Secure parallel computation on national scale volumes of data

Sahar Mazloom*
George Mason University
sseyedma@gmu.edu

Phi Hung Le*
George Mason University
ple13@gmu.edu

Samuel Ranellucci
Unbound Tech
samuel.ranellucci@unboundtech.com

S. Dov Gordon
George Mason University
gordon@gmu.edu

Abstract

We revisit the problem of performing secure computation of graph-parallel algorithms, focusing on the applications of securely outsourcing matrix factorization, and histograms. Leveraging recent results in low-communication secure multiparty computation, and a security relaxation that allows the computation servers to learn some differentially private leakage about user inputs, we construct a new protocol that reduces overall runtime by $320X$, reduces the number of AES calls by $750X$, and reduces the total communication by $200X$. Our system can securely compute histograms over 300 million items in about 4 minutes, and it can perform sparse matrix factorization, which is commonly used in recommendation systems, on 20 million records in about 6 minutes.[1] Furthermore, in contrast to prior work, our system is secure against a malicious adversary that corrupts one of the computing servers.

1 Introduction

Instances of data breach and exfiltration continue to occur in great number. Secure computation offers an appealing avenue for defense. This cryptographic tool allows user data to be secret-shared across multiple computational servers, ensuring that the breach of any single server provides no information to an adversary, while still enabling the servers to perform arbitrary computation on the data. As compared with standard encryption, which provides security only while the data remains at rest, secure computation allows the data to remain secure throughout its life-cycle, from the moment it is uploaded by the user, through its incorporation into some statistic or learned model.

The theory of secure computation has been studied since the 1980's, and a rich literature has given rise to a line of practical work that has focused on reducing concrete costs to a near minimum. Of course, there are no free lunches, and computing on secret-shared data will always require increased communication and computation when compared with the cost of computing on plaintext data. However, several recent research directions have helped narrow the gap between secure data processing and plaintext computations.

Low communication MPC. Several results in secure computation have recently minimized the communication requirements by restricting the number of computing servers to three [2, 4, 17] or four [10], and assuming an honest majority of the servers. When representing the computation as an arithmetic circuit over a ring (as we will do here), the cheapest of these results, by Gordon et al. [10], requires sending only 1.5 ring elements per party, per circuit gate. In contrast, the best two-party protocol requires 290 bytes per party, per Boolean gate [22], and the best honest-majority protocol (supporting arbitrary numbers of parties) requires 12 field elements per party, per gate [4].

Parallelizing secure computation. Nayak et al. [18] propose a framework for securely computing *graph parallel algorithms*. In such algorithms, the data is assumed to reside in a graph structure, and the result of the computation is reached through an iterative process in which a) the data is *gathered* from all edges to their neighboring nodes, b) a simple computation is *applied* on the data at each node, and c) the processed data is *scattered* back to the neighboring edges before the processes are repeated. Such frameworks have become very popular for plaintext computations on large amounts of data, because the Apply phase can be easily distributed among many processors, making parallelization straight-forward [6, 9, 13, 14]. In this work we implement gradient descent, yielding a secure protocol for sparse matrix factorization (commonly used in recommendation systems), as well as histograms. Graph parallel frameworks are also used for PageRank, Markov random field parameter learning, parallelized Gibbs samplers, name entity resolution, and many other computations.

Allowing differentially private leakage. Very recently, researchers have explored the idea of relaxing security to al-

*Lead co-authors

[1] These numbers are for computation in a LAN. For results in a WAN, see Section 5.

low leakage in secure computation, coupled with a bound demonstrating that the leakage preserves differential privacy [3, 12, 16, 20]. Mazloom and Gordon [16] demonstrated a protocol for computing graph parallel algorithms with differentially private leakage, shaving a $\log E$ factor off of the fully secure protocol of Nayak et al., where E is the number of edges in the graph.

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.

Results. In this work, 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 4.17 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 [18] and OblivGraph [16] had 1M inputs, and required 13 hours and 2 hours of runtime, respectively, while using 4 times the number of processors that we employ, and tolerating only semi-honest corruptions. End-to-end, our construction is 320X faster than OblivGraph, the faster of these 2 systems.

