[x]\to m_x},\mathcal{A}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{[x]\to m_x},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof:

Case 0: First, we consider the case where P_1 follows the protocol honestly. In this case, the joint distribution of the view of P_1 and the output in the hybrid and ideal execution is:

$$\left\{\text{HYBRID}_{\pi_{[x]\to m_x},\mathcal{A}(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{[\lambda'_X]_1,[\lambda'_X]_2,[\lambda_X]_1,[m_X]_2,h_1,o_1,o_2,o_3,o_4\}$$

$$\left\{\text{IDEAL}_{\mathcal{F}_{[x]\to m_x},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{[\widetilde{\lambda'_X}]_1,[\widetilde{\lambda'_X}]_2,[\widetilde{\lambda_X}]_1,[\widetilde{m_X}]_2,\widetilde{h}_1,\widetilde{o}_1,\widetilde{o}_2,\widetilde{o}_3,\widetilde{o}_4\}$$

According to the simulation, $[m_X]_2$ and $[\widetilde{m_X}]_2$ are identical as S can compute the message exactly from the output received from the ideal functionality and P_1's input. h_1 and $\widetilde{h}_1$ are also identical as d_X is uniquely determined from the received output. $[\lambda'_X]_i$ and $[\widetilde{\lambda'_X}]_i$ are all distributed uniformly at random, and satisfy the condition that $[\lambda'_X]_1+[\lambda'_X]_2 = [\widetilde{\lambda}'_X]_1+[\widetilde{\lambda}'_X]_2$. Thus, the joint distributions in both worlds are identical.

Case 1: Let $D \neq 0 \mod 2^k$ be the additive terms that P_1 use to modified the opened value of m_X. No matter what the value of D is, if P_1 cheats in Step 7 by sending P_3 $[m'_X+D']$ and $[Y'+E]$ where $D' \neq 0 \mod 2^k$ and $E \neq 0 \mod 2^{k+s}$: in the ideal world, S aborts with the joint distribution $\{[\widetilde{\lambda'_X}]_1,[\widetilde{\lambda'_X}]_2,[\widetilde{\lambda_X}]_1,[\widetilde{m_X}]_2+D,\perp\}$. In the hybrid world, the cheating in Step 7 may go undetected if P_1 chooses D' and E such that $Z' = \beta(m'_X+D')-(Y'+E) = 0 \mod 2^{k+s}$ or $\beta D'-E = 0 \mod 2^{k+s}$. With the same analysis in Section 4.6.1, this happens with probability of at most $1-2^{-s}$. In case $\mathcal{F}_{\mathsf{CheckZero}}$ outputs $b = \mathsf{false}$, the joint distribution

in the hybrid world is $\{[\lambda'_X]_1, [\lambda'_X]_2, [\lambda_X]_1, [m_X]_2 + D, \perp\}$, which is identical to that in the ideal world. So, the joint distributions in both worlds are statistically close.

Case 2: If P_1 does not cheat in Step 7, but he cheats in Step 6 by opening $m_X + D$ with $D \neq 0 \mod 2^k$ in stead of m_X. In Step 9, P_2 and P_4 will complain when they compare the hashes and call abort. Thus, in both worlds, the executions end up in abort. The joint distributions in the ideal and hybrid world are $\{[\widetilde{\lambda'_X}]_1, [\widetilde{\lambda'_X}]_2, [\widetilde{\lambda_X}]_1, [\widetilde{m_X}]_2 + D, \widetilde{h}_1, \perp\}$ and $\{[\lambda'_X]_1, [\lambda'_X]_2, [\lambda_X]_1, [m_X]_2 + D, h_1, \perp\}$ respectively. The joints distributions are identical.

Case 3: P_1 is honest up to Step 9. The only way that P_1 deviates from the protocol is to send the wrong h_1 to P_3, causing all parties to abort. In this case, the joint distributions in both worlds are also identical.

In conclusion, the joint distributions in both worlds are statistically close. ■

Now, a simulation for P_3:

- S receives P_3's inputs from the distinguisher and puts them in the input tape of P_3. S submits the inputs to the ideal functionality and receives ($\widetilde{m}'_X$, $\widetilde{\lambda}_X$, $[\widetilde{\lambda'_X}]_1$).

- $[\widetilde{\lambda}'_X]_1$: S hands P_3 $[\widetilde{\lambda'_X}]_1$, which was obtained from the functionality, to simulate the messages P_3 receives from $\mathcal{F}_{\mathsf{coin}}$ in the hybrid world in Step 1.

- $[\widetilde{\lambda}_X]_1$: S hands P_3 random ring elements as shares $[\widetilde{\lambda_X}]_1$ to simulate the messages P_3 receives from $\mathcal{F}_{\mathsf{coin}}$ in the hybrid world in Step 3.

- $[\widetilde{\lambda}_X]_2$: S computes shares $[\widetilde{\lambda_X}]_2$ from λ_X and the previous message, and sends $[\widetilde{\lambda_X}]_2$ to P_3 to simulate the messages P_3 receives from $\mathcal{F}_{\mathsf{coin}}$ in the hybrid world in Step 4.

- $[\widetilde{m}'_X]_1, [\widetilde{Y'}]_1$: S samples random ring elements in $Z_{2^{k+s}}$ as shares $[\widetilde{m}'_X]_1, [\widetilde{Y'}]_1$ and hands them to P_3 to simulate the messages P_3 receives from P_1 in Step 7.

- $\widetilde{b}$: S computes $[Z]$ itself to mirror P_3's action. S then observes the shares that P_3 sends to $\mathcal{F}_{\mathsf{CheckZero}}$. If P_3 sends $[Z]+E$ for $E \neq 0 \mod 2^{k+s}$, S hands $\widetilde{b} = \mathsf{false}$ to P_3 as output

of $\mathcal{F}_{\mathsf{CheckZero}}$, aborts, and outputs the partial view.

- $[\widetilde{m}'_X]_2$: S computes $[m'_X]_2 = m'_X - [m'_X]_1$ and sends it to P_3 to simulate the message P_1 receives from P_4 in Step 8 when they open m'_X. S also observes the message P_3 send to P_4, if P_3 sends $[m'_X]_1 + D$ where $D \neq 0 \mod 2^k$, S sets $\mathsf{abort}_1 = 1$.

- $\widetilde{h}_1$: S computes $d_X = m'_X + \lambda_X$ and sends $\widetilde{h}_1 = H(d_X)$ to P_3 to simulate the message P_3 receives from P_1 in Step 9. If $\mathsf{abort}_1 = 1$, S aborts and outputs the partial view. If $\mathsf{abort}_1 = 0$, S observes the message $\hat{h}_1$ that P_1 sends to P_3. If $\hat{h}_1 \neq \widetilde{h}_1$, S also aborts and outputs the partial view. Else, S outputs whatever P_3 outputs.

Claim 4. *For the simulator S corrupting party P_3 as described above, and interacting with the functionality $\mathcal{F}_{[x]\to m_x}$,*

$$\left\{\textsc{hybrid}_{\pi_{[x]\to m_x},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad \stackrel{c}{\equiv} \quad \left\{\textsc{ideal}_{\mathcal{F}_{[x]\to m_x},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof:

Case 0: First, we consider the case where P_3 follows the protocol honestly. In this case, the joint distribution of the view of P_1 and the output in the hybrid and ideal execution is:

$$\left\{\textsc{hybrid}_{\pi_{[x]\to m_x},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} =$$

$$\{[\lambda'_X]_1, [\lambda_X]_1, [\lambda_X]_2, [m'_X]_1, [Y']_1, b, [m'_X]_2, h_1, o_1, o_2, o_3, o_4\}$$

$$\left\{\textsc{ideal}_{\mathcal{F}_{[x]\to m_x},S(z)}(X,Y,\kappa)\right\} -_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} =$$

$$\{[\widetilde{\lambda'_X}]_1, [\widetilde{\lambda_X}]_1, [\widetilde{\lambda_X}]_2, [\widetilde{m'_X}]_1, [\widetilde{Y}']_1, \widetilde{b}, [\widetilde{m}'_X]_2, \widetilde{h}_1, \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3, \widetilde{o}_4\}$$

According to the simulation, $b = \widetilde{b} = \mathsf{true}$ and $h_1 = \widetilde{h}_1$ as d_X is uniquely determined

from the received output. $[\lambda'_X]_1$ and $[\widetilde{\lambda}'_X]_1$ are identical as both are provided by the ideal functionality. $[\lambda_X]_i$ and $[\widetilde{\lambda_X}]_i$ are all distributed uniformly at random, and satisfy the condition that $[\lambda_X]_1 + [\lambda_X]_2 = [\widetilde{\lambda}_X]_1 + [\widetilde{\lambda}_X]_2$. The same applies for shares of m'_X. Thus, the joint distributions in both worlds are identical.

Case 1: If P_3 sends $[Z]+E$ for $E \neq 0 \mod 2^{k+s}$ to $\mathcal{F}_{\mathsf{CheckZero}}$ in Step 8, in the ideal world, S hands P_3 $\widetilde{b} = \mathsf{false}$, aborts, and output the partial view. In the hybrid world, execution also ends in abort with $\mathcal{F}_{\mathsf{CheckZero}}$ outputs $b = 0$. Clearly, the joint distributions in both worlds are identical.

Case 2: If P_3 cheats in Step 8 by opening $m'_X + D$ with $D \neq 0 \mod 2^k$ in stead of m'_X. In Step 9, P_2 and P_4 will complain when they compare the hashes and call abort. Thus, in both worlds, the executions end up in abort. The joint distributions in the ideal and hybrid world are also identical.

Case 3: P_3 is honest up to Step 9. The only way that P_3 deviates from the protocol is to send the wrong h_1 to P_1, causing all parties to abort. In this case, the joint distributions in both worlds are also identical.

In conclusion, the joint distributions in both worlds are statistically close.

■

4.6.3 Mask-Share Conversion

Mask-Share conversion is needed after the four-party mask evaluation step. The output of the mask evaluation is masked values and needs to be converted to additive shares again before updating the edges. This conversion step is very simple, and is done without any interaction. To make everything modular, we also provide the ideal functionality and the protocol box for it in Figure 4.8 and 4.9.

FUNCTIONALITY $\mathcal{F}_{m_x \to [x]}$

Inputs: P_1 and P_2 provide $X + \lambda_X$ and $[\lambda_X]$.
Functionality:

- Wait for input from P_1 and P_2. Reconstruct λ and compute X.

Output: Send $[X]$ to P_1 and P_2.

Figure 4.8: Ideal Functionality to convert Masked Values To Additive Shares

PROTOCOL $\Pi_{m_x \to [x]}$

Inputs: P_1 and P_2 have $X + \lambda_X$ and $[\lambda_X]$.
Protocol:

- P_1 and P_2 computes $[X] \leftarrow (X + \lambda_X) - [\lambda_X]$

Output: P_1 and P_2 output $[X]$

Figure 4.9: Real-world protocol to convert Masked Values To Additive Shares

4.6.4 Four-Party Mask Evaluation With Truncation

This section presents the small sub-components that are utilized in the Apply operation.

Fixed point arithmetic: A fixed point number is represented by an element of the ring $\mathbb{Z}_{2^k}$. The d least significant bits are used for fractional part of the number. We provide a way to perform multiplication with masked value on fixed point numbers.

Masked value: In our protocol, we use masked values for the computation. Instead of holding shares $[x]$, one group has ($m_x = x + \lambda_x$, λ'_x, $[\lambda_x]$) and the other has ($m'_x = x + \lambda'_x$, λ_x, $[\lambda'_x]$).

Addition: Addition is performed locally by adding the masked values together.

For P_1 and P_2: $(m_x, \lambda'_x, [\lambda_x]) + (m_y, \lambda'_y, [\lambda_y]) = (m_x + m_y, \lambda'_x + \lambda'_y, [\lambda_x] + [\lambda_y])$.

For P_3 and P_4: $(m'_x, \lambda_x, [\lambda'_x]) + (m'_y, \lambda_y, [\lambda'_y]) = (m'_x + m'_y, \lambda_x + \lambda_y, [\lambda'_x] + [\lambda'_y])$.

Multiplication Without Truncation: Assume that P_1 and P_2 want to perform a secure multiplication on the mask values $(x + \lambda_x)$ and $(y + \lambda_y)$, and the desired output is $(xy + \lambda_z, \lambda'_z, [\lambda_z])$. P_1 and P_2 hold secret shares $[\lambda_x]$, $[\lambda_y]$, and $[\lambda_x\lambda_y + \lambda_z]$. These shares are provided by P_3 and P_4.

Locally P_1 and P_2 compute

P_1: $[m_z]_1 = m_x m_y - [\lambda_x]m_y - [\lambda_y]m_x + [\lambda_z + \lambda_x\lambda_y]$.

P_2: $[m_z]_2 = -[\lambda_x]m_y - [\lambda_y]m_x + [\lambda_z + \lambda_x\lambda_y]$.

and exchange the shares to reconstruct $m_z = xy + \lambda_z$. They output $(m_z, \lambda'_z, [\lambda_z])$.

Similarly, P_3 and P_4 output $(m'_z, \lambda_z, [\lambda'_z])$.

Multiplication With Truncation: In our setting, x and y are fixed-point numbers with d bits for the fraction. The result of the multiplication is a number that has its least $2d$ significant bits in the fractional portion. A truncation is needed to throw away the d least significant bits: the output of the multiplication is the masked value of $\lfloor \frac{xy}{2^d} \rfloor$ in stead of that of xy. We provide a method to handle the truncation for our four party mask evaluation. First, we have a simple observation:

$\lfloor \frac{xy+\lambda_z+\lambda'_z}{2^d} \rfloor = \lfloor \frac{xy+\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \epsilon_1$

$= \lfloor \frac{xy}{2^d} \rfloor + \lfloor \frac{\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \epsilon_1 + \epsilon_2$, where $\epsilon_i \in \{0, 1\}$.

We denote:

$m_z = \lfloor \frac{xy+\lambda_z+\lambda'_z}{2^d} \rfloor - \lfloor \frac{\lambda'_z}{2^d} \rfloor = (\lfloor \frac{xy}{2^d} \rfloor + \epsilon_1 + \epsilon_2) + \lfloor \frac{\lambda_z}{2^d} \rfloor$

$m'_z = \lfloor \frac{xy+\lambda_z+\lambda'_z}{2^d} \rfloor - \lfloor \frac{\lambda_z}{2^d} \rfloor = (\lfloor \frac{xy}{2^d} \rfloor + \epsilon_1 + \epsilon_2) + \lfloor \frac{\lambda'_z}{2^d} \rfloor$

where $\epsilon_i \in \{0, 1\}$. The error due to the truncation is $\epsilon = \epsilon_1 + \epsilon_2 \in \{0, 1, 2\}$.

P_1 and P_2 can compute m_z and $\lfloor \frac{\lambda'_z}{2^d} \rfloor$ themselves without any interaction as they know $xy + \lambda_z$ and λ'_z. P_3 and P_4 can provide P_1 and P_2 with shares $[\lfloor \frac{\lambda_z}{2^d} \rfloor]$. At the end, P_1 and P_2 obtain the output of the truncated mask evaluation: $(m_z, \lfloor \frac{\lambda'_z}{2^d} \rfloor, [\lfloor \frac{\lambda_z}{2^d} \rfloor])$. Similarly, P_3 and P_4

obtain $(m'_z, \lfloor \frac{\lambda_z}{2^d} \rfloor, [\lfloor \frac{\lambda'_z}{2^d} \rfloor])$. The error of the truncated multiplication is at most $\frac{1}{2^{d-1}}$.

Vectorization for dot products: A naive way to perform a dot product between two vectors $\boldsymbol{u} = \{u_1, ..., u_n\}, \boldsymbol{v} = \{v_1, ..., v_n\}$ is to perform n multiplications then add the shares up. We use the vectorization technique to bring this down to the cost of one multiplication. The details are shown in Figure 4.11.