Technical contributions. Merging the four-party protocol of Gordon et al. [10] with the construction of Mazloom and Gordon [16] raises several challenges and opportunities:

Fixed point arithmetic. There are few results in the MPC literature that support fixed point computation with malicious security. 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 [17], modifying the protocol of Furakawa et al. [7]. Their construction requires each party then sends 8 ring elements for each multiplication without truncation. The parties execute two subtraction circuits in pre-processing phase for each truncation. The pre-processing costs each party at least $21 \cdot (2k-d)$ bits for each truncation where k is the size of the ring, and d the length of the fraction bits. With a bit of care, we show that we can extend the four-party protocol of Gordon et al. [10] 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 20X improvement in communication over Mohassel and Rindal. The protocol of Gordon et al. proceeds through a dual 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.

Four party, linear-time, oblivious shuffle. The experimental results of Mazloom and Gordon have complexity $O(V\alpha + E)\log(V\alpha + E)$, where $\alpha = \alpha(\varepsilon, \delta)$ 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.)

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. [5] 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.

One of the benefits of using 4 parties is that we can separate the operations between two groups of parties, such that one group, for example Alice and Bob, is responsible for accessing the data during Gather and Scatter, while the other group, Charlotte and David, performs the shuffling. In contrast, in the 2-party setting, if one party knows the shuffling permutation, then the other party must access each data element in a manner that hides the data index. This seemingly requires using a short decryption key inside the secure computation, rather than a more efficient, 2-party secret sharing scheme. On the other hand, if neither party knows the shuffling permutation, we need to use a permutation network incurring the additional log overhead. When comparing our four-party, maliciously secure, oblivious shuffling protocol with the semi-honest construction of Mazloom and Gordon, they require 540X more AES calls and 140X communication than we do.

Computation over a ring. Both the work of Nayak et al. [18] and Mazloom and Gordon [16] 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. [10] 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 [16] requires 550X more AES calls and 370X more bandwidth than ours.

2 Preliminaries

2.1 Graph-parallel computation

The Graph-parallel abstraction as it is used in several frameworks such as MapReduce [6], GraphLab [13] 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. [18, 19] constructed frameworks for securely computing graph-parallel algorithms. They did this by designing a nicely parallelizable circuit for the gather and scatter phases.

2.2 MPC with differentially private leakage

The security definition for secure computation is built around the notion of protocol simulation in an *ideal world* execution [8]. 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.[3]

In prior work, Mazloom and Gordon [16] 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

[3] This brushes over some of the important technical details, but we refer the reader to a formal treatment of security in Goldreich's book [8].

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 1 *[16] 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 *$(\kappa,\varepsilon,\delta)$*-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*

$$\left\{\text{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}(E_1,E_2,E_3,E_4,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv}$$
$$\left\{\text{IDEAL}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\forall j:|e_j|)}(E_1,E_2,E_3,E_4,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad (1)$$

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

Neighboring graphs: 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 define a metric on these multi-sets in the natural way: $|D_1 - D_2| = \sum_{i=1}^{|V|} |D_1(i) - D_2(i)|$.

Applying this to graphs, for each $v \in V$, we let $\mathsf{in-deg}(v)$ denote the in-degree of node v, and we define the *in-degree profile* of a graph G as the multi-set $\mathsf{D_{in}}(G) = \{\mathsf{in-deg}(v_1),\ldots,\mathsf{in-deg}(v_n)\}$. Then, we have the following definition.

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: $|\mathsf{D_{in}}(G) - \mathsf{D_{in}}(G')| \le d$. We say that G and G' are* neighboring *if they have distance 1.*

Definition 3 *A randomized algorithm $\mathcal{L} : \mathcal{G} \to \mathcal{R}_{\mathcal{L}}$ is (ε,δ)-edge private if for all neighboring graphs, $G_1, G_2 \in \mathcal{G}$, we have:*

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

2.3 4-party computation protocol

We use the secure computation protocol by Gordon et al. for four parties, tolerating one malicious corruption [10]. 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.

2.4 Notation

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 values and back.

Function inputs Our protocol includes many function calls in which P_1 and P_2 either provide additive shares of some input, or they each provide duplicates of the same input. The same is true for P_3 and P_4. We therefore denote inputs to functionalities and protocols as a pair: the first element denotes the input of P_1 and P_2, and the second denotes that of P_3 and P_4. When P_1 and P_2 each provide an additive share of some value E, we simply denote the input by $[E]$. For example, the input to $\mathcal{F}_{\mathsf{MAC}}$ is denoted by $(([X],\alpha),[X])$: P_1 and P_2 submit additive shares of X, and each separately provide a copy of α. P_3 and P_4 provide a different additive sharing of X.

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 masked values, defined 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 fractional 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 [5] for authenticating ring elements. For a value $x \in Z_{2^k}$ and for a MAC key $\alpha \in Z_{2^s}$, the MAC on value x is defined as $\mathsf{MAC}_\alpha(x) \equiv \alpha x \in Z_{2^{k+s}}$. In our framework, $\mathsf{MAC}_\alpha(x)$ is always kept in the form of additive secret shares. [4]

We note that in our framework, all the values, the additive shares, and the masked values 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}.

[4]Technically, calling this a MAC is an abuse of terminology, since it is not a secure authentication code if αx is ever revealed. However, when computing on secret shared data, it is common to use shares of αx to prevent any incorrect manipulation of the data.

3 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 parties into 2 groups, with the first consisting of P_1 and P_2, and the second P_3 and P_4. For ease of explanation, we name the parties in the first group, Alice and Bob, and parties in the second group, Charlotte and David.

3.1 MAC Computation and Verification

One of the main challenges we face in constructing a malicious secure version of the graph operations is that we have to authenticate the values before each operation begins, and then verify correctness of the results after the operation is done. This is simple in a Field, but we choose to compute in a ring to help support fixed point operations. We adapt the MAC computation and Verification technique proposed in SPDZ2k [5]. In this part, we describe the ideal functionality and the real world protocol to generate MAC values for additive secret shares over a ring.

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

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

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, P_2 submit different values of α, send abort to all parties.
- Compute $Y = \alpha X$.

Outputs: P_1, P_2 receive nothing.
P_3, P_4 receive $[Y]$.

Figure 1: MAC computation ideal functionality

Theorem 1 *The MAC computation protocol Π_{MAC} (Figure 2) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{MAC}}$ (Figure 1) with abort, under a single malicious corruption.*

3.2 Share-Mask Conversion

We construct a method for securely converting the shared, authenticated values which was used in the Shuffle and Gather phases, into the "masked" ring values required for our four-party computation of the Apply phase.

Theorem 2 *The share-mask conversion protocol $\Pi_{\mathsf{sharemask}}(\Pi_{[x]\to m_x})$ (Figure 4) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{sharemask}}(\mathcal{F}_{[x]\to m_x})$ (Figure 3) with abort, under a single malicious corruption.*

PROTOCOL Π_{MAC}

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

1. P_1, P_2 sample a random PRF key k, by making a call to $\mathcal{F}_{\mathsf{coin}}$.
2. P_1 sends $[Y^{(1)}] = \{\alpha[X_i] + F_k(i) | i = 1, ..., n\}$ to P_3.
3. P_2 sends $[Y^{(1)}] = \{\alpha[X_i] - F_k(i) | i = 1, ..., n\}$ to P_4.
4. Four parties make a call to $\mathcal{F}_{\mathsf{mult}}(\alpha, [X]_{3,4})$.
P_3, P_4 receive $[\alpha]$ and $[Y] \leftarrow [\alpha X]$.
P_1, P_2 receive nothing.
5. P_3, 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.

Outputs: P_1, P_2 output nothing.
P_3, P_4 output $[Y]$.

Figure 2: MAC computation protocol

FUNCTIONALITY $\mathcal{F}_{\mathsf{sharemask}}(\mathcal{F}_{[x]\to m_x})$

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

- Reconstruct β, X, and Y from P_1 and P_2. Verify that P_3 and P_4 have sent shares of 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 $m'_X = X + \lambda'_X$.

Outputs: P_1 receive $(m_X, \lambda'_X, [\lambda_X]_1)$, P_2 $(m_X, \lambda'_X, [\lambda_X]_2)$
P_3 receive $(m'_X, \lambda_X, [\lambda'_X]_3)$, P_4 $(m'_X, \lambda_X, [\lambda'_X]_4)$.

Figure 3: Ideal Functionality to convert additive secret-shares to masked values

3.3 Mask-Share Conversion

At the end of the Apply phase, the result of the 4-party computation is masked values that need to be converted back to additive shares, before updating the edges. This conversion step is very simple. Each party locally converts their masked values to additive shares, without any interaction: given $x + \lambda_x$ and $[\lambda_x]$, simply output $[x] = x + \lambda_x - [\lambda_x]$.

3.4 Four-Party 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 the fractional part of the number. We provide a way to perform multiplication with masked values on fixed point numbers.