FUNCTIONALITY $\mathcal{F}_{\text{eval}}$

Inputs: For each input wire w: P_1, P_2 provide $m_w = x_w + \lambda_w$, $[\lambda_w]$, and λ'_w; P_3, P_4 hold $m'_w = x_w + \lambda'_w$, $[\lambda'_w]$, and λ_w.
Functionality:

- Wait for inputs from all parties. Reconstruct λ received from P_1, P_2, and verifies if it is equal to λ received from P_3, P_4. Do the same verification for λ. If any of the verification fails, send abort to all parties.
- Compute
  - $(m_w^{(1)}, \lambda_w'^{(1)}, [\lambda_w^{(1)}]) \leftarrow func\,(m_w, \lambda'_w, [\lambda_w])$
  - $(m_w'^{(1)}, \lambda_w^{(1)}, [\lambda_w'^{(1)}]) \leftarrow func\,(m'_w, \lambda_w, [\lambda'_w])$

Output: P_1, P_2 receive $(m_w^{(1)}, \lambda_w'^{(1)}, [\lambda_w^{(1)}])$. P_3, P_4 receive $(m_w'^{(1)}, \lambda_w^{(1)}, [\lambda_w'^{(1)}])$.

Figure 4.10: Ideal Functionality to handle Masked Evaluation With Truncation

Theorem 5. *The protocol Π_{eval} (Figure 4.11) securely realizes the ideal functionality $\mathcal{F}_{eval}$ (Figure 4.10) with abort, under a single malicious corruption.*

4.6.5 Assumed Ideal Functionalities

We assume that we have access to the following oracles: $\mathcal{F}_{\mathsf{coin}}$ (Figure 4.12), $\mathcal{F}_{\mathsf{checkZero}}$ (Figure 4.13), $\mathcal{F}_{\mathsf{Triple}}$ (Figure 4.15).

Theorem 6. *The protocol Π_{Mult} (Figure 4.17) securely realizes the ideal functionality $\mathcal{F}_{Mult}$ (Figure 4.16) with abort, under a single malicious corruption.*

The simulation for P_1 and P_2 is identical, and is straight forward as they only make calls to $\mathcal{F}_{\text{coin}}$ to sample random Beaver triplets and send the shares of the triplets to P_3 and P_4. If any of them deviate from the protocol, P_3 and P_4 will catch it by comparing the messages sent from P_1 and those from P_2.

For P_3 and P_4, once they receive the shares of the Beaver triplets, they follow the same known procedure to perform the multiplication. A corrupted party can cheat by adding additive terms u when they open $(x - \lambda_x)$, causing them to compute shares of $\alpha(x + u)$ instead of those of αx.

4.7 Secure GAS Model of Computation and its Oblivious Graph Operations

In this section, we explain the details of each graph operation, present their real vs. ideal world functionalities, and provide the security proofs for each of them, under a single malicious corruption.

4.7.1 Four-Party Oblivious Shuffle

Shuffle is one of the main operations in our framework that helps to hide the edge structure of the graph. During the Gather and Scatter operations, the data on each side of the edge is being accessed, therefore by shuffling the edges between these two phases, we can hide the connection between neighboring nodes. Another benefit of the Shuffle operation is obscuring the dummy edges into the real ones.

Theorem 7. *The oblivious* Shuffle *protocol* Π_{shuffle} *(Figure 4.19) securely realizes the ideal functionality* $\mathcal{F}_{\text{shuffle}}$ *(Figure 4.18) with abort, under a single malicious corruption.*

To prove the security of our Oblivious Shuffle, we provide a simulation for P_1 and P_3. The simulations for other parties are identical.

First, a simulation for P_1:

- $\widetilde{\alpha}$: S samples a random $\widetilde{\alpha}$ and hands it to P_1 to simulate the output P_1 receives from $\mathcal{F}_{\mathsf{coin}}$. S receives P_1's input $[E]_1$ from the distinguisher and places it in the input tape of P_1. S observes the message that P_1 sends to $\mathcal{F}_{\mathsf{MAC}}$: if P_1 does not send the intended messages $(\widetilde{\alpha}, [E]_1)$, S submits abort to $\mathcal{F}_{\mathsf{shuffle}}$, and outputs the partial transcript. Else, S submits P_1's input $[E]$ to the ideal functionality $\mathcal{F}_{\mathsf{shuffle}}$ and receives $[E^{(1)}]$.

- $[\widetilde{E}^{(1)}], [\widetilde{M}^{(1)}]$: S samples random ring elements as shares $[\widetilde{M}^{(1)}]$, hands $[\widetilde{E}^{(1)}]$ (where $[\widetilde{E}^{(1)}] \equiv [E^1]$) and $[\widetilde{M}^{(1)}]$ to P_1 to simulate the messages $[E^{(1)}], [M^{(1)}]$ P_1 receives from $\mathcal{F}_{\mathsf{MAC}}$. S computes $[Z]$ himself to mirror P_1's action.

- $\widetilde{b}$: S observes the messages that P_1 sends to $\mathcal{F}_{\mathsf{checkZero}}$. If P_1 modifies his shares $[Z]$, S hands $\widetilde{b} = \mathsf{false}$ to P_1 as the output of $\mathcal{F}_{\mathsf{CheckZero}}$, output the partial view, and abort. Else, S hands $\widetilde{b} = \mathsf{true}$ to P_1 and output whatever P_1 outputs.

Claim 5. *For the simulator S corrupting party P_1 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{shuffle}}$,*

$$\left\{\textsc{hybrid}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}\left(E, M, \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad \stackrel{c}{\equiv} \quad \left\{\textsc{ideal}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E, M, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof: Case 0: If P_1 follows the protocol honestly, the joint distributions in the hybrid and ideal execution is:

$$\left\{\textsc{hybrid}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}\left(E, M, \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\alpha, [E^{(1)}], [M^{(1)}], b = \mathsf{true}, o_1, o_2, o_3, o_4\}$$

$$\left\{\textsc{ideal}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E, M, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{\alpha}, [\widetilde{E}^{(1)}], [\widetilde{M}^{(1)}], \widetilde{b} = \mathsf{true}, \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3, \widetilde{o}_4\}$$

The messages $[\alpha]$, $[\widetilde{\alpha}]$, $[M^{(1)}]$ and $[\widetilde{M}^{(1)}]$ are all uniformly distributed, $[E^{(1)}]$ and $[\widetilde{E}^{(1)}]$

are identical, thus, the joint distributions between both worlds are identical.

Case 1: If P_1 deviates from the protocol in Step 2 by providing the incorrect α or incorrect shares $[E]$ to $\mathcal{F}_{\mathsf{MAC}}$, abort happens in both worlds, and the joint distributions in both worlds are $\{\alpha, \perp\}$ and $\{\widetilde{\alpha}, \perp\}$ respectively, and they are identically distributed.

Case 2: If P_1 deviates from the protocol in Step 4 by providing the wrong shares $[Z]$ to $\mathcal{F}_{\mathsf{checkZero}}$, S hands $\widetilde{b} = \mathsf{false}$ to P_1 in the ideal world and aborts. In the hybrid world, $\mathcal{F}_{\mathsf{CheckZero}}$ outputs $b = \mathsf{false}$ and all parties abort. It is clear that the joint distributions in both worlds are identical.

In conclusion, the joint distributions between the two worlds are identical. ■

Now, a simulation for P_3:

- $[\widetilde{Y}]$: S receives $[E]$ and places it in the input tape of P_3. S observes the message that P_3 sends to $\mathcal{F}_{\mathsf{MAC}}$: if P_3 modifies $[E]$ before sending it to the functionality, S aborts and outputs the partial view. Else, S samples random ring elements as shares $[\widetilde{M}]$ and hands them to P_3 to simulate the message P_3 receives from $\mathcal{F}_{\mathsf{MAC}}$ in the hybrid world.

- $\widetilde{\pi}$: S queries the ideal functionality with P_3's input, $[E]$, and obtains the output $[E^{(1)}]$. S computes $\widetilde{\pi}$ such that $\pi([E]) = [E^{(1)}]$, then agrees on the permutation $\widetilde{\pi}$ with P_3 in Step 3. S computes $[m^{(1)}] \leftarrow [\widetilde{\pi}(\widetilde{m})]$ to mirror P_3's action.

- $\widetilde{b}$: S observes the messages that P_3 sends to P_1 in Step 3. If P_3 sends $[E'^{(1)}] = [E^{(1)} + D]$ or $[m'^{(1)}] = [m^{(1)} + E]$ where $D \neq 0 \mod 2^k, E \neq 0 \mod 2^{k+s}$, S aborts and outputs the partial view. Else, S outputs whatever P_3 outputs.

Claim 6. *For the simulator S corrupting party P_3 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{shuffle}}$,*

$$\left\{\textsc{hybrid}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}\left(E, M, \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad \stackrel{c}{\equiv} \quad \left\{\textsc{ideal}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E, M, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof: Case 0: If P_3 follows the protocol honestly, the joint distributions in the hybrid and ideal execution is:

$$\left\{\text{HYBRID}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}\left(E,M,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad = \quad \{[M],\pi,b,o_1,o_2,o_3,o_4\}$$

$$\left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E,M,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad = \quad \{[\widetilde{M}],\widetilde{\pi},\widetilde{b},\widetilde{o}_1,\widetilde{o}_2,\widetilde{o}_3,\widetilde{o}_4\}$$

The messages $[M],[\widetilde{M}]$ are distributed uniformly at random and independent from one another. π and $\widetilde{\pi}$ are identical. Thus, the joint distributions between both worlds are identical.

Case 1: If P_3 deviates from the protocol in Step 2 by sending the wrong shares $[E]$, abort happens in both worlds, and the joint distributions in both worlds are both $\{\perp\}$ and identical.

Case 2: S observes what P_3 sends to P_1 in Step 3. If he does not send the intended messages: P_3 sends $[E'^{(1)}]=[E^{(1)}+D]$ or $[M'^{(1)}]=[M^{(1)}+D']$ where $D\neq 0 \mod 2^k, D'\neq 0 \mod 2^{k+s}$, S abort in the ideal execution. The joint distribution in the ideal world is $\{[\widetilde{M}],\widetilde{\pi},\widetilde{b}=\mathsf{false},\perp\}$. In the hybrid world, there is a small chance that P_1 and P_2 do not abort. This happens if P_3 chooses the additive terms D and E such that $\alpha D+D'=0 \mod 2^{k+s}$. The probability that this happen is at most 2^{-s} as shown in Section 4.6.1. So, with probability $1-2^{-s}$, the joint distribution in the hybrid world is $\{[M],[\pi],b=\mathsf{false},\perp\}$. Thus, the joint distributions in both worlds are statistically close.

In conclusion, the joint distributions in both worlds are statistically close. ■

4.7.2 Four-Party Oblivious Gather

Theorem 8. *The oblivious* Gather *protocol (Figure 4.21) securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{gather}}$ *(Figure 4.20) with abort, under a single malicious corruption.*

We provide a simulation for P_1 and P_3. The simulations for others are almost identical. First, a simulation for P_1:

1. S obtains P_1's input from the distinguisher and puts it in the input tape of P_1. S submits the input, $[E]$, to the ideal functionality and obtains the output $[\widetilde{V}]$ and $[\widetilde{W}]$.

2. $[\widetilde{J}]$ (where $\widetilde{J} \equiv \beta E.l_{id}$): S hands P_1 random ring elements to simulate the output of $\mathcal{F}_{\mathsf{MAC}}$ on $[E.l_{id}]$.

3. $[\widetilde{R}] \equiv [\beta E.r_{\mathsf{data}}], \widetilde{I} \equiv \{\widetilde{e}_1.l_{id}, ..., \widetilde{e}_{|E|}.l_{id}\}$: S samples a random permutation π and compute $\widetilde{I} \leftarrow \pi(\{id_1, ..., id_1, ..., id_n, ..., id_n\}), [\{id_{1_1}..id_{1_i}\}, ..., \{id_{n_1}..id_{n_j}\}]$ in which the number of id_i's is the in-degree of the vertex that has the left id to be id_i. S uses $\widetilde{I}$ and $\widetilde{W}$ to generate $[\widetilde{R}]$, the MAC of the right data, in correct order. S hands $[\widetilde{R}]$ to P_1 to simulate the corresponding message that P_1 receives from $\mathcal{F}_{\mathsf{MAC}}$. To simulate the opening of an edge, S uses $\widetilde{I}$ and the input $[E]$ to compute the share it needs to send to P_1, opening the edge to the correct value. S also observes the shares that P_1 sends to P_2. If P_2 cheats by not sending the intended shares, S sets $\mathsf{abort}_1 = 1$. Also, to simulate the movement of the data, if the right of an edge e is to be moved to the vertex v, and it's the j^{th} one to be moved to v, S sends the j^{th} shares of $[\widetilde{V}]$ and $[\widetilde{W}]$ to P_1.

4. S observes the messages that P_1 sends to P_3, I and $[\widetilde{J}]$. If P_1 modifies any of these messages, or if $\mathsf{abort}_1 = 1$, S aborts and outputs the partial view.

Proof: We give a proof sketch here.

First, S can perfectly simulate the output of $\mathcal{F}_{\mathsf{MAC}}$ as the shares of $[J]$ in the hybrid world are also uniformly distributed. From P_1's input, $[E]$ and the output of the ideal functionality $[\widetilde{V}], [\widetilde{W}]$, S can perfectly simulate the opening of the indices and moving of the data. The sequence of opened indices in the ideal world and that in the hybrid world is identically distributed as $[E]$ has been randomly shuffled prior to this step, so any sequence happens with equal probability.

Second, if P_1 deviates from the protocol at any step, the executions in both worlds end up in abort, the the partial views are identically distributed.

In conclusion, the joint distributions in both worlds are identical. ■

Now we provide a simulation for P_3:

1. $\widetilde{I}, [\widetilde{J}]$: S samples a random permutation π and compute $\widetilde{I} \leftarrow \pi(\{id_1, ..., id_1, ..., id_n, ..., id_n\})$, in which the number of id_i's is the in degree of the vertex that has the left id to be id_i. S hands $\widetilde{I}$ and random values as shares $[\widetilde{J}]$ to P_3 to simulate the messages that P_3 receives from P_1.

2. $\widetilde{b}$: S observes the messages that P_3 sends to $\mathcal{F}_{\mathsf{verifyMAC}}$. If P_3 does not send the intended messages, S hands P_3 $\widetilde{b} = \mathsf{false}$, aborts and outputs the partial view. Else, S hands P_3 $\widetilde{b} = \mathsf{true}$, and output whatever P_3 outputs.

Proof: We give a proof sketch here.

Similar to the argument in the previous proof, any sequence of open indices are equally likely to happen, thus, a random sequence in the ideal world and the actually sequence in the hybrid world is identically likely to happen. Also, the shares $[\widetilde{J}]$ and $[J]$ are uniformly distributed. Thus, if P_3 honest in the MAC verification step, the joint distributions in both worlds are identical. If P_3 cheats, the executions in both worlds end up in abort, and the joint distributions are also identical.

In conclusion, the joint distributions in both worlds are identical. ■

4.7.3 **Four-Party Oblivious** Apply

Apply computes the main functionality of the framework on the input data. During the Gather the data is aggregated into vertices, therefore Apply runs the computation on the vertices data.

Theorem 9. *The oblivious Apply protocol* Π_{apply} *(Figure 4.23) securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{apply}}$ *(Figure 4.22) with abort, under a single malicious corruption.*

To prove the security of our Oblivious Apply, we provide a simulation for P_1. The simulations for other parties are identical.

Simulation for P_1:

- $(\widetilde{m}_V, \widetilde{\lambda'}_V, [\widetilde{\lambda}_V])$: S receives P_1's inputs $[\beta]$, $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\}$, ... , $\{W_{n_1}..W_{n_j}\}]$, and places them in the input tape of P_1. S observes the messages that P_1 sends to $\mathcal{F}_{\mathsf{sharemask}}$: if P_1 does not send the intended messages $([V], [W], [\beta])$, S submits abort to $\mathcal{F}_{\mathsf{apply}}$, and outputs the partial transcript. Else, S samples random ring elements as shares $(\widetilde{m}_V, \widetilde{\lambda}_V, [\widetilde{\lambda}_V])$ and hands them to P_1 to simulate the message P_1 receives from $\mathcal{F}_{\mathsf{sharemask}}$ in the hybrid world.

- $(\widetilde{m}_V^{(1)}, \widetilde{\lambda'}_V^{(1)}, [\widetilde{\lambda}_V^{(1)}])$: S observes the messages that P_1 sends to $\mathcal{F}_{\mathsf{eval}}$: if P_1 does not send the intended messages $(m_V, \lambda'_V, [\lambda_V])$, S submits abort to $\mathcal{F}_{\mathsf{apply}}$, and outputs the partial transcript. Else, S samples random ring elements as shares $(\widetilde{m}_V^{(1)}, \widetilde{\lambda'}_V^{(1)}, [\widetilde{\lambda}_V^{(1)}])$ and hands them to P_1 to simulate the messages P_1 receives from $\mathcal{F}_{\mathsf{apply}}$ in the hybrid world.