PROTOCOL $\Pi_{\mathsf{sharemask}}(\Pi_{[x]\to m_x})$

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

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

Figure 4: Real-world protocol to convert additive shares to masked values

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

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 the truncation of xy 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: if z, λ_z, λ'_z are integers, the following holds:

$\lfloor \frac{z+\lambda_z+\lambda'_z}{2^d} \rfloor = \lfloor \frac{z+\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \varepsilon_1$

$= \lfloor \frac{z}{2^d} \rfloor + \lfloor \frac{\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \varepsilon_1 + \varepsilon_2$, where $\varepsilon_i \in \{0,1\}$.

For $z \in Z_{2^k}$, $trun(z) = \begin{cases} \lfloor \frac{z}{2^d} \rfloor, & \text{if } 0 \le z \le 2^t \\ 2^k - \lfloor \frac{2^k - z}{2^d} \rfloor, & \text{if } 2^k - 2^t \le z < 2^k \end{cases}$

Assume that $-2^t \le xy < 2^t$ is the domain where xy lies in. We have two different cases.

First, we consider the case of a non-negative xy, which is represented by a ring element $z = xy$ in the range $[0; 2^t]$. The above equation works without any modifications when $(z+\lambda_z)$ and $(z+\lambda'_z)$ are both less than 2^k. This happens with probability of at least $1 - 2^{t-k+1}$ (we note that $2^t \ll 2^k$).

Second, we consider the case of a negative xy. A negative xy is represented by a ring element $z = 2^k - |xy|$ in the range $[2^k - 2^t; 2^k - 1]$. With probability of at least $1 - 2^{t-k+1}$, both λ_z and λ'_z will be chosen such that $(z+\lambda_z) \ge 2^k$ and $(z+\lambda'_z) \ge 2^k$, causing modular reduction in our computation. Specifically, for group 1, P_1 and P_2 hold $z + \lambda_z - 2^k = z + \lambda_z \mod 2^k, \lambda'_z$ and can compute the following in the integer domain:

$\lfloor \frac{(z+\lambda_z \mod 2^k)+\lambda'_z}{2^d} \rfloor = \lfloor \frac{(z+\lambda_z-2^k)+\lambda'_z}{2^d} \rfloor = \lfloor \frac{(z-2^k)+\lambda_z+\lambda'_z}{2^d} \rfloor$.

$= -\lfloor \frac{2^k - z}{2^d} \rfloor + \lfloor \frac{\lambda_z}{2^d} \rfloor + \lfloor \frac{\lambda'_z}{2^d} \rfloor + \varepsilon$, where $\varepsilon \in \{0,2\}$

Let $m_z = (2^k - \lfloor \frac{2^k - z}{2^d} \rfloor + \varepsilon) + \lfloor \frac{\lambda_z}{2^d} \rfloor = \lfloor \frac{(z+\lambda_z \mod 2^k)+\lambda'_z}{2^d} \rfloor - \lfloor \frac{\lambda'_z}{2^d} \rfloor \mod 2^k$ and $m'_z = (2^k - \lfloor \frac{2^k - z}{2^d} \rfloor + \varepsilon) + \lfloor \frac{\lambda'_z}{2^d} \rfloor = \lfloor \frac{(z+\lambda'_z \mod 2^k)+\lambda_z}{2^d} \rfloor - \lfloor \frac{\lambda_z}{2^d} \rfloor \mod 2^k$. They are the masked value of the truncation of xy for group 1 and 2 respectively.

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 $\frac{1}{2^{d-1}}$. Importantly, the error does not impact proper cross-checking of the two parallel evaluations.

Vectorization for dot products A naive way to perform a dot product between two vectors $u = \{u_1, ..., u_n\}, v = \{v_1, ..., v_n\}$ 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 6.

Communication cost Each multiplication with truncation requires the four parties to communicate only 6 rings in total when done in batch. For each gate, $\mathcal{F}_{\mathsf{triple}}$ costs 2 rings (one ring sent from P_3 to P_2, and the other from P_1 to P_4) and the opening of m_c and m'_c each costs 2 rings. F_{coin} is free when common random seeds are used, and two hashes are needed to be sent for the whole batch. We note that the cost is the same for dot product gate.