- $[\{\widetilde{V}_{1_1}^{(1)}..\widetilde{V}_{1_i}^{(1)}\}, ..., \{\widetilde{V}_{n_1}^{(1)}..\widetilde{V}_{n_j}^{(1)}\}]$: S observes the messages that P_1 sends to $\mathcal{F}_{\mathsf{maskshare}}$: if P_1 does not send the intended messages $(m_V^{(1)}, \lambda_V'^{(1)}, [\lambda_V^{(1)}])$, S submits abort to $\mathcal{F}_{\mathsf{apply}}$, and outputs the partial transcript. Else, S submits P_1's input $[V], [W], [\beta]$ to the ideal

functionality and receives $[\{V^{(1)}{}_{1}..V^{(1)}{}_{1_i}\}, ..., \{V^{(1)}{}_{n_1}..V^{(1)}{}_{n_j}\}]$, and hands them to P_1 to simulate the messages P_1 receives from $\mathcal{F}_{\mathsf{maskshare}}$ in the hybrid world.

Claim 7. *For the simulator S corrupting party P_1 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{apply}}$,*

$$\left\{\text{HYBRID}_{\pi_{\mathsf{apply}},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad \stackrel{c}{\equiv} \quad \left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{apply}},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof:

Case 0: If P_1 follows the protocol honestly, the joint distributions in the hybrid and ideal execution is:

$$\left\{\text{HYBRID}_{\pi_{\mathsf{apply}},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{m_V, \lambda'_V, [\lambda_V], m_V^{(1)}, \lambda_V'^{(1)}, [\lambda_V^{(1)}], [V^{(1)}], o_1, o_2, o_3, o_4\}$$

$$\left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{apply}},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{m}_V, \widetilde{\lambda}'_V, [\widetilde{\lambda}_V], \widetilde{m}_V^{(1)}, \widetilde{\lambda}'^{(1)}_V, [\widetilde{\lambda}_V^{(1)}], [\widetilde{V}^{(1)}], \widetilde{o}_1, \widetilde{o}_2, \widetilde{o}_3, \widetilde{o}_4\}$$

The messages m_V and $\widetilde{m}_V$, λ'_V and $\widetilde{\lambda}'_V$, $[\lambda_V]$ and $[\widetilde{\lambda}_V]$, $m_V^{(1)}$ and $\widetilde{m}_V^{(1)}$, $\lambda_V'^{(1)}$ and $\widetilde{\lambda}'^{(1)}_V$ are all uniformly distributed, $[V^{(1)}]$ and $[\widetilde{V}^{(1)}]$ are identical, thus, the joint distributions between both worlds are identical.

Case 1: If P_1 deviates from the protocol in any step, abort happens in both worlds, and partial view of P_1 up to that point in both worlds are identically distributed.

In conclusion, the joint distributions between the two worlds are identical. ■

4.7.4 Four-Party Oblivious Scatter

During the Oblivious Scatter, the updated data in the vertices pushed back to their corresponding edges in the graph, and replace the old profile values in the edges. This step is

done locally by each party, P_1 and P_2, with no interaction between them. Therefore, this step is secure. If any of the parties cheats and modifies the data before scattering to the edges, it will be detected in the following phase, which is Shuffle operation in the next round.

4.7.5 Four-Party Secure GAS model of computation (the overall protocol)

In this section, we prove the security of the overall framework, where we refer to it as sgas framework. But before them main proof, first we provide a description of the leakage function $\mathcal{L}(G)$ in the following theorem:

Theorem 10. *The randomized algorithm $\mathcal{L}$ is (ϵ, δ)-approximate differentially private, defined as following:*

A randomized algorithm $\mathcal{L} : \mathcal{G} \rightarrow \mathcal{R}_{\mathcal{L}}$ is (ϵ, δ)-edge private if for all neighboring graphs, $G_1, G_2 \in \mathcal{G}$, we have:

$$\Pr[\mathcal{L}(G_1) \in T] \leq e^{\epsilon} \Pr[\mathcal{L}(G_2) \in T] + \delta$$

Theorem 11. *The protocol Π_{sgas} (Figure 4.1) securely computes the ideal functionality $\mathcal{F}_{\mathsf{sgas}}$ with $\mathcal{L}$ leakage in the*

$(\mathcal{F}_{\mathsf{shuffle}}, \mathcal{F}_{\mathsf{gather}}, \mathcal{F}_{\mathsf{apply}}, \mathcal{F}_{\mathsf{scatter}})$-hybrid model with abort, under a single malicious corruption.

To prove the security of our overall framework sgas, we provide a simulation for P_1 and P_3. The simulations for other parties are identical.

Simulation for P_1:

- S receives P_1's input $[E]$ from the distinguisher and places it on the input tape of P_1. S submits P_1's input $[E]$ to the ideal functionality $\mathcal{F}_{\mathsf{sgas}}$ and receives $[E^{(2)}]$, $\mathcal{L}(G)$, where $\mathcal{L}(G)$ (Theorem 10) is the leakage function that indicates the noisy degree of each vertex in the graph.

- $[\widetilde{E}^{(1)}]$: S observes the message that P_1 sends to $\mathcal{F}_{\mathsf{shuffle}}$: if P_1 does not send the intended messages $[E]$, S submits abort to $\mathcal{F}_{\mathsf{sgas}}$, and outputs the partial transcript. Else, S

simulates the output of $\mathcal{F}_{\mathsf{shuffle}}$ by sampling random ring elements as shares of $[\widetilde{E}^{(1)}]$, and handing them to P_1 to simulate the output of $\mathcal{F}_{\mathsf{shuffle}}$.

- $[\{\widetilde{V}_{1_1}..\widetilde{V}_{1_i}\}, ..., \{\widetilde{V}_{n_1}..\widetilde{V}_{n_j}\}]$, $[\{\widetilde{W}_{1_1}..\widetilde{W}_{1_i}\}, ..., \{\widetilde{W}_{n_1}..\widetilde{W}_{n_j}\}]$, $[\widetilde{\beta}]$: S observes the message that P_1 sends to $\mathcal{F}_{\mathsf{gather}}$: if P_1 does not send the intended messages $[E^{(1)}]$, S submits abort to $\mathcal{F}_{\mathsf{sgas}}$, and outputs the partial transcript. Else, S uses the leakage function $\mathcal{L}(G)$ to sample random ring elements as shares $[\{\widetilde{V}_{1_1}..\widetilde{V}_{1_i}\}, ..., \{\widetilde{V}_{n_1}..\widetilde{V}_{n_j}\}]$ and $[\{\widetilde{W}_{1_1}..\widetilde{W}_{1_i}\}, ..., \{\widetilde{W}_{n_1}..\widetilde{W}_{n_j}\}]$, and also sample a random share of $[\widetilde{\beta}]$, and hands them to P_1 to simulate the output P_1 receives from $\mathcal{F}_{\mathsf{gather}}$ in the hybrid world.

- $[\{\widetilde{V}^{(1)}_{1_1}..\widetilde{V}^{(1)}_{1_i}\}, ... , \{\widetilde{V}^{(1)}_{n_1}..\widetilde{V}^{(1)}_{n_j}\}]$: S observes the message that P_1 sends to $\mathcal{F}_{\mathsf{apply}}$: if P_1 does not send the intended messages $[\beta]$, $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\}$, ... , $\{W_{n_1}..W_{n_j}\}]$, S submits abort to $\mathcal{F}_{\mathsf{sgas}}$, and outputs the partial transcript. Else, S samples random ring elements as shares $[\{\widetilde{V}^{(1)}_{1_1}..\widetilde{V}^{(1)}_{1_i}\}, ..., \{\widetilde{V}^{(1)}_{n_1}..\widetilde{V}^{(1)}_{n_j}\}]$, and hands them to P_1 to simulate the output P_1 receives from $\mathcal{F}_{\mathsf{apply}}$ in the hybrid world.

- $[\widetilde{E}^{(2)}]$: S observes the message that P_1 sends to $\mathcal{F}_{\mathsf{scatter}}$: if P_1 does not send the intended messages $[\{V^{(1)}_{1_1}..V^{(1)}_{1_i}\}, ..., \{V^{(1)}_{n_1}..V^{(1)}_{n_j}\}]$, S submits abort to $\mathcal{F}_{\mathsf{sgas}}$, and outputs the partial transcript. Else, S hands $[\widetilde{E}^{(2)}]$ (where $[\widetilde{E}^{(2)}] \equiv [E^2]$) to P_1 to simulate the output P_1 receives from $\mathcal{F}_{\mathsf{scatter}}$ in the hybrid world and then outputs whatever P_1 outputs.

Claim 8. *For the simulator S corrupting party P_1 as described above, and interacting with the functionality $\mathcal{F}_{\mathsf{sgas}}$,*

$$\left\{\textsc{hybrid}_{\pi_{sgas},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad \stackrel{c}{\equiv} \quad \left\{\textsc{ideal}_{\mathcal{F}_{sgas},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Proof: Case 0: If P_1 follows the protocol honestly, the joint distributions in the hybrid and

ideal execution is:

$$\left\{\text{HYBRID}_{\pi_{\text{sgas}},\mathcal{A}(z)}\left(X,Y,\kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{[E^{(1)}],[\beta],[V],[W],[V^{(1)}],[E^{(2)}],o_1,o_2,o_3,o_4\}$$

$$\left\{\text{IDEAL}_{\mathcal{F}_{\text{sgas}},S(z)}(X,Y,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} = \{\widetilde{E}^{(1)}],[\widetilde{\beta}],[\widetilde{V}],[\widetilde{W}],\{\widetilde{V}^{(1)}],[\widetilde{E}^{(2)}],\widetilde{o}_1,\widetilde{o}_2,\widetilde{o}_3,\widetilde{o}_4\}$$

The messages $\{[E^{(1)}],[\widetilde{E}^{(1)}]$, $[\beta],[\widetilde{\beta}]$, $[V],[\widetilde{V}]$, $[W],[\widetilde{W}]$, $\{[V^{(1)}],[\widetilde{V}^{(1)}]$ are distributed uniformly at random and independent from one another. $\{[E^{(2)}],[\widetilde{E}^{(2)}]$ are identical. Thus, the joint distributions between both worlds are identical.

Case 1: As soon as P_1 deviates in any step of the protocol, abort happens in both worlds, and partial view of P_1 up to that point in both worlds are identically distributed. ■

4.8 Implementation and Evaluation

We implemented our four-party secure computation framework in C++. We measure the performance of our framework on a set of benchmark algorithms in order to evaluate our design. These benchmarks consist of histogram and matrix factorization problems which are commonly used for evaluating highly-parallelizable frameworks. In all scenarios, we assume that the data is secret-shared across four non-colluding cloud providers, as motivated in Section 1. For comparison, we compare our results with the closest large-scale secure parallel graph computation schemes, such as GraphSC [11] and OblivGraph [47].

4.8.1 Implementation

Using our four-party framework, the histogram and matrix factorization problems can be represented as directed bipartite graphs.

Histogram: In the histogram computation, which, for example, might be used to count the number of people in each zip code, left vertices represent data elements (people), right vertices are the counters for each type of data element (the zip code), and existence of an edge indicates that data element on the left has the data type of the right node (e.g. the user on the left belong to the zip code on the right).

Matrix Factorization: In matrix factorization, left vertices represent users, right vertices are items (e.g. movies in movie recommendation systems or a product in targeted advertising systems), an edge indicates that a user ranked that item, and the weight of the edge represents the rating value.

Vertex and Edge representation: In all scenarios, our statistical security parameter $s = 40$. We chose $k = 40$ to represent k-bit fixed-point numbers, in which the least d significant bits are used for the fractional part. For histogram $d = 0$ and for matrix factorization $d = 20$. This requires data and MACs to be secret share in the $Z_{2^{80}}$ ring. In our matrix factorization experiments, we factorized the ratings matrix into two matrices, represented by feature vectors that each has dimension 10. We chose these parameters as to be compatible with the GraphSC and OblivGraph representations.

4.8.2 Evaluation

We run the Histogram experiments on graphs with sizes ranging from 1 million to more than 300 million edges, which can simulate the counting operation in census data gathering. For example, if each user contributed a salary value and a zip-code, using our framework we can compute the average salary in each zip-code, while ensuring that the access patterns preserve user privacy. We run matrix factorization with gradient descent on the real-world MovieLens datasets[44] that contains different numbers of ratings provided by different numbers of users on the movies. We report the result for one complete iteration of the protocol, performing GAS operations one time on both the left and right nodes. The results are the average of five executions of the experiments.

Experiment settings: We run all the experiments on AWS (Amazon Web Services) using four r4.8xlarge instances spread across Northern Virginia and Oregon data centers. Each machine has 32 processors and 244 GiB RAM, with 10 Gbps network connectivity. In our four party protocol, each pair (P_1, P_4) or (P_2, P_3) communicates O(1) ring elements to verify the execution across the groups, thus, it is not necessary for P_1 and P_4 to locate in the same

data center (and similarly for P_2 and P_3).

We use three metrics in evaluating the performance of our framework: running time in seconds, communication cost in number of bits transferred between parties, and circuit size in number of AND Gates/AES operations.

Table 4.1 gives the input parameters used in the experiments. In Table 4.7, we explore the performance impact of using different ϵ and δ privacy parameters.

Table 4.1: The details of input size and privacy parameters for all the different experiment conducted on Histogram and Matrix Factorization

Edges	Users	Items	ϵ	δ
1M	6K	4K	0.3	2^{-40}
10M	72K	10K	0.3	2^{-40}
20M	138K	27K	0.3	2^{-40}
300M	300M	42K	0.3	2^{-40}

Run time and Communication Cost: Figure 4.24 demonstrates that the run time required to perform Histogram (counting) operation on a graph of size 1 million to 300 millions can be less than 3.3 mins, by using our framework running on multiprocessor machines. The values on the graph are also represented in Table 4.2. Figure 4.25 shows the amount of data in MB, transferred between the parties during the Histogram protocol. Communication cost shows linear decrease with increasing the number of processors. Both graphs are in log-log scale.

Similarly, Figure 4.26 shows that computing Matrix Factorization on large scale graph data sets takes less than 5 minutes, using our four-party framework, on AWS multiprocessor machines. The running time is expected to decrease linearly as we increase the number of processors, however due to some small overhead to parallelization, the run time improvement is slightly sub-linear. Table 4.3 shows the results in details. Figure 4.27 shows the communication cost during Matrix Factorization on large data sets. Both graphs are in

Table 4.2: Details of running time (sec) for computing Histogram problem on different input sizes

Processors / Edges	1M	10M	20M	300M
1	13.8	85.0	207.7	2149.4
2	7.5	46.5	98.1	1136.5
4	4.3	28.0	57.78	643.2
8	2.7	16.2	34.39	382.5
16	1.8	11.2	23.3	279.2
32	1.5	10.1	21.67	250.4

log-log scale.

Table 4.3: Details of running time (sec) for computing Matrix Factorization problem on different input sizes

Processors / Edges	1M	10M	20M
1	258.3	1639.7	3401.8
2	132.9	834.7	1913.7
4	80.4	455.57	1055.95
8	44.6	292.2	613.1
16	28.2	190.6	423.7
32	25.1	163.4	357.2

We measured the running time for each of the graph oblivious operation in our framework, to understand the effect of each step in the performance of the framework as a whole. Figure 4.28 and 4.29 demonstrates the run time break-down of each oblivious operation in Histogram and Matrix Factorization problem, on the input graph with only 1 million edges. The oblivious shuffle operation has the highest cost in calculating the Histogram, while Apply phase is taking the most time in Matrix Factorization, due to the calculation of gradient descent values, which are more expensive than counting.

Comparison with previous work: We compare our results with OblivGraph which is the

closest large-scale secure parallel graph computation. OblivGraph used garbled circuits for all the phases of the graph computation, while we use arithmetic circuits. In both approaches, the amount of time needed to send and receive data, and the time spent computing AES, are the dominant costs. We compare the two protocols by the communication cost and the number of AES calls in each of them. In Table 4.4 and 4.5, we demonstrated both the gain in our four party oblivious shuffle against the two party shuffle [45] used in OblivGraph and the gain in the Apply phase with the use of arithmetic circuit in four party setting.