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

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

Functionality:

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

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

Figure 5: Ideal Functionality to handle Masked Evaluation With Truncation

Theorem 3 *The protocol Π_{eval} (Figure 6) securely realizes the ideal functionality $\mathcal{F}_{eval}$ (Figure 5) with abort, under a single malicious corruption.*

4 Differentially Private Graph Parallel Computation in Maliciously Secure Four-Party Settings

Our construction follows the graph-parallel computation model in which the computation is done using three main operations; Gather, Apply and Scatter. We partition the players into two 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, Charlotte and David (P_3, P_4). These parties collaboratively compute a functionality, for example Matrix Factorization. During the computation, each group is responsible for performing an operation that its results then will be verified by the other group. For example, one group securely shuffles the data, and the other group verifies that the data is not maliciously tampered, then the latter group performs the operations that access the data (e.g., gather), and then 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

π_{eval}

Inputs: For each input wire w: P_1, P_2: $m_w = x_w + \lambda_w$, λ'_w, $[\lambda_w]$; P_3, P_4: $m'_w = x_w + \lambda'_w$, λ_w, $[\lambda'_w]$.
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) $(\left[\sum_{i=1}^{n} \lambda_{a_i}\lambda_{b_i} + \lambda_c\right], [\lfloor \lambda_c / 2^d \rfloor]) \leftarrow \mathcal{F}_{\text{Triple}}(a,b,c)$;
   (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) $m_c \leftarrow \text{open}([m_c])$; $m_c \leftarrow \lfloor (m_c + \lambda'_c)/2^d \rfloor - \lfloor \lambda'_c / 2^d \rfloor$; $\lambda'_c \leftarrow \lfloor \lambda'_c / 2^d \rfloor$; $[\lambda_c] \leftarrow \left[\lfloor \lambda_c / 2^d \rfloor\right]$

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/Multiplication Gate)
   (a) $(\left[\sum_{i=1}^{n} \lambda'_{a_i}\lambda'_{b_i} + \lambda'_c\right], [\lfloor \lambda'_c / 2^d \rfloor]) \leftarrow \mathcal{F}_{\text{Triple}}(a,b,c)$;
   (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) $m'_c \leftarrow \text{open}([m'_c])$; $m'_c \leftarrow \lfloor (m'_c + \lambda_c)/2^d \rfloor - \lfloor \lambda_c / 2^d \rfloor$; $\lambda_c \leftarrow \lfloor \lambda_c / 2^d \rfloor$; $[\lambda'_c] \leftarrow \left[\lfloor \lambda'_c / 2^d \rfloor\right]$

Cross Check

1. All parties make a call to $\mathcal{F}_{\mathsf{coin}}$ to sample the same random nonce r, compute the double masked value for each wire $d_w = m_w + \lambda'_w = m'_w + \lambda_w$. They each computes $h_i \leftarrow \mathsf{hash}(d_1||...||d_n||r)$.
2. P_1 sends h_1 to P_2 and P_4. P_3 sends h_3 to P_2 and P_4.
3. P_2 verifies that $h_1 = h_3$. If true, he sends 0 to F_{or} [a] functionality, else he sends 1. P_4 does the same thing when verifying $h_1 = h_3$.
4. Repeat the previous instructions with the variable exchanged as follows, P_2 sends h_2 to P_1 and P_3, and P_4 sends h_4 to P_1 and P_3.
5. P_1 and P_3 separately verify they received same values from P_2 and P_4, and provide input to the $\mathcal{F}_{or}$ functionality, accordingly.
6. All the parties will receive the result from $\mathcal{F}_{or}$ in order to determine to continue or to 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 $(m_V^{\prime(1)}, \lambda_V^{(1)}, [\lambda_V^{\prime(1)}])$.

[a] 4-party logical OR

Figure 6: Protocol to handle Masked Evaluation With Truncation

structure. As long as the group that accesses the data does not know the permutation pattern 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 Charlotte and David handle the shuffling. At the beginning of each phase, all four parties contribute to compute MAC values of data. After computation, the verification group verifies MAC values, to prevent the malicious adversary from modifying the data.

4.1 Construction Overview

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 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 aggregated 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 performed once at the beginning of the any computation, regardless of how many iterations the computation has (both the histogram and the matrix factorization computations require only one dummy generation operation). These dummy edges are marked with a (secret shared) flag isReal, indicating that dummies should not influence the computation during the Apply phase. However, they still have node identifiers, so they contribute to the number of memory accesses to these nodes during the Gather and $\mathsf{Scatter}$ phases. The protocol we use

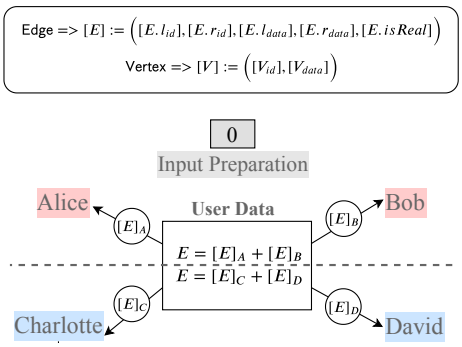


Figure 7: Input preparation phase: input data is secret-shared between both groups of parties

for generating dummy edges appears in Figure 2 of Mazloom and Gordon ([15], Definition 2).