Table 4.4: Estimated AES operations per party for 1 complete matrix factorization iteration: $|E|, |E'|, |V|$ are the number of real edges, number of real and dummy edges, and number of vertices respectively

	OblivGraph	This work
Oblivious Shuffle	$7128(\|E\| \log \|E\| - \|E\| + 1)$	$132\|E'\|$
Oblivious Gather	0	$72\|E'\|$
Share Conversion	-	$72\|E'\| + 30\|V\|$
Oblivious Apply	$279048\|E\| + 4440\|V\|$	$252\|E'\| + 4\|V\|$
Oblivious Scatter	0	$20\|E'\|$
Total	$7128\|E\| \log \|E\| + 271920\|E\| + 4440\|V\| + 7128$	$548\|E'\| + 34\|V\|$

Table 4.6, compares the running time of this work with previous works, GraphSC [11] and OblivGraph [47], to solve matrix factorization problem in scale, with real-world dataset, MovieLens with 6040 users ranked 3883 movies and 1M ratings, and 128 processors.

Effect of differential privacy parameters on the run time: We study the effect of differential privacy parameters on the performance of our framework in Figure 4.30. We also provide the number of dummy edges required for different value of ϵ and δ in Table 4.7. Note that these number of dummy edges are per each right node in the graph, which means an item in Histogram computation, or a movie in Matrix Factorization problem. For example in a movie recommendation system based on our framework, we require 118 dummy

Table 4.5: Estimated total communication cost for all parties in bits for 1 complete matrix factorization iteration: κ is the number of bits per ciphertext, s = 40, $|E|, |E'|, |V|$ are the number of real edges, number of real and dummy edges, and number of vertices respectively. The length of the fixed point numbers used is k = 40 bits

	OblivGraph	This work
Oblivious Shuffle	$4752\kappa(\|E\|\log\|E\| - \|E\| + 1)$	$336(k+s)\|E'\|$
Oblivious Gather	$32\kappa\|E\|$	$142(k+s)\|E'\|$
Share Conversion	-	$192(k+s)\|E'\| + 120(k+s)\|V\|$
Oblivious Apply	$186032\kappa\|E\| + 2960\kappa\|V\|$	$252(k+s)\|E'\| + 16(k+s)\|V\|$
Oblivious Scatter	0	0
Total	$4752\kappa\|E\|\log\|E\| + 181312\kappa\|E\| + 2960\kappa\|V\| + 4752$	$922(k+s)\|E'\| + 126(k+s)\|V\|$

Table 4.6: Running time of a single iteration of this work vs. OblivGraph and GraphSC to solve matrix factorization problem in scale, with real-world dataset, MovieLens with 6040 users ranked 3883 movies and 1M ratings

	GraphSC[11]	**OblivGraph[47]**	**This work**
Time	13 hrs	2 hrs	25 s

edges per movie, to achieve $(0.3, 2^{-40})$-Differential Privacy.

Table 4.7: Number of dummy elements required for each type depending on different privacy parameters

	ϵ=0.05	ϵ=0.3	ϵ=1	ϵ=5
$\delta = 2^{-40}$	707	118	35	7
$\delta = 2^{-16}$	374	62	19	4

LAN vs. WAN runtime All of our experiments so far are conducted on AWS and all four of the computation parties are deployed in the same data center, in Northern Virginia. To demonstrate the effect of network latency on our performance, we deploy two of the computation parties, one from each group, in a data center located in East coast, and the other

two in a data center located in West coast: Alice and David are located in Northern Virginia data center, while Bob and Charlie are deployed in Oregon data center. Figure 4.31 shows a dramatic slowdown in the run time when we deployed the server across data centers as opposed to having them in the same geographic region.

$\pi_{\textbf{eval}}$

Inputs:

1. For each input wire w: P_1, P_2 hold $m_w (= x_w + \lambda_w)$, $[\lambda_w]$, and λ'_w; P_3, P_4 hold $m'_w (= x_w + \lambda'_w)$, $[\lambda'_w]$, and λ_w.
2. For each multiplication gate $(a, b, c, \times)$ or dot product gate $((a_1, ..., a_n), (b_1, ..., b_n), c, \cdot)$ ($c = \sum_{i=1}^{n} a_i b_i$).

Evaluation: For each gate (a, b, c, T) following topological order:
Evaluation Group 1 (P_1 and P_2)

1. if $T = +$: $m_c \leftarrow m_a + m_b$; $[\lambda_c] \leftarrow [\lambda_a] + [\lambda_b]$; $\lambda'_c \leftarrow \lambda'_a + \lambda'_b$
2. if $T = \cdot$ (Dot Product/Multiplication Gate)
   (a) $\left(\left[\sum_{i=1}^{n} \lambda_{a_i} \lambda_{b_i} + \lambda_c\right], \left[\lfloor \lambda_c / 2^d \rfloor\right]\right) \leftarrow \mathcal{F}_{\text{Triplet}}(a, b, c)$;
   (b) $[m_c] \leftarrow \sum_{i=1}^{n} (m_{a_i} \cdot m_{b_i} - m_{a_i} \cdot [\lambda_{b_i}] - m_{b_i} \cdot [\lambda_{a_i}]) + [\sum_{i=1}^{n} \lambda_{a_i} \cdot \lambda_{b_i} + \lambda_c]$
   (c) $m_c \leftarrow \text{open}([m_c])$; $m_c \leftarrow \lfloor (m_c + \lambda'_c)/2^d \rfloor - \lfloor \lambda'_c / 2^d \rfloor$;
   (d) $\lambda'_c \leftarrow \lfloor \lambda'_c / 2^d \rfloor$; $[\lambda_c] \leftarrow \left[\lfloor \lambda_c / 2^d \rfloor\right]$

Note: for the case of $n = 1$, the dot product gate is a multiplication gate.
Evaluation Group 2 (P_3 and P_4)

1. if $T = +$: $m'_c \leftarrow m'_a + m'_b$; $[\lambda'_c] \leftarrow [\lambda'_a] + [\lambda'_b]$; $\lambda_c \leftarrow \lambda_a + \lambda_b$
2. if $T = \cdot$ (Dot Product)
   (a) $\left(\left[\sum_{i=1}^{n} \lambda'_{a_i} \lambda'_{b_i} + \lambda'_c\right], \left[\lfloor \lambda'_c / 2^d \rfloor\right]\right) \leftarrow \mathcal{F}_{\text{Triplet}}(a, b, c)$;
   (b) $[m'_c] \leftarrow \sum_{i=1}^{n} (m'_{a_i} \cdot m'_{b_i} - m'_{a_i} \cdot [\lambda'_{b_i}] - m'_{b_i} \cdot [\lambda'_{a_i}]) + \left[\sum_{i=1}^{n} \lambda'_{a_i} \cdot \lambda'_{b_i} + \lambda'_c\right]$
   (c) $m'_c \leftarrow \text{open}([m'_c])$; $m'_c \leftarrow \lfloor (m'_c + \lambda_c)/2^d \rfloor - \lfloor \lambda_c / 2^d \rfloor$;
   (d) $\lambda_c \leftarrow \lfloor \lambda_c / 2^d \rfloor$; $[\lambda'_c] \leftarrow \left[\lfloor \lambda'_c / 2^d \rfloor\right]$

Cross Check

1. All parties make a call to $\mathcal{F}_{\mathsf{coin}}$ to sample the same random nonce r.
2. All parties compute the double masked value for each wire $d_w = m_w + \lambda'_w = m'_w + \lambda_w$. They each computes $h_i \leftarrow \mathsf{hash}(d_1 || ... || d_n || r)$.
3. P_1 and P_3 swap h_1, h_3 and verify that they are equal. Otherwise, they call abort.
4. P_2 and P_4 swap h_2, h_4 and verify that they are equal. Otherwise, they call abort.

Output: All parties output masked values of the output wires. P_1, P_2 output ($m_V^{(1)}$, $\lambda_V^{\prime(1)}$, $[\lambda_V^{(1)}]$). P_3, P_4 output ($m_V^{\prime(1)}$, $\lambda_V^{(1)}$, $[\lambda_V^{\prime(1)}]$).

Figure 4.11: Protocol to handle Masked Evaluation With Truncation

FUNCTIONALITY $\mathcal{F}_{\text{coin}}$ - Generating Random Value

The ideal functionality $\mathcal{F}_{\text{coin}}$ chooses a random $r \in Z_{2^{k+s}}$ then gives r to all the parties.

Figure 4.12: Sample a random ring element

FUNCTIONALITY $\mathcal{F}_{\text{checkZero}}$

Input Two parties (P_1, P_2 or P_3, P_4) hold shares of $[Z]$. **Functionality**

- The ideal functionality waits for shares $[Z]$ from the parties, reconstruct Z.

Output If $z_i = 0 \mod 2^{k+s}\ \forall i \in \{1, ..., n\}$, output True. Else, send False to all parties.

Figure 4.13: Ideal Functionality to verify if $[Z]$ is a share of 0.

PROTOCOL $\Pi_{\text{checkZero}}$

Input P_1 and P_2 provide shares $[Z]$ where $Z = \{z_1, ..., z_n\}$.
Protocol

1. P_1 and P_2 make a call to $\mathcal{F}_{\text{coin}}$ to sample a random nonce r.
2. P_1 and P_2 set $[Z] \leftarrow [Z] \mod 2^k$.
3. P_1 computes $h_1 \leftarrow \mathsf{hash}([z_1]_1||...||[z_n]_1||r)$. P_2 computes $h_2 \leftarrow \mathsf{hash}([z_1]_2||...||[z_n]_2||r)$
4. P_1 sends h_1 to P_2. P_2 sends h_2 to P_1.
5. P_1 verifies that $h_2 \equiv \mathsf{hash}((-[z_1]_1||...|| - [z_n]_1||r)$, otherwise, he aborts.
6. P_2 verifies that $h_1 \equiv \mathsf{hash}(-[z_1]_2||...|| - [z_n]_2||r)$, otherwise, he aborts.

Output If the check pass, output True. Else, output False.

Figure 4.14: Real-world Protocol to verify if $[Z]$ is a share of 0.

$\mathcal{F}_{\text{Triple}}$

Inputs: All parties have input (A, B, c), where A, B are input wires, and c is output wire. $A = \{a_1, ..., a_n\}$, $B = \{b_1, ..., b_n\}$, $c = \sum_{i=1}^{n} a_i b_i$.
P_1 and P_2 both provide λ'_A, λ'_B. P_3 and P_4 both provide λ_A, λ_B.
Functionality:

- If either pair sends mismatched messages, send abort to all parties.
- Sample λ'_c, λ_c uniformly at random.
- Compute $\sum_{i=1}^{n} \lambda'_{a_i} \lambda'_{b_i}$, $\sum_{i=1}^{n} \lambda_{a_i} \lambda_{b_i}$, $\lfloor \lambda'_c / 2^d \rfloor$, and $\lfloor \lambda_c / 2^d \rfloor$.

Output: P_1 and P_2 receive $[\sum_{i=1}^{n} \lambda_{a_i} \lambda_{b_i} + \lambda_c]$, and $[\lfloor \lambda_c / 2^d \rfloor]$.
P_3 and P_4 receive $[\sum_{i=1}^{n} \lambda'_{a_i} \lambda'_{b_i} + \lambda'_c]$, and $[\lfloor \lambda'_c / 2^d \rfloor]$.

Figure 4.15: Triple Generation

$\mathcal{F}_{\text{Mult}}$ **Ideal Functionality to perform multiplication up to an additive attack**

Inputs: P_1 and P_2 have inputs α. P_3 and P_4 have inputs $[X]$ ($X = \{x_1, ..., x_n\}$).
Functionality:

- Verify that P_1 and P_2 send the same α. If not, send abort to all parties.
- If the corrupted party is P_3 or P_4: wait for the attack terms $U = \{u_1, ..., u_n\}$ from that party, compute $Z = \alpha(X + U) \mod 2^{k+s}$.
- Send P_3 and P_4 shares $[\alpha]$ and $[Z]$.

Output: P_3 and P_4 output $[\alpha]$ and $[Z]$. P_1 and P_2 output nothing.

Figure 4.16: Multiplication up to an attack

$\Pi_{\textbf{Mult}}$ **Real-world protocol to perform multiplication up to an additive Attack**

Inputs: P_1 and P_2 have inputs α. P_3 and P_4 have inputs $[X]$. F is a PRF.
Protocol:

1. P_1 and P_2 make two calls to $\mathcal{F}_{\text{coin}}$ to sample two random numbers λ_α, r. They both send r to P_3 and $\lambda_\alpha - r$ to P_4. Then they compute $(\alpha - \lambda_\alpha)$. They both send $(\alpha - \lambda_\alpha)$ to P_3 and P_4. P_3 and P_4 verify that they receive the same values, otherwise, they abort.
2. P_1 and P_2 agree on a random key k_1, k_2. They both send k_1 to P_3, then k_2 to P_4. P_3 and P_4 verify that they receive the same values, otherwise, they abort.
3. P_1, P_2, and P_3 compute $[\lambda_{x_i}]_1 = F_{k_1}(i)$, $[\lambda_{z_i}]_1 = F_{k_1}(i+n)$
4. P_1, P_2, and P_4 compute $[\lambda_{x_i}]_2 = F_{k_2}(i)$.
5. P_1 and P_2 reconstruct λ_{x_i} and compute $[\lambda_{z_i}]_2 = \lambda_\alpha \lambda_{x_i} - [\lambda_{z_i}]_1$. P_1 sends $[\lambda_{z_i}]_2$ to P_4 while P_2 send $\mathsf{hash}([\lambda_{z_i}]_2)$ to P_4. P_4 verifies that they receive the correct messages from P_1 and P_2. If not, he calls abort.
6. P_3 and P_4 compute $[x_i - \lambda_{x_i}] \leftarrow [x_i] - [\lambda_{x_i}]$. They open $(x_i - \lambda_{x_i})$.
7. P_3 and P_4 compute $[z_i] \leftarrow (\alpha - \lambda_\alpha)(x_i - \lambda_{x_i}) + [\lambda_\alpha](x_i - \lambda_{x_i}) + [\lambda_{x_i}](\alpha - \lambda_\alpha) + [\lambda_{z_i}]$

Output: P_3 and P_4 output $[\alpha]$ and $[Z] = \{[z_1], ..., [z_n]\}$. P_1 and P_2 output nothing.

Figure 4.17: Multiplication up to an attack

$\mathcal{F}_{\mathsf{shuffle}}$

Inputs: P_1 and P_2 provide shares of X ($E = [E]_1 + [E]_2$).
P_3 and P_4 provide shares of X ($E = [E]_3 + [E]_4$).
Functionality:

- Verify that $[E]_1 + [E]_2 = [E]_3 + [E]_4$. If the check fails, send abort to all parties.
- Sample a random permutation π. Shuffle and re-randomize the shares $[E]_3$ and $[E]_4$ according to π: $[E^{(1)}]_3 \leftarrow \pi([E]_3)$, $[E^{(1)}]_4 \leftarrow \pi([E]_4)$
- Send $[E^{(1)}]_3$ to P_1 and P_3. Send $[E^{(1)}]_4$ to P_2 and P_4.

Output: All parties receive $[E^{(1)}]$.

Figure 4.18: Oblivious Shuffle Ideal Functionality

Π_{shuffle}

Inputs: P_1 and P_2 have shares of E ($E = [E]_1 + [E]_2$). P_3 and P_4 have shares of E ($E = [E]_3 + [E]_4$).

Protocol:

1. P_1 and P_2 make a call to $\mathcal{F}_{\mathsf{coin}}$ to sample a random MAC key $\alpha \in Z_{2^s}$.
2. Four parties make a call to $\mathcal{F}_{\mathsf{MAC}}((\alpha, [E]_1), (\alpha, [E]_2), [E]_3, [E]_4)$. P_3 and P_4 receive $[M] \equiv [\alpha E]$.
3. P_3 and P_4 make a call to $\mathcal{F}_{\mathsf{coin}}$ to sample a random value to fix the permutation π and shuffle their shares ($[E]$ and $[M]$) according to π: $[E^{(1)}] \leftarrow \pi([E])$, $[M^{(1)}] \leftarrow \pi([M])$. They send their shares of $[E^{(1)}], [M^{(1)}]$ to P_1 and P_2 respectively.
4. P_1 and P_2 compute $[Z] = \alpha[E^{(1)}] - [M^{(1)}]$ and call $\mathcal{F}_{\mathsf{checkZero}}([Z])$. If the functionality returns false, they all abort.