Step 0. Input preparation: We assume the input data is additively secret-shared between 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 7.

Step 1. Oblivious Shuffle: In this step, Charlotte 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, $[M]([M] \equiv [\alpha E])$. To perform the shuffle, Charlotte 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 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 Charlotte 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, Charlotte and David should be able to verify the correctness of the Gather operation. Therefore, before the Gather operation, Charlotte 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 create 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 13: for each edge, Alice and Bob first reconstruct the id of the left vertex $\mathsf{E.l_{id}}$, locate the corresponding vertex, and then append the data of the other 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 the left side of each edge, they learn the number of times each left vertex is accessed, which leaks the degree of each vertex in the graph. However, due to the dummy edges that we shuffled-in with the real ones, what they learn is the noisy degree of each vertex, which preserve deferential privacy. At the end of this phase, Charlotte and David verify that Gather was executed correctly by calling $\mathcal{F}_{\mathsf{checkZero}}$, verifying that the data was unmodified. 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, additive shares of data are converted to masked values, then the main functionality (e.g. gradient descent) is applied on the masked values (at each vertex), and finally the masked values are converted back to additive secret-shares, which then will be used in the following phases 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-shared values need to be transformed into "masked" values. In order to convert shares to masked values, each group agrees on a vector of random mask values, denoted as λ for Alice-Bob and λ' for Charlotte-David. Then they call the $\mathcal{F}_{\mathsf{sharemask}}$ functionality and collaboratively transform the share values $[V]$ to masked values $V + \lambda$ and $V + \lambda'$.

Step 3.2. Computing the function of interest on input data: As part of the Apply phase, the parties compute the function of interest on the input data: for example, they perform addition for Histograms, or gradient descent for Matrix Factorization. The parties execute the four-party protocol described in Figure 6 to evaluate the relevant circuit.

Step 3.3. Secure Mask-Share Conversion: At the end of the Apply phase, data is in the masked format and needed to be converted to secret-shared values. As described previously, each party can locally convert their masked values to additive

secret-shares, without interacting with other parties.

Step 4. Oblivious Scatter: 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 left vertices of each edge. To complete one round of the graph computation, we need to repeat the steps 1-4 on the right vertices as well.

4.2 Oblivious Graph Operations

The hybrid world protocol is presented in Figure 9. There we assume access to ideal functionalities for Shuffle, Gather, Apply and Scatter. In this section, we explain how we instantiate each of these ideal functionalities, and provide the security proofs for each protocol under a single malicious corruption.

$\mathcal{F}_{\text{sgas}}$: Four-Party Secure Graph Parallel Computation Functionality

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

Functionality:

1. Waits for input from all parties.
2. Verifies that $[E]_A + [E]_B = [E]_C + [E]_D$. If not, sends abort to all parties.
3. Reconstructs E, then computes $E^{(1)} = func(E)$.
4. Secret shares $E^{(1)}$ to P_1, P_2; and $E^{(1)}$ to P_3, P_4.
5. Computes the leakage $\mathcal{L}(G)$, sends it to all parties.

Output: Secret shares of the updated edge values (e.g. user and item profiles). The parties also obtain the leakage $\mathcal{L}(G)$.

Figure 8: $\mathcal{F}_{\text{sgas}}$: Four-party ideal functionality for securely applying the graph parallel model of computation.

4.2.1 Four-Party Oblivious Shuffle

The Shuffle operation is used to hide the edge structure of the graph: during the Gather and Scatter operations, the vertex on each side of an edge is accessed, and shuffling the edges between these two phases hides the connection between the neighboring vertices. Additionally, the Shuffle operation mixes the dummy edges in with the real ones, which hides the exact degree of each vertex.