Output: The parties output $[E^{(1)}]$

Figure 4.19: Oblivious Shuffle Real-World Protocol

$\mathcal{F}_{\mathbf{gather}}$

Inputs: P_1, P_2 provide their shares of edges $[E]_{1,2}$, and P_3, P_4 provide their shares $[E]_{3,4}$.

Functionality:

- Sample a random MAC key β.
- Wait for shares $[E]$ from all parties. Verify that $[E]_1 + [E]_2 = [E]_3 + [E]_4$. If the verification fails, send abort to all parties. Else, reconstruct E.
- For all vertices $v \in V$, set $v \leftarrow \emptyset$.
- For each edge $\mathsf{e} \in E$ do:

  For $v \in V$ s.t. $v.\mathsf{id} = l_{\mathsf{id}}$: $v.\mathsf{Append}(e.r_{\mathsf{data}})$
- Compute $W \leftarrow \beta V$.

Output: Send additive shares $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\}, ..., \{W_{n_1}..W_{n_j}\}]$ and $[\beta]$ to P_1, P_2. P_3, P_4 receive MAC key β.

Figure 4.20: Oblivious Gather Ideal Functionality

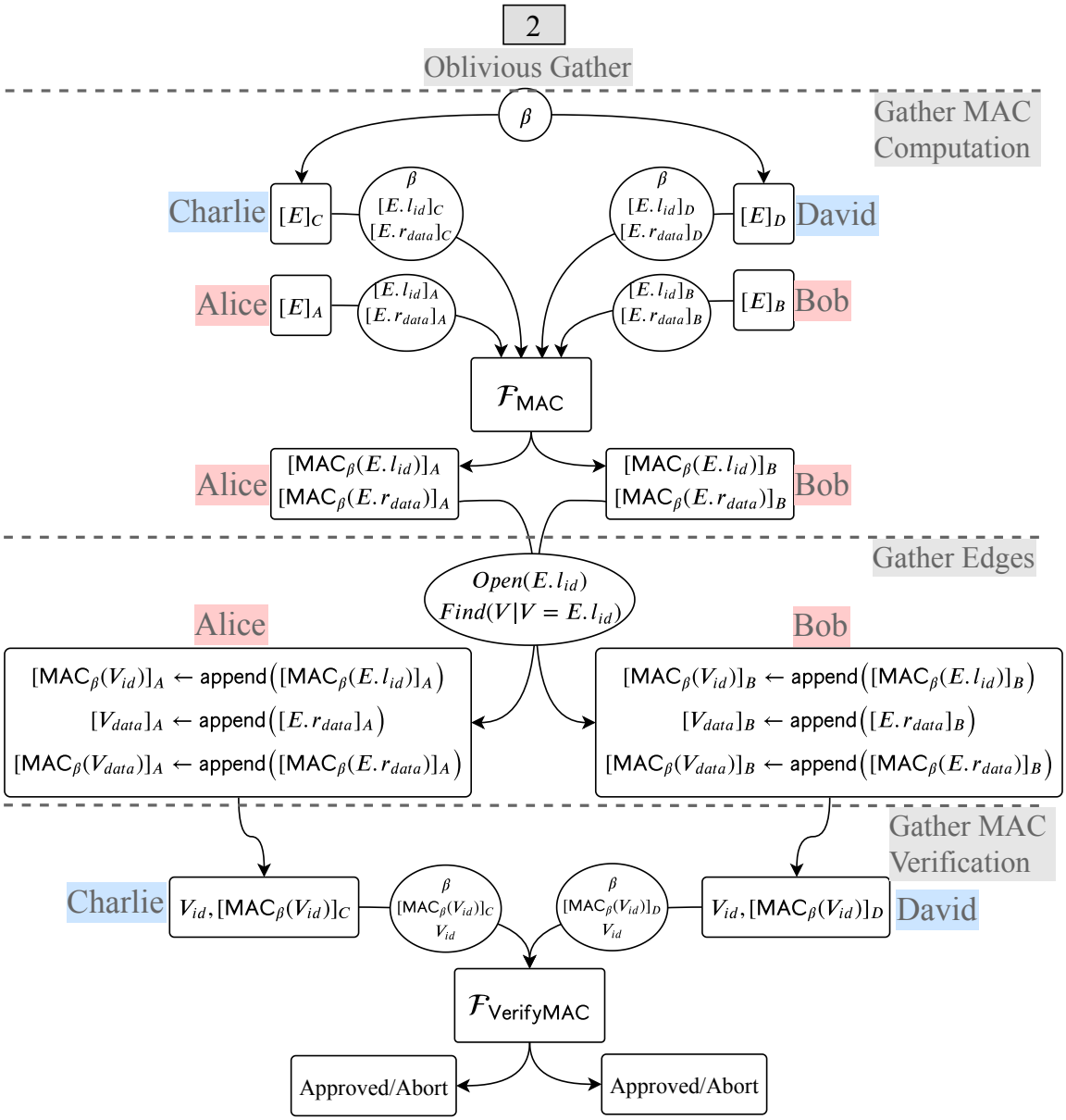


Figure 4.21: Oblivious Gather Real-World Protocol

$\mathcal{F}_{\mathsf{apply}}$

Inputs: P_1, P_2 provide their shares of vertices $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]_{1,2}$, their corresponding MAC values $[\{W_{1_1}..W_{1_i}\}, ..., \{W_{n_1}..W_{n_j}\}]_{1,2}$ and $[\beta]_{1,2}$. P_3, P_4 provide β.

Functionality:

- Verify that $\beta[V] = [W]$. If the verification fails, send abort to all parties. Else, reconstruct V.
- For $v \in [\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$:
  Compute $v^{(1)} \leftarrow func(v)$.
  note: $func$ is the computation applied on the data, e.g. computing Gradient Decent for Matrix Factorization or Addition in Histogram algorithm.

Output: Send additive shares $[\{V_{1_1}^{(1)}..V_{1_i}^{(1)}\}, ..., \{V_{n_1}^{(1)}..V_{n_j}^{(1)}\}]$ to all parties.

Figure 4.22: Oblivious Apply ideal functionality

Π_{apply}

Inputs: P_1, P_2 have shares $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\}, ..., \{W_{n_1}..W_{n_j}\}]$, $[\beta]$. P_3, P_4 have only β.

Protocol:

1. **Setting up the circuit** Four parties agree on a circuit, C_v, for each vertex they want to compute based on the structure of the vertices in V. P_1, P_2 initialize the input wires with shares $[\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$.
2. **Secure Share-Mask Conversion** Four parties call $\mathcal{F}_{[x]\to m_x}$, converting the input wires' additive shares to masked values. P_1, P_2 receive $(m_V, \lambda'_V, [\lambda_V])$. P_3, P_4 receive $(m'_V, \lambda_V, [\lambda'_V])$. e.g. $V_{11} \to (m_{V_{11}}, \lambda'_{V_{11}}, [\lambda_{V_{11}}])$.
3. **Apply Functionality**
   - For $v \in [\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$:
     Four parties execute $\mathcal{F}_{\mathsf{eval}}$ (Figure 4.10), to obtain the masked values of the updated vertex data.
4. **Secure Mask-Share Conversion** Four parties call $\mathcal{F}_{m_x\to[x]}$ to converting the masked values at the output wires to additive shares. e.g. $(m_{V_{11}}, \lambda'_{V_{11}}, [\lambda_{V_{11}}]) \to V^{(1)}_{1_1}$.

Output: The parties output $[\{V^{(1)}_{1_1}..V^{(1)}_{1_i}\}, ..., \{V^{(1)}_{n_1}..V^{(1)}_{n_j}\}]$

Figure 4.23: Oblivious Apply real-world Protocol

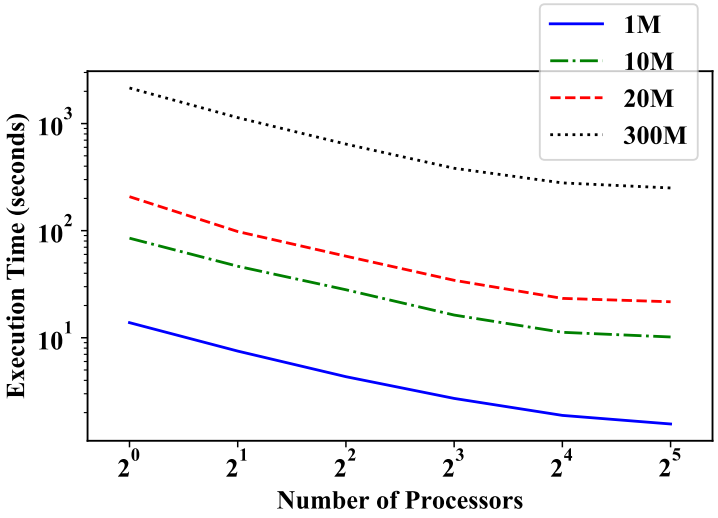


Figure 4.24: Run time (sec) for Histogram protocol with 1M, 10M, 20M and 300M edges using four-party secure computation framework with one malicious party

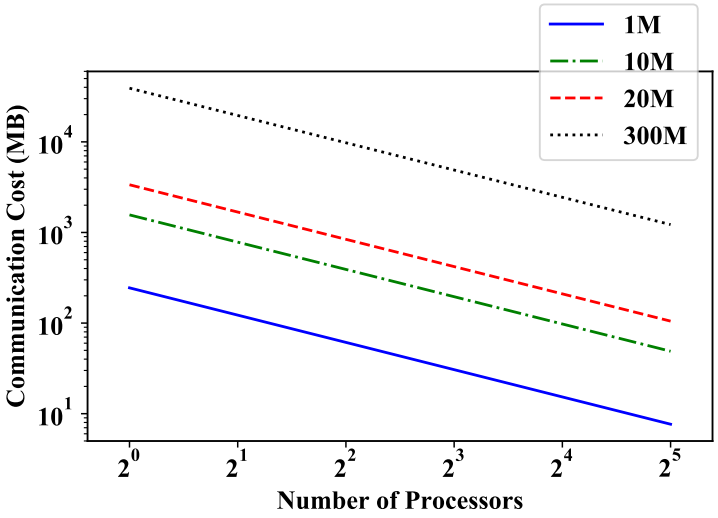


Figure 4.25: Communication cost (in MB) for Histogram protocol with 1M, 10M, 20M and 300M edges using four-party secure computation framework with one malicious party

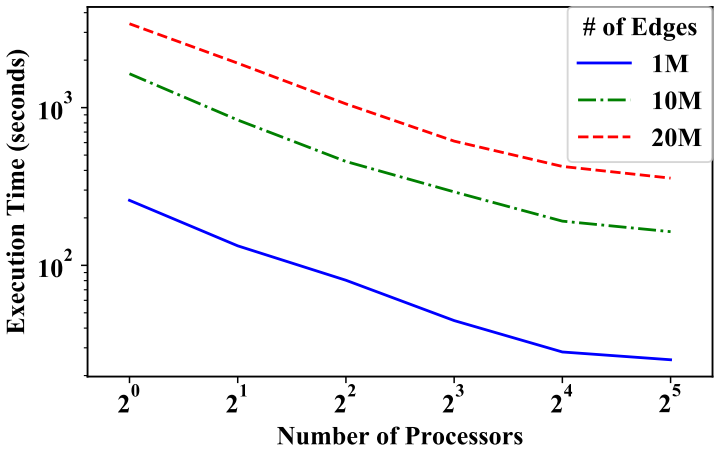


Figure 4.26: Run time (sec) for Matrix Factorization protocol with 1M, 10M and 20M edges using four-party secure computation framework with one malicious party

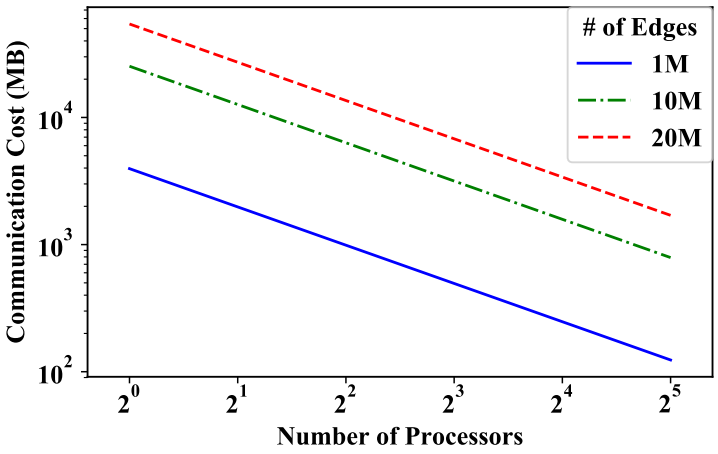


Figure 4.27: Communication cost (in MB) for Matrix Factorization protocol with 1M, 10M and 20M edges using four-party secure computation framework with one malicious party

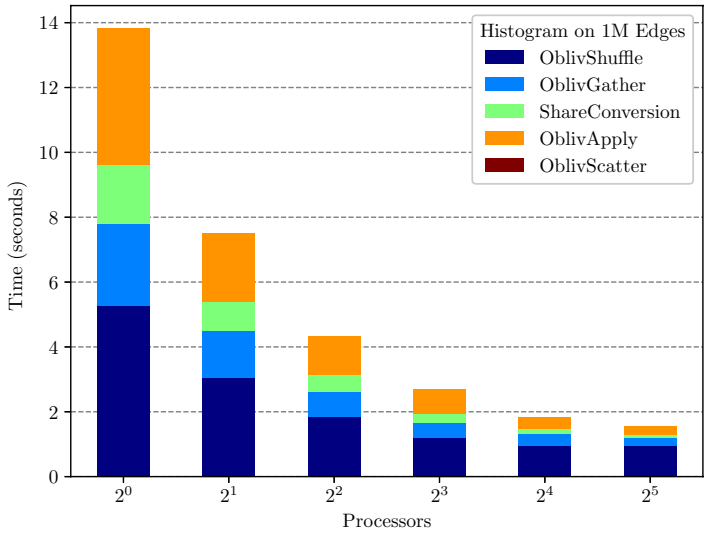


Figure 4.28: Run time for each single oblivious operation in Histogram, on input graph with 1M edges

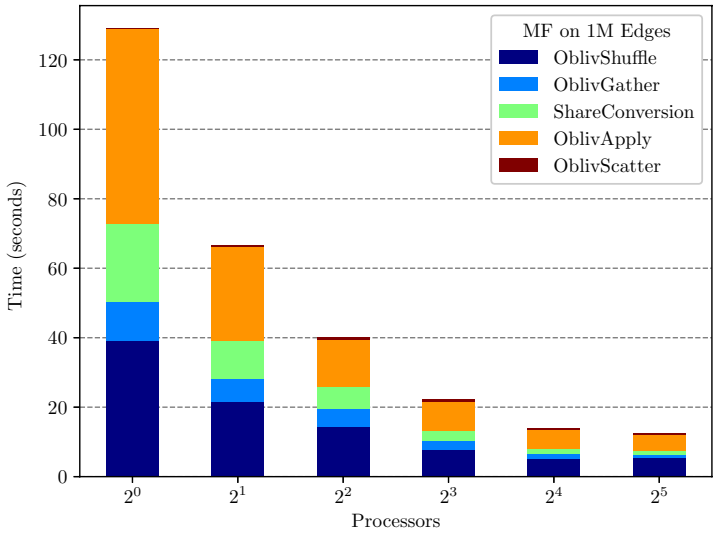


Figure 4.29: Run time for each single oblivious operation in Matrix Factorization, on input graph with 1M edges

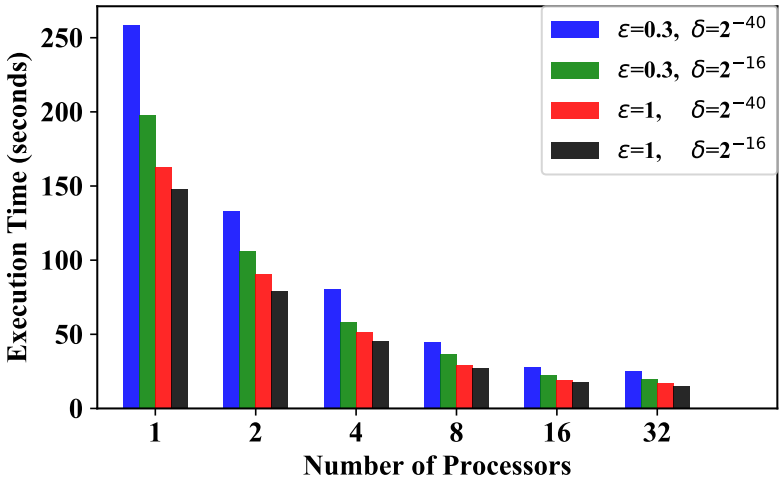
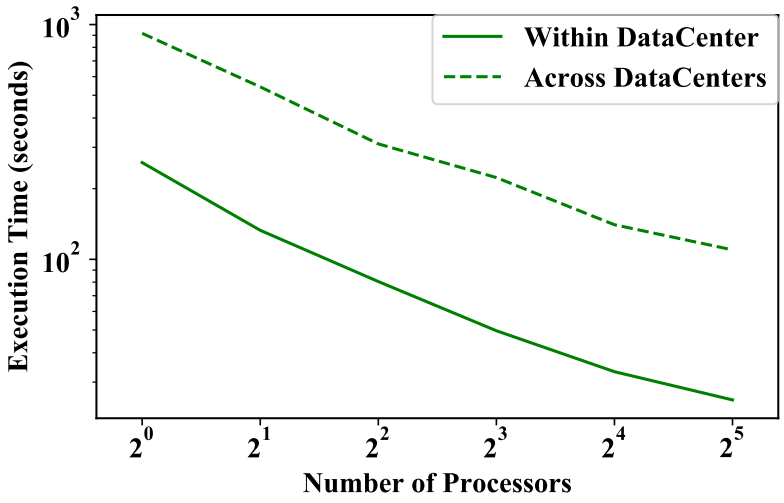


Figure 4.30: Effect of privacy parameters, ϵ and δ on running time in Matrix Factorization problem, with 6016 users and 4000 types and 1M edges



1

Figure 4.31: Run time for Matrix Factorization on input graphs with 1M edges, when all the parties are in the same data center versus when they are in different data centers to demonstrate the effect of network delay.