Π_{sgas}: Four-Party Secure Graph Parallel Computation Protocol

Input: User input is a directed graph, $\mathsf{G}(E,V)$, secret shared between the parties:
Alice, *Bob* hold secret shares of E, s.t. for each edge, $[E]_A + [E]_B = E \bmod 2^{k+s}$.
Charlotte, *David* hold secret shares of E, s.t. for each edge, $[E]_C + [E]_D = E \bmod 2^{k+s}$.
($[E]_A, [E]_B, [E]_C, [E]_D \in \mathbb{Z}_{2^{k+s}}$, 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** The parties call $\mathcal{F}_{\mathsf{gather}}([E^{(1)}])$ to aggregate edge data into vertices. Alice, Bob receive:
   $[V] = [\{V_{1_1}..V_{1_i}\}, ..., \{V_{n_1}..V_{n_j}\}]$,
   $[W] = [\{W_{1_1}..W_{1_i}\}, ..., \{W_{n_1}..W_{n_j}\}]$, and $[\beta]$, where V is the vector of gathered vertices, and $W \equiv \beta V$ is V's MAC.
   Charlotte, David receive MAC key β.
   Note: Gather leaks the noisy degree of the vertices, however, this leakage preserves differential privacy.
3. **Oblivious Apply** The players call $\mathcal{F}_{\mathsf{apply}}$ to compute the function of interest on the vertex data. Alice and Bob use input $([V], [W], [\beta])$ while Charlotte, David each provide β. 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** This step is done locally without any interaction with other parties, and each party uses $([\{V^{(1)}_{1_1}..V^{(1)}_{1_i}\}, ..., \{V^{(1)}_{n_1}..V^{(1)}_{n_j}\}])$ to update the edges and receive $[E^{(2)}]$.
   Each group sends $[E^{(2)}]$ to $\mathcal{F}_{\mathsf{rerand}}$ and receives $[E^{(3)}]$ before entering the next round of computation (Step 1).

Output: Secret shares of the edge values (e.g. user and item profiles)

Figure 9: Π_{sgas}: Four-party protocol in the hybrid-world for securely applying the graph parallel model of computation.

Theorem 4 *The Oblivious* Shuffle *protocol* Π_{shuffle} *(Figure 11 securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{shuffle}}$ *(Figure 10) with abort, under a single malicious corruption.*

Proof Theorem 4. *The Oblivious* Shuffle *protocol:* To prove

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

Inputs: P_1, P_2: $[E]$ (s.t. $[E]_1 + [E]_2 = E$).
P_3, P_4: $[E]$ (s.t. $[E]_3 + [E]_4 = E$).

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 the shares $[E]_3$ and $[E]_4$, according to π: $[E^{(1)}]_3 \leftarrow \pi([E]_3)$, $[E^{(1)}]_4 \leftarrow \pi([E]_4)$

Outputs: P_1, P_3 receive $[E^{(1)}]_3$.
P_2, P_4 receive $[E^{(1)}]_4$.

Figure 10: Oblivious Shuffle Ideal Functionality

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:

- S receives P_1's input $[E]_1$ from the distinguisher and places it in the input tape of P_1.
- $\widetilde{\alpha}$: S samples a random $\widetilde{\alpha}$ and hands it to P_1 to simulate the output from $\mathcal{F}_{\mathsf{coin}}$. S then 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 $[E]_1$ 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 $\mathcal{F}_{\mathsf{CheckZero}}$, outputs the partial view, and aborts. Else, S hands $\widetilde{b} = \mathsf{true}$ to P_1 and outputs whatever P_1 outputs.

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

$$\left\{\mathrm{HYBRID}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

Case 0: If the adversary behaves honestly, the joint distributions in the hybrid and ideal executions are:

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

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

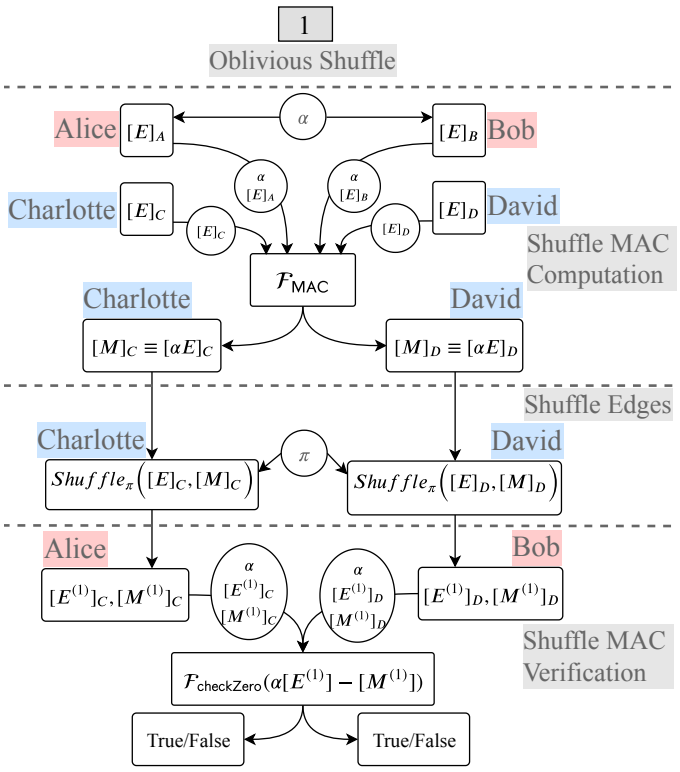


Figure 11: Oblivious Shuffle Real-World Protocol

The messages $[\alpha]$, $[\widetilde{\alpha}]$, $[M^{(1)}]$, $[\widetilde{M}^{(1)}]$, $[E^{(1)}]$ and $[\widetilde{E}^{(1)}]$ are all uniformly and independently distributed. Furthermore, $[\alpha]$, $[\widetilde{\alpha}]$, $[M^{(1)}]$, $[\widetilde{M}^{(1)}]$ are independent of the output, and the output distributions 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 $[E]_1$ to $\mathcal{F}_{\mathsf{MAC}}$, abort occurs in both worlds, and the joint distributions, $\{\alpha, \perp\}$ and $\{\widetilde{\alpha}, \perp\}$, 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:

- $[\widetilde{M}]$: S receives $[E]_3$ 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 $[E]_3$ before sending it to the functionality, S aborts and outputs the partial view. Else, S samples random ring elements as shares $[\widetilde{M}]_3$ and hands them to P_3 to simulate the output P_3 receives from $\mathcal{F}_{\mathsf{MAC}}$ in the hybrid world.
- $\widetilde{\pi}$: S queries the ideal functionality with P_3's input, $[E]_3$,

and obtains the output $[E^{(1)}]_3$. S computes $\widetilde{\pi}$ such that $[E^{(1)}]_3 \leftarrow \pi([E]_3)$, then agrees on the permutation $\widetilde{\pi}$ with P_3 in Step 3 (playing the part of P_4). 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 $[E'^{(1)}]_3 = [E^{(1)}]_3 + D$ or $[m'^{(1)}] = [m^{(1)}]_3 + D'$ where $D \neq 0 \mod 2^k, D' \neq 0 \mod 2^{k+s}$, S aborts and outputs the partial view. Else, S outputs whatever P_3 outputs.

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

$$\left\{\text{HYBRID}_{\pi_{\mathsf{shuffle}},\mathcal{A}(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{F}_{\mathsf{shuffle}},S(z)}(E,\kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}}$$

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

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

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

The messages $[M],[\widetilde{M}]$ and π, $\widetilde{\pi}$ are all distributed uniformly at random, and independently from the remainder of the view, including the joint distribution over the output shares. The output distribution is identical in both worlds as well. 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 of $[E]$, 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 D' such that $\alpha D + D' = 0 \mod 2^{k+s}$. The probability that this happens is at most 2^{-s} as shown in Section 3.1. 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. ∎

4.2.2 Four-Party Oblivious Gather

Gather operation aggregates the data from neighboring edges to each vertex. The data will be stored at the vertices for further computation handled by Apply operation.

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

Inputs:P_1, P_2: $[E]$ (s.t. $[E]_1 + [E]_2 = E$).
P_3, P_4: $[E]$ (s.t. $[E]_3 + [E]_4 = E$).
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 $W \leftarrow \beta V$.

Outputs: P_1, P_2 receive $[\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\},...,\{W_{n_1}..W_{n_j}\}]$, $[\beta]$. P_3, P_4 receive β.

Figure 12: Oblivious Gather Ideal Functionality

Theorem 5 *The Oblivious* Gather *protocol (Figure 13) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{gather}}$ (Figure 12) with abort, under a single malicious corruption.*

4.2.3 Four-Party Oblivious Apply

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

Theorem 6 *The oblivious* Apply *protocol Π_{apply} (Figure 15) securely realizes the ideal functionality $\mathcal{F}_{\mathsf{apply}}$ (Figure 14) with