Chapter 5: Privacy-Preserving Federated Learning [54]

In this chapter, we present our secure and privacy-preserving machine learning solutions that are focused on distributed architecture introduced in Collaborative Learning techniques a.k.a Federated Learning. In this architecture, the sensitive data never leaves the user's system, such that, users train a local model on their private data and only share their learning parameters with the computation servers to aggregate and train a global model on all participants data. We designed and developed secure and privacy-preserving federated learning frameworks that are secure against semi-honest and malicious-secure corruption, with low communication overhead and robust against users dropouts. We leverage secure computation and differential privacy techniques to preserve the privacy of user's data as well as the trained model.

5.1 Secure and Privacy-Preserving Federate Learning

Federated Learning has recently emerged as an alternative to centralized ML algorithms. Google has conducted a comprehensive survey study on Federate Learning, its challenges and its open problems [25]. It allows multiple participants to jointly build a model on their own local training set. Each participant trains a local model on its own data and exchange these parameters with other participants (sometimes through a server or curator) to train a global model. Several architectures have been proposed for federated learning [26–30] with and without a central server. One of the main goals in developing these frameworks, is protecting the privacy of participants in the training process [13,31]. Because the training data never leave the participants' local device, federated learning can be a good candidate for the scenarios where the data is highly sensitive and cannot be shared with untrusted

parties, for example foe some of the GDPR-aware applications. Even-though these parameter updates are ephemeral and very small as compared to the user's high-dimensional private data vectors, observing these parameters by the untrusted server or any other entity, or even in some scenarios the model itself may still leak important information about user input, hence they can lead to potential adversarial attacks such as the model inversion attacks [32] or membership inference attacks [33].

One of the ways to mitigate these attacks, is to use cryptography techniques to securely share these updates with the server, and then using secure computation techniques to aggregate those parameters in order to update the global model. Secure aggregation protocols presented by [34–36] allow the untrusted central server to only learn the summation of the input vectors of many clients securely. Their protocol is robust against a fraction of users dropping out. [36] improved the efficiency of the previous secure aggregation protocols and constructs secure aggregation protocols that achieve polylogarithmic communication and computation per client. Their semi-honest construction handles billions of clients and semi-malicious construction supports tens of thousands of clients for the same per client cost. Their solutions have low-communication cost, but do not handle aggregating noisy learning parameters. Handling noise aggregation is trivial if we know how many people participate from the beginning, but much more subtle if clients frequently join and dropout during the protocol.

5.2 Our Main Framework Intuition

In our client-server scenario, a single server is interacting with n users (clients) denoted as $\mathcal{U}$. We assume each user $u \in \mathcal{U}$ holds a private vector $x_u \in \mathbb{Z}_R$, with length m. Our goal is to compute the noisy sum of users input:

$$z := \sum_{u \in \mathcal{U}} x_u + \eta \tag{5.1}$$

where $z \in \mathbb{Z}_R$, and η is a random noise sampled from a DP distribution $\mathcal{D}$. It is guaranteed that the server only learns the noisy sum of non-aborting users' input, and users learns nothing.

5.3 Definitions

5.3.1 Differential Privacy Mechanisms

Definition 10. ***(Sensitivity of a function)*** *The sensitivity of a query function $f : D \to R^d$, denoted Δf, is defined by:*

$$\Delta f = \max_{D_1, D_2} |f(D_1) - f(D_2)|$$

where the maximum is over all pairs of neighboring inputs D_1, D_2, differing in at most one element. For functions with higher dimensions where $d \geq 2$, the sensitivity is measured under ℓ_1 or ℓ_2 norms.

$$\Delta_2 f = \max_{D_1, D_2} \|f(D_1) - f(D_2)\|_2$$

Definition 11. ***(Laplace Mechanism)*** *The Laplace mechanism [6] adds noise drawn from a Laplace distribution:*

$$\mathcal{M}_{\mathsf{Lap}}(D, f, \epsilon) \triangleq f(D) + \eta$$

where $\eta \sim \mathsf{Lap}(\mu = 0, b = \frac{\Delta f}{\epsilon})$

Definition 12. ***(Gaussian Mechanism)*** *The Gaussian mechanism [6] adds noise drawn from a Gaussian distribution, and satisfies (ϵ, δ)-differential privacy according to Definition 1:*

$$\mathcal{M}_{\mathsf{Gauss}}(D, f, \epsilon, \delta) \triangleq f(D) + \eta$$

where $\eta \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{2\ln(1.25/\delta) \cdot (\Delta f)^2}{\epsilon^2})$

Definition 13. ***(Distributed Gaussian Noise Sampling)*** *Since the infinite divisibility of the Gaussian (Normal) distribution is stable, drawing a single random variable can be simulated by the sum of Gaussian random variables (independent and identically distributed). We define our noise distribution as sum of Gaussian random variables, where the number of these random variables depends on the number of corrupted users $\mathcal{C}$, ($\mathcal{C} \subseteq \mathcal{U}$), as follows:*

$$\mathcal{N}_{b,\mathcal{T}}(\sigma^2) = \sum_{u} \mathcal{N}(\frac{\sigma^2}{|\mathcal{T}|}) \tag{5.2}$$

where

$$\begin{cases} u \in \mathcal{T} & \text{if } b = 0 \text{ (honest server)} \\ u \in \mathcal{T} \setminus C & \text{if } b = 1 \text{ (corrupted server)} \end{cases} \tag{5.3}$$

5.4 MPC protocol in Semi-Honest setting

We define the ideal functionality $\mathcal{F}_{\text{PP-FL}}$ to compute noisy aggregation on clients' inputs during Federated Learning in Functionality 5.1, which is secure against semi-honest adversaries. The protocol $\pi_{\text{PP-FL}}$ that realizes this functionality appears in Protocol 5.2.

The main idea is that, users compute the masked value of their input and send it to the server. Each user generates a self-mask, and each pair of users agrees on a pairwise mask. They all send the server their data vectors, summed with each of their self-mask and the pairwise masks. They also locally samples some DP noise according to a specific distribution, and send a secret share of their masked noise to the server. Server aggregates all the noisy double-masked inputs from the users. In order to unmask the aggregated result, server needs to receive the masking values. However, due to dropout users, some of the masks will be removed in the recovery step. To handle dropout users, server notifies the surviving users of the dropout users, and have them each reply with the pairwise seed they computed with the dropout user. Pairwise seeds can be recovered even if additional

parties drop out during the recovery, as long as some minimum number of parties equal to the threshold remain alive and respond with the shares of the dropout users' pairwise seed. The server subtracts these recovery values from the masked vectors received from non-dropout users, and correctly learns the sum of the non-dropout users' data.

$\mathcal{F}_{\text{PP-FL}}$

Parameters: Let t be the dropout threshold parameter, and $m \in \mathbb{N}$ be the size of the committee.

Inputs: Each user $u \in \mathcal{U}$ holds as input a private vector $x_u \in \mathbb{Z}_R$, with length μ. The server does not provide any input.

Functionality:

1. The functionality receives (x_u, i) from each user, where $i \in \{1, \ldots, 5\}$ indicates that $u \in \mathcal{U}_i \setminus \mathcal{U}_{i+1}$. We let $\mathcal{U}' = \mathcal{U}_4$ denote the set of users for which x_u is included in the aggregation. (See Figure **??**.)
2. If $|\mathcal{U}_5| \leq t$, output $\perp$.
3. Select a random subset $\mathcal{T} \subseteq \mathcal{U}'$ with size m.
4. Let $b \in \{0, 1\}$ indicate whether the server is corrupted or not. Sample $\eta_u \leftarrow \mathcal{D}$, and set $\eta = \eta_u(|\mathcal{T} \setminus \mathcal{C}| + (1 - b)|\mathcal{T} \cap \mathcal{C}|)$, where $\mathcal{C}$ is the set of corrupt parties.
5. Compute $z := \sum_{u \in \mathcal{U}'} x_u + \eta$.

Output: Send $(z, \{\mathcal{U}_i\}_{i \in [5]})$ to the server.

Figure 5.1: Ideal Functionality for Privacy Preserving Federate Learning against semi-honest adversary

5.5 MPC protocol in Malicious Adversary setting

We define the ideal functionality $\mathcal{F}^*_{\text{PP-FL}}$ to compute noisy aggregation on clients' inputs during Federated Learning in Functionality 5.5, which is secure against malicious adversaries. The protocol $\pi^*_{\text{PP-FL}}$ that realizes this functionality appears in Protocol 5.6.

5.5.1 Secure Noise Sampling

In this section, we define the ideal functionality $\mathcal{F}^*_{\text{NoiseSample}}$ to sample DP noise in Figure 5.9. The protocol described in Figure 5.10 is using TFHE to securely sample differential privacy noise.

In order to generate noise for differential privacy, random variables can be sampled from a particular distribution, such as Laplace or Gaussian. One of the ways to sample the noise is using the inverse transform sampling method, which is useful to design and implement these sampling mechanisms inside secure computation.

[55, 56] provide constructions to sample noise from different distributions in MPC, using their inverted Cumulative Distribution Function (CDF) and *inverse transform sampling method*.

Here we describe how to sample DP noise from Laplace distribution in secure computation. Other DP distributions such as Gaussian can be securely approximated in similar way.

In order to generate a random variable ψ in Laplace distribution with parameter λ, at first, a uniformly random variable $r \in \{0, 1\}$ is generated. Then the inverse transform sampling method is used to determine

$$F^{-1}(r) = \lambda\Big(ln(r) - b\big(ln(r) + ln(1-r)\big)\Big)$$

which they optimized it further into:

$$\psi \leftarrow b\lambda\big(ln(r)\big) \tag{5.4}$$

Secure Evaluation of $ln(r)$

To implement formula 5.4, we need to perform a secure evaluation of $ln(r)$, based on approximation [55, 56]. The input r has to be scaled down to a number in interval $[1/2, 1]$.

Suppose there is a k such that $\frac{r}{2^k} \in [\frac{1}{2}, 1]$, then it holds:

$$ln(r) = ln\big(\frac{r}{2^k} \cdot 2^k\big) = ln\big(\frac{r}{2^k}\big) + k \cdot ln(2)$$

Here, $ln\big(\frac{r}{2^k}\big)$ is approximated with the following approximation method given by Hart [57]. The polynomials are given in the form

$$p(r) = \sum_{i=0}^{n} p_i \cdot r^i = \sum_{i=0}^{n} (b \cdot 10^c)_i \cdot r^i$$

where p_i is the i-th coefficient given in the form $b \cdot 10^c$. Following table 5.12, provides parameters c, b corresponding to the coefficients of the polynomial. The polynomial that is being used to approximate $ln(x)$ is referenced as p_{2607} in the book [57].

5.6 Security Analysis

In this section, we will prove the security of our protocol, assuming honest majority of parties. We consider the following two scenarios – in the first scenario the server is honest, and in the second one the server colludes with some corrupted users.

5.6.1 Semi-Honest Adversary

In this section, we will prove the security of our protocol against semi-honest adversary.

Theorem 12. *The protocol $\pi_{PP\text{-}FL}$ (Figure 5.2) securely realizes the ideal functionality $\mathcal{F}_{PP\text{-}FL}$ (Figure 5.1) with abort, under honest majority.*

Proof: To prove the theorem, we consider the following two cases: honest server $(b = 0)$, and corrupted server $(b = 1)$ that is colluding with some corrupted users.

Let $\mathcal{C}$ be the set of corrupted users and $\mathcal{U}'$ be the parties that are alive in the last round of the protocol, whom their inputs will be used for the aggregation. To prove security, we

construct a simulator $\mathcal{S}$ that interacts with the honest parties and the ideal functionality $\mathcal{F}_{\text{PP-FL}}$, and simulates the view of the corrupted parties. We first describe our simulator $\mathcal{S}$ as follow:

Simulator in the case of Honest Server: we describe the simulator as follows.

Key Distribution- For every honest party $h \in \mathcal{U} \setminus C$, $\mathcal{S}$ simulates h by following the KeyGen honestly and sends the public key pk_h to all the corrupted users.

Sharing Mask Seeds- $\mathcal{S}$ simulates each honest party h by following the KeyAgree honestly to obtain pairwise mask seed $p_{h,c}$ between honest user h and corrupted user $c \in \mathcal{C}$. To simulate Shamir-shares of mask values of honest users, $\mathcal{S}$ sends uniformly random values to corrupted parties.

Sharing Noise Seeds- $\mathcal{S}$ simulates the Shamir-shares of the two seeds (from each honest user) by random values, and sends them to the corrupted parties.

Finally, $\mathcal{S}$ calls the ideal functionality and learns $(z, \{\mathcal{U}_i\}_{i\in[5]})$. $\mathcal{S}$ then *discard*, according to the drop out pattern in $\{\mathcal{U}_i\}_{i\in[5]})$, the prior simulated messages of the honest parties, i.e., excluding the simulated messages for a party after the round it drops out. We notice that this simulation strategy is independent of the semi-honest adversary's behavior. Therefore, we can first simulate all the messages of the honest parties, and discard some of them after learning who dropped out in which round. As there is no further message sent to the corrupted users, $\mathcal{S}$ simply output this view.

Next, we are going to prove that the ideal experiment interacting with the simulator is indistinguishable from the real experiment interacting with the adversary.

Lemma 1. *For any polynomial-time semi-honest adversary $\mathcal{A}$ that corrupts at most $n/2$ users (but not the server), the random variable $\text{REAL}_{\pi,\mathcal{A}}$ is identical to that of $\text{IDEAL}_{\mathcal{F}_{\text{PP-FL}},\mathcal{S}}$.*

Proof: [Sketch.] The only difference between the real and the ideal experiments is the generation of the Shamir secret shares (in the steps Sharing masked seeds and sharing noise seeds). Since the semi-honest adversary only corrupts a minority of the users, by the security property of Shamir Secret Sharing Scheme, the distribution of the shares received by

the adversary is identical to the random distribution. Thus, the simulation is perfect.

■

Simulator in case of corrupted server: The simulator begins by submitting input 0 to the functionality on behalf of all malicious parties. The simulator learns $(z, \{\mathcal{U}_i\}_{i\in[5]})$, where z is a noisy sum of the honest inputs. The simulator simulates the first three steps of the protocol – key distribution, the sharing of mask seeds, and the sharing of noise seeds – by executing the protocol honestly on behalf of the live sub-set ($\mathcal{U}_i$ at protocol step i) of honest users.

Computing Double-masked Input: $\mathcal{S}$ simulates random $\{y_u\}_{u\in\mathcal{U}'\setminus\mathcal{C}}$ under the constraint:

$$\sum_{u\in\mathcal{U}'\setminus\mathcal{C}} y_u = z + \sum_{u\in\mathcal{U}'\setminus\mathcal{C}, v\in\mathcal{U}\setminus\mathcal{U}'} G(p_{v,u}) + \sum_{u\in\mathcal{U}'\setminus\mathcal{C}} G(q_u)$$

$$+ \sum_{u\in\mathcal{U}'\setminus(\mathcal{T}\cup\mathcal{C})} G(s_u^{(1)}) - \sum_{u\in\mathcal{T}\setminus\mathcal{C}} \Big(G(s_u^{(2)}) + (\vec{\eta_u}^{\ell\mu} \oplus \gamma_u)\Big).$$

Input Aggregation and Unmasking: The simulator generates the honest messages as the protocol, using the seeds that were previously generated.

Lemma 2. *(Semi-Honest server)*
For any polynomial-time semi-honest adversary $\mathcal{A}$ that corrupts at most $n/2$ users and the server, the random variable $\text{REAL}_{\pi,\mathcal{A}}$ *is identical to that of* $\text{IDEAL}_{\mathcal{F}_{PP\text{-}FL},\mathcal{S}}$.

Proof: We prove the above theorem by a hybrid argument. We define a simulator $\mathcal{S}$ through a series of subsequent modifications to the random variable REAL, so that any two subsequent random variables are computationally indistinguishable.

HYBRID_0 This random variable is distributed exactly as REAL, the joint view of the parties $\mathcal{P}$ in a real execution of the protocol.

HYBRID_1: Generate $\{y_u\}_{u\in\mathcal{U}'\setminus\mathcal{C}}$ randomly, and later on program the random oracle to

make the outcome consistent.

HYBRID$_2$: the ideal experiment.

5.6.2 Malicious Adversary

In this section, we will prove the security of our protocol against malicious adversary, assuming honest majority of parties, considering both scenarios; in the first scenario the server is honest, and in the second one it colludes with corrupted users.

Theorem 13. *The protocol $\pi^*_{PP\text{-}FL}$ (Figure 5.6) securely realizes the ideal functionality $\mathcal{F}^*_{PP\text{-}FL}$ (Figure 5.5), under honest majority.*

Proof: We prove the theorem for the two cases, honest server $(b = 0)$, and corrupted server $(b = 1)$ that is colluding with corrupted users.

Let $\mathcal{C}$ be the set of corrupted users and $\mathcal{U}'$ be the parties that are alive in the last round of the protocol, whom their input will be used for the aggregation. To prove security, we need to construct a simulator $\mathcal{S}$ that interacts with honest parties and the ideal functionality $\mathcal{F}_{\text{PP-FL}}$, and simulates the view of corrupted parties. We first describe our simulator $\mathcal{S}$ as follow:

Simulator in the case of Honest Server: we describe the simulator as follows.

Key Distribution- For every honest party $h \in \mathcal{U} \setminus C$, $\mathcal{S}$ simulates h by following the KeyGen honestly and sends the public key pk_h to all the corrupted users.

Sharing Mask Seeds- $\mathcal{S}$ simulates each honest party h by following the KeyAgree honestly to obtain pairwise mask seed $p_{h,c}$ between honest user h and corrupted user $c \in \mathcal{C}$. To simulate Shamir-shares of mask values of honest users, $\mathcal{S}$ sends uniformly random values to corrupted parties.

Committee Assignment- $\mathcal{S}$ simulates the Functionality $\mathcal{F}_{\text{CommitteeAssign}}$ to form the committee $\mathcal{T}$.

Noise Sampling- $\mathcal{S}$ simulates the Functionality $\mathcal{F}_{\text{NoiseSample}}$ honestly, and obtains shares of sampled noise. Each committee member gets its own share of noise.

$\pi_{\text{PP-FL}}$

Parameters: Let t be the dropout threshold parameter, and $m \in \mathbb{N}$ be the size of the committee.

Inputs: Each user $u \in \mathcal{U}$ holds as input a private vector $x_u \in \mathbb{Z}_R$, with length μ. The server has no input and can communicate with the users through secure channels.

Protocol:

- Key Distribution:
  - User u calls $(\mathsf{sk}_u, \mathsf{pk}_u) \leftarrow \mathsf{KeyGen}(pp)$ to obtain its public and private key pair.
  - User u sends its public key pk_u to the server.
  - Server broadcast the list of public keys to all users.
- Sharing Mask Seeds:
  - User u generates 'self' mask seed q_u.
  - User u computes the t-out-of-n Shamir-secret shares of self mask seed $\langle q_u \rangle_v$ and secret key $\langle \mathsf{sk}_u \rangle_v$ for $v \in \mathcal{U}$, and sends them to the server.
  - Server broadcasts each share of mask seed and that of secret key to the corresponding user.
- Sharing Noise Seeds:
  - User u samples a noise vector from a DP distribution for each of the μ slots in user input $\vec{\eta_u}^{\ell\mu} \leftarrow \mathcal{D}$, where ℓ is the bit-length of each noise value.
  - User u chooses two random seeds $s_u^{(1)}, s_u^{(2)}$, computes t-out-of-n Shamir-secret shares of them $\langle s_u^{(1)} \rangle_v$, $\langle s_u^{(2)} \rangle_v$, and sends the shares to the server. Server broadcasts them as shares of noise seeds to all users.
  - User u computes 3-out-of-3 sharing of $0^{\ell\mu}$ using a PRG G, such that: $G(s_u^{(1)}) \oplus G(s_u^{(2)}) \oplus \gamma_u \leftarrow 0^{\ell\mu}$
  - User u uses γ_u to mask noise vector $\vec{\eta_u}^{\ell m}$ by computing $\vec{\eta_u}^{\ell\mu} \oplus \gamma_u$, sends the masked noise to the server.
- Computing Double-Masked Input:
  - User u calls $p_{u,v} \leftarrow \mathsf{KeyAgree}(\mathsf{sk}_u, \mathsf{pk}_v)$ to obtain a 'pairwise' mask seed between u and v.
  - User u computes y_u and sends it to the server.

$$y_u := x_u + \sum_{v \in \mathcal{U}} G(p_{u,v}) + G(q_u) + G(s_u^{(1)})$$

Figure 5.2: Protocol for Privacy-Preserving Federated Learning against semi-honest adversary

$\pi_{\text{PP-FL}}$

- Input Aggregation and Unmasking:
  - Server receives y_u from all $u \in \mathcal{U}' \subseteq \mathcal{U}$. (Parties in $\mathcal{U} \setminus \mathcal{U}'$ are assumed to have aborted.)
  - Each user u sends a list of masking shares of other users to the server, $\langle q_v \rangle_u$ for all $v \in \mathcal{U}'$, and $\langle p_{v,u} \rangle_u$ for all $v \in \mathcal{U} \setminus \mathcal{U}'$.
  - Server randomly chooses a committee $\mathcal{T} \subset \mathcal{U}'$ of size m, asks users in $\mathcal{T}$ to send their mask seed $s_u^{(2)}$, and asks other users $\mathcal{U}' \setminus \mathcal{T}$ to send their mask seed $s_u^{(1)}$.
  - Server aggregates y_u values received from users, and unmask the result after collecting at least t shares of each masking value.

$$z := \sum_{u \in \mathcal{U}'} y_u - \sum_{u \in \mathcal{U}', v \in \mathcal{U} \setminus \mathcal{U}'} G(p_{v,u}) - \sum_{u \in \mathcal{U}'} G(q_u)$$

$$- \sum_{u \in \mathcal{U}' \setminus \mathcal{T}} G(s_u^{(1)}) + \sum_{u \in \mathcal{T}} \left(G(s_u^{(2)}) + (\vec{\eta_u}^{\ell\mu} \oplus \gamma_u) \right)$$

Output: z (noisy aggregated sum of users inputs).

Figure 5.3: Rest of the Protocol for Privacy-Preserving Federated Learning against semi-honest adversary

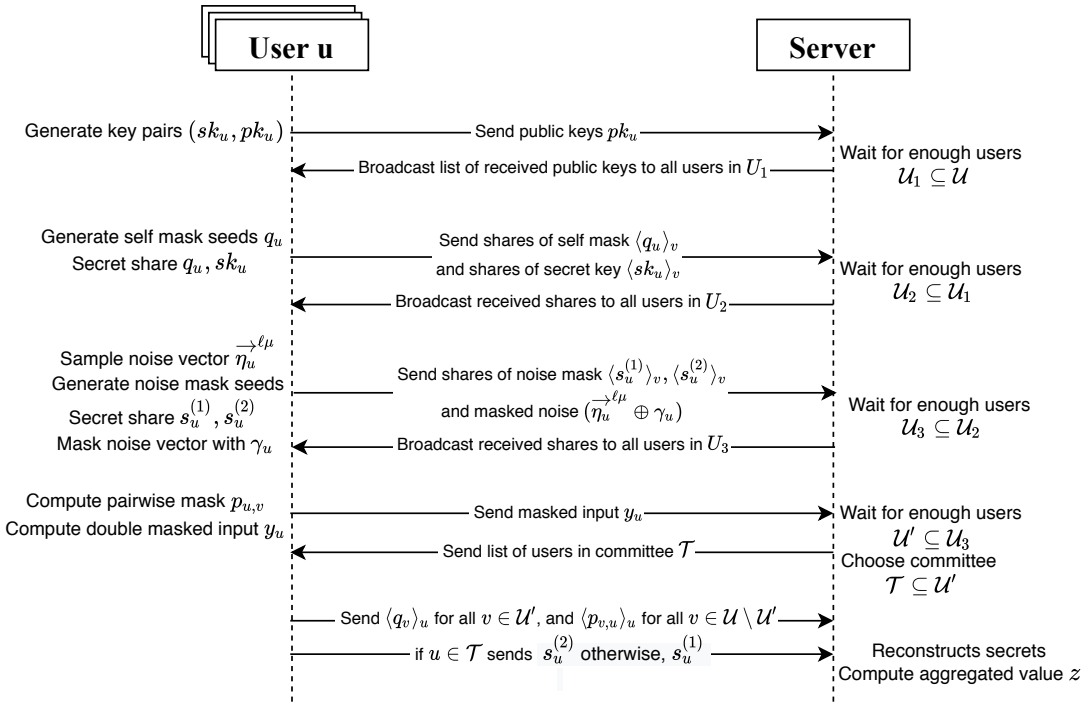


Figure 5.4: Diagram of the protocol for Privacy-Preserving Federated Learning in Semi-honest setting

$\mathcal{F}^{*}_{\text{PP-FL}}$

Parameters: Let t be the dropout threshold parameter.

Inputs: Each user $u \in \mathcal{U}$ holds as input a private vector $x_u \in \mathbb{Z}_R$, with length μ. The server does not provide any input.

Functionality:

1. The functionality receives (x_u, i) from each user, where $i \in \{1, \ldots, 5\}$ indicates that $u \in \mathcal{U}_i \setminus \mathcal{U}_{i+1}$. We let $\mathcal{U}' = \mathcal{U}_4$ denote the set of users for which x_u is included in the aggregation. (See Figure 5.8.)
2. If $|\mathcal{U}_5| \leq t$, output $\perp$.
3. Sample noise value $\eta \leftarrow \mathcal{D}$.
4. Compute $z := \sum_{u \in \mathcal{U}'} x_u + \eta$.

Output: Send $(z, \{\mathcal{U}_i\}_{i \in [5]})$ to the server.

Figure 5.5: Ideal Functionality for Privacy-Preserving Federate Learning against malicious adversary

$\pi^*_{\text{PP-FL}}$

Parameters: Let t be the dropout threshold parameter, and $m \in \mathbb{N}$ be the size of the committee $\mathcal{T}$.

Inputs: Each user $u \in \mathcal{U}$ holds as input a private vector $x_u \in \mathbb{Z}_R$, with length μ. The server has no input and can communicate with the users through secure channels.

Protocol:

- Key Distribution:
  - User u calls $(\mathsf{sk}_u, \mathsf{pk}_u) \leftarrow \mathsf{KeyGen}(pp)$ to obtain its public and private key pair.
  - User u sends its public key pk_u to the server.
  - Server broadcast the list of public keys to all the users in $\mathcal{U}$.
- Sharing Mask Seeds:
  - User u generates 'self' mask seed q_u.
  - User u computes the t-out-of-n Shamir-secret shares of self mask seed $\langle q_u \rangle_v$ and secret key $\langle \mathsf{sk}_u \rangle_v$ for $v \in \mathcal{U}$, and sends them to the server.
  - Server broadcasts shares of mask seed and shares of secret key to all users.
- Committee Assignment:
  - Parties in $\mathcal{U}$ participate in committee assignment by calling $\mathcal{T} \leftarrow \mathcal{F}_{\text{CommitteeAssign}}(m)$.
- Noise Sampling:
  - Parties in committee $\mathcal{T}$ call $([\eta]_u, \langle [\eta]_1 \rangle_u, \ldots, \langle [\eta]_m \rangle_u) \leftarrow \mathcal{F}_{\text{NoiseSample}}$, to obtain shares of sampled noise.
- Computing Noisy Double-Masked Input:
  - User u calls $p_{u,v} \leftarrow \mathsf{KeyAgree}(\mathsf{sk}_u, \mathsf{pk}_v)$ to obtain a 'pairwise' mask seed between u and v.
  - Let G be a PRG. If user $u \in \mathcal{T}$, it computes $y_u := x_u + \Sigma_{v \in \mathcal{U}'} G(p_{u,v}) + G(q_u) + [\eta]_u$, and if $u \in \mathcal{U} \setminus \mathcal{T}$ computes $y_u := x_u + \Sigma_{v \in \mathcal{U}'} G(p_{u,v}) + G(q_u)$. It sends the value y_u to the server.
- Input Aggregation and Unmasking:
  - Server receives y_u from all $u \in \mathcal{U}' \subseteq \mathcal{U}$.
  - Each user u sends a list of masking shares of other users to the server, $\langle q_v \rangle$ for all live users $v \in \mathcal{U}'$, and $\langle p_{v,u} \rangle$ for all dropout users $v \in \mathcal{U} \setminus \mathcal{U}'$. u also sends its shares of noise value $\langle [\eta]_v \rangle_u$ for all dropout users $v \in \mathcal{T} \setminus \mathcal{U}'$.

Figure 5.6: Protocol for Privacy-Preserving Federated Learning against malicious adversary

$\pi^{*}_{\text{PP-FL}}$

- Server attempt to open $p_{u,v}$; if fails, it drops y_u.
- Server attempt to open q_u; if fails, request to receive $p_{u,v}$; if v is alive and honest, it uses $p_{u,v}$ for unmasking; if v is alive and malicious, it drops both y_u and y_v.
- Server aggregates the y_u values received from users and unmask the result after collecting at least t shares of each masking value.

$$z := \sum_{u \in \mathcal{U}'} y_u - \sum_{u \in \mathcal{U}', v \in \mathcal{U} \setminus \mathcal{U}'} G(p_{v,u}) - \sum_{u \in \mathcal{U}'} G(q_u) - \sum_{u \in \mathcal{U}', v \in \mathcal{T} \setminus \mathcal{U}'} \langle [\eta]_v \rangle_u$$

Output: z (noisy aggregated sum of users inputs).

Figure 5.7: Rest of the Protocol for Privacy-Preserving Federated Learning against malicious adversary

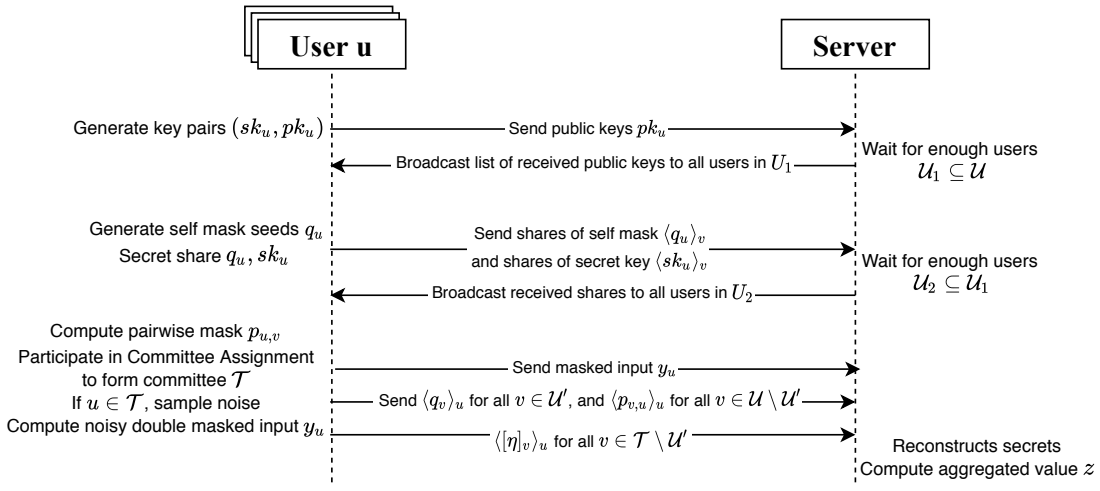


Figure 5.8: Diagram of the protocol for Privacy-Preserving Federated Learning in Malicious setting

$\mathcal{F}_{\text{NoiseSample}}$

Parameters:
Let $\mathcal{T}$ be the committee of size m, and t be the dropout threshold parameter.

Inputs: None.

Functionality:

1. Sample noise value $\eta \leftarrow \mathcal{D}$.
2. Compute (m, m)-Additive shares of the noise value, $\Sigma_{u \in m}[\eta]_u \leftarrow \eta$.
3. Compute (t, m)-Shamir shares of each $[\eta]_u$, $\langle [\eta]_u \rangle_1, \ldots, \langle [\eta]_u \rangle_m \leftarrow [\eta]_u$.

Output: Party u receives $[\eta]_u, \langle [\eta]_1 \rangle_u, \ldots, \langle [\eta]_m \rangle_u$.

Figure 5.9: Ideal Functionality for Malicious-Secure Noise Sampling

$\pi_{\text{NoiseSample}}$

Parameters: Let $\mathcal{T}$ be the committee of size m, and t be the dropout threshold parameter.

Inputs: None.

Protocol:

- Party u in committee $\mathcal{T}$ executes $(\mathsf{sk}_u, \mathsf{pk}, \mathsf{evk}) \leftarrow \mathsf{TFHE.KeyGen}(pp)$ to obtain its secret key sk_u, the common public key pk, and the evaluation key evk.
- Party u Shamir-secret shares their secret key $\langle \mathsf{sk}_u \rangle_i$ to other parties for $i \in \mathcal{U}$.
- Party u chooses a random string α_u as an initial randomness to generate DP noise, encrypts it $\beta_u \leftarrow \mathsf{TFHE.Enc}_{pk}(\alpha_u)$, and broadcasts β_u to other parties in $\mathcal{T}$.
- Parties in $\mathcal{T}$ call $\lambda \leftarrow \mathsf{TFHE.Eval}_{\mathsf{evk}}(\beta_1, \ldots, \beta_m, \mathcal{C}_{\text{DP}})$ and sample DP noise.
- Party u receives its own additive share of noise by locally decrypting DP noise λ with their own secret key, $[\eta]_u \leftarrow \mathsf{TFHE.Dec}_{\mathsf{sk}_u}(\lambda)$.
- Party u obtains threshold shares of dropout users' noise, by locally decrypting DP noise λ with their threshold shares of dropout user's secret key, $\langle [\eta]_i \rangle_u$ for $i \in \mathcal{T} \leftarrow \mathsf{TFHE.Dec}_{\langle \mathsf{sk}_i \rangle_u}(\lambda)$.

Output: Party u receives $[\eta]_u, \langle [\eta]_1 \rangle_u, \ldots, \langle [\eta]_m \rangle_u$.

Figure 5.10: Protocol for Malicious Secure Noise sampling using TFHE

$\mathcal{C}_{\mathbf{DP}}$

Inputs: random strings a_u.

Protocol:

- Reconstruct initial randomness from users' additive shares required to generate DP noise $a \leftarrow \sum_{u=1}^{m} [a]_u$
- Sample DP noise $\eta \leftarrow \mathcal{D}(a)$

Output: η.

Figure 5.11: Circuit for Noise Sampling

i	c	b
0	+1	-0.30674666858
1	+2	+0.1130516183486
2	+2	-0.2774666470302
3	+2	+0.5149518504454
4	+2	-0.6669583732238
5	+2	+0.5853503340958
6	+2	-0.3320167436859
7	+2	+0.1098927015084
8	+1	-0.161300738935

Figure 5.12: Approximation of $ln(r) \approx p_{2607}(r)$ with $r \in [1/2, 1]$ $p(r) = \sum_{i=0}^{n} (b \cdot 10^c)_i \cdot r^i$

Chapter 6: Conclusion

In this dissertation, we demonstrated how to design and develop secure and privacy-preserving machine learning frameworks based on secure computation techniques in two different adversarial models, semi-honest and malicious-secure settings, in which some small information is leaked to the computation servers, but this leakage is proven to preserve differential privacy for the users that have contributed data. More technically, the leakage is a random function of the input, revealed in the form of access patterns to memory. In these frameworks, we leverage the Graph-Parallel Computation techniques to reduce the overall running time. These frameworks follow a centralized architecture in which computation servers collects users private data in order to compute on them.

We also presented our secure and privacy-preserving machine learning solutions based on Federated Learning architecture, in which the sensitive data never leaves the user's system, such that, users train a local model on their private data and only share their learning parameters with the computation servers to aggregate and train a global model on all participants data. We designed and developed secure and privacy-preserving federated learning frameworks that are secure against semi-honest and malicious-secure corruption, with low communication overhead and robust against users dropouts. We leverage secure computation and differential privacy techniques to preserve the privacy of user's data as well as trained model.

Chapter 7: Future Work

We finish this dissertation by highlighting some open research problems for future work.

7.1 Deferentially Private Leakage in Secure Computation

In Chapter 3 and Chapter 4 we designed privacy-reserving solutions for a special class of machine learning algorithms that can be modeled in a graph parallel computation framework, such as matrix factorization and pagerank. Our work demonstrates that differentially private leakage is useful, in that it provides opportunity for more efficient protocols, and the protocol we present has broad applicability. But we leave open the very interesting question of determining, more precisely, for which class of computations this leakage might be help. Graph-parallel algorithms have the property that the access pattern to memory can be easily reduced to revealing only a histogram of the memory that is accessed, and histograms are the canonical example in the differential privacy literature. Looking at other algorithms will likely introduce very interesting leakage functions that are new to the differential privacy literature, and security might not naturally follow from known mechanisms in that space.

7.2 Security and Privacy in Deep Learning

The concept of Federate Machine Learning presented in Chapter 5, can be used with any of these learning mechanisms as the underlying learning scheme that user's use to train their local model, and send their learning parameters to the server to aggregate. Deep Learning algorithms are a class of popular and powerful techniques in machine learning for extracting complex models from data and are a great candidate to build their secure version that

could train well on encrypted data. Protecting privacy of deep learning model requires both preventing leakage of the training data during the learning process, and ensuring that the final model does not reveal unintended private information. There are many attack methods that target deep learning models, such as record or participant membership inference attacks [33, 58] or model inversion attacks [32]. For future research direction, we can focus on several differentially private deep learning schemes. Differential privacy guarantees that the output of deep learning model does not show significant statistical differences when the model was trained on adjacent datasets. The goal of the DP mechanism is to provide privacy protection for the training dataset, preventing privacy leakage in white-box or black-box scenarios. There are two main architectures to deploy deep neural networks, centralized and federated settings, and we propose several research questions for each scenario.

(A) Security and Privacy in Centralized Deep Learning

In an interesting attempt to train privacy-aware deep learning models with a centralized server (curator), [59–61] used differential privacy to protect the privacy of training data by adding noise to the learning parameters that result during the training process, aka. gradient descent values. One of the main limitations of this framework is that it assume there is always a trusted server (curator) that can access sensitive data and add DP noise to them, only protecting the final model from external adversaries. In most real world applications there is no trusted server. We can easily replace the server with a generic, fully oblivious MPC solution, however it would require high overhead since it has to touch every training sample in each iteration to hide the access pattern. Usually in most training mechanisms, only a very small fraction of samples is used in each iteration. Therefore, a good research question could be to investigate if we replace the curator with a MPC protocol, since we are revealing a subset of training samples in each iteration, are we ruining the effect of our deployed differential privacy mechanism and consequently the privacy of the final model?

Do we need full memory obliviousness to restore privacy?

(B) Security and Privacy in Federated Deep Learning

Training a deep neural network on a large dataset can be time- and resource-consuming. A recent approach is to partition the training dataset, concurrently training separate models on each subset, and exchanging parameters via a parameter server. During training, each local model pulls the parameters from this server, calculates the updates based on its current batch of training data, then pushes these updates back to the server, which again updates the global parameters. One of the recent works on the idea of collaborative learning with privacy protection is proposed by Shokri and Shmatikov [31]. The critical problem in such a scheme is that increment of training iterations and the addition of noise in each iteration, has a high negative effect on model utility. [62]. An interesting research question here is that can we use secure computation to replace the differential privacy mechanism that adds noise to parameter selection step, and how much would this improve in terms of utility?

Bibliography

Bibliography

[1] A. C. Yao, "Protocols for secure computations," in *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 1982, pp. 160–164.

[2] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of cryptography conference*. Springer, 2006, pp. 265–284.

[3] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 218–229.

[4] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," Cryptology ePrint Archive, Report 2020/521, 2020, https://eprint.iacr.org/2020/521.

[5] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1220–1237.

[6] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3-4, pp. 211–407, 2014. [Online]. Available: http://dx.doi.org/10.1561/0400000042

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[8] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1408.2041, 2014. [Online]. Available: http://arxiv.org/abs/1408.2041

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[10] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *ACM CCS 13*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM Press, Nov. 2013, pp. 801–812.

[11] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "GraphSC: Parallel secure computation made easy," in *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 377–394.

[12] J. KoneÄŊnÃ¡, B. McMahan, and D. Ramage, "Federated optimization:distributed optimization beyond the datacenter," 2015.

[13] H. B. McMahan, E. Moore, D. Ramage, S. Hampson *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[14] R. Ostrovsky and V. Shoup, "Private information storage (extended abstract)," in *29th ACM STOC*. ACM Press, May 1997, pp. 294–303.

[15] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis, "Secure two-party computation in sublinear (amortized) time," in *ACM CCS 12*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM Press, Oct. 2012, pp. 513–524.

[16] X. S. Wang, Y. Huang, T.-H. H. Chan, abhi shelat, and E. Shi, "SCORAM: Oblivious RAM for secure computation," in *ACM CCS 14*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM Press, Nov. 2014, pp. 191–202.

[17] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 359–376.

[18] S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root ORAM: Efficient random access in multi-party computation," in *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016, pp. 218–234.

[19] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," Cryptology ePrint Archive, Report 2015/1153, 2015, http://eprint.iacr.org/2015/1153.

[20] S. Wagh, P. Cuff, and P. Mittal, "Root ORAM: A tunable differentially private oblivious RAM," *CoRR*, vol. abs/1601.03378, 2016. [Online]. Available: http://arxiv.org/abs/1601.03378

[21] T. H. Chan, K.-M. Chung, B. M. Maggs, and E. Shi, "Foundations of differentially oblivious algorithms," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2019, pp. 2448–2467.

[22] X. He, A. Machanavajjhala, C. Flynn, and D. Srivastava, "Composing differential privacy and secure computation: A case study on scaling private record linkage," 2017.

[23] A. Papadimitriou, A. Narayan, and A. Haeberlen, "Dstress: Efficient differentially private computations on distributed data," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, G. Alonso, R. Bianchini, and M. Vukolic, Eds. ACM, 2017, pp. 560–574. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064218

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[25] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. GascÃşn, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. KoneÄŊnÃ¡, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. ÃŰzgÃijr, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. TramÃĺr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao, "Advances and open problems in federated learning," 2019.

[26] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[27] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[28] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[29] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[30] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[31] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. ACM, 2015, pp. 1310–1321.

[32] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1322–1333.

[33] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 3–18.

[34] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1175–1191.

[35] J. Bell, K. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly)logarithmic overhead," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 704, 2020. [Online]. Available: https://eprint.iacr.org/2020/704

[36] J. Bell, K. A. Bonawitz, A. GascÃşn, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly)logarithmic overhead," Cryptology ePrint Archive, Report 2020/704, 2020, https://eprint.iacr.org/2020/704.

[37] S. Mazloom and S. D. Gordon, "Secure computation with differentially private access patterns," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 490–507.

[38] B. Beauquier and É. Darrot, "On arbitrary size waksman networks and their vulnerability," *Parallel Processing Letters*, vol. 12, no. 03n04, pp. 287–296, 2002.

[39] R. Canetti, "Security and composition of multiparty cryptographic protocols," *Journal of Cryptology*, vol. 13, no. 1, pp. 143–202, Jan. 2000.

[40] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 21–37. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134053

[41] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, Heidelberg, Aug. 2012, pp. 643–662.

[42] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel RAM," in *ASIACRYPT 2017, Part I*, ser. LNCS, T. Takagi and T. Peyrin, Eds., vol. 10624. Springer, Heidelberg, Dec. 2017, pp. 567–597.

[43] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, "Our data, ourselves: Privacy via distributed noise generation," in *EUROCRYPT 2006*, ser. LNCS, S. Vaudenay, Ed., vol. 4004. Springer, Heidelberg, May / Jun. 2006, pp. 486–503.

[44] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2827872

[45] A. Waksman, "A permutation network," *Journal of the ACM (JACM)*, vol. 15, no. 1, pp. 159–163, 1968.

[46] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon, "Secure parallel computation on national scale volumes of data," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2487–2504. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/mazloom

[47] S. Mazloom and S. D. Gordon, "Secure computation with differentially private access patterns," in *ACM CCS 18*. ACM Press, 2018, pp. 490–507.

[48] S. D. Gordon, S. Ranellucci, and X. Wang, "Secure computation with low communication from cross-checking," ser. LNCS, T. Peyrin and S. Galbraith, Eds. Springer, Heidelberg, Dec. 2018, pp. 59–85.

[49] O. Goldreich, *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2009, vol. 2.

[50] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM CCS 18*. ACM Press, 2018, pp. 35–52.

[51] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *EUROCRYPT 2017, Part II*, ser. LNCS, J. Coron and J. B. Nielsen, Eds., vol. 10211. Springer, Heidelberg, Apr. / May 2017, pp. 225–255.

[52] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier," in *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 843–862.

[53] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority," in *CRYPTO 2018, Part II*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10992. Springer, Heidelberg, Aug. 2018, pp. 769–798.

[54] S. Mazloom and L. F.-H. Gordon, S Dov, "Secure and privacy-preserving federated learning," in *In preparation*.

[55] K. Thissen, I. L. Schoenmakers, I. R. Koster, and I. P. van Liesdonk, "Achieving differential privacy in secure multiparty computation," Ph.D. dissertation, MasterâĂŹs thesis, Eindhoven University of Technology, the Netherlands, 2019.

[56] M. V. PHI and P. K. PHI, "D3. 4 differential privacy and leakage control."

[57] J. F. Hart, *Computer Approximations*. USA: Krieger Publishing Co., Inc., 1978.

[58] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," *arXiv preprint arXiv:1805.04049*, 2018.

[59] S. Song, K. Chaudhuri, and A. D. Sarwate, "Stochastic gradient descent with differentially private updates," in *2013 IEEE Global Conference on Signal and Information Processing*. IEEE, 2013, pp. 245–248.

[60] R. Bassily, A. Smith, and A. Thakurta, "Private empirical risk minimization: Efficient algorithms and tight error bounds," in *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. IEEE, 2014, pp. 464–473.

[61] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 308–318.

[62] M. Abadi, U. Erlingsson, I. Goodfellow, H. B. McMahan, I. Mironov, N. Papernot, K. Talwar, and L. Zhang, "On the protection of private information in machine learning systems: Two recent approches," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 1–6.

Curriculum Vitae

Sahar Mazloom received her Bachelor of Science in Software engineering and Masters of Science in Artificial Intelligence from Azad University. Currently, she is working as a senior cryptography research scientist at JP Morgan Chase AI Research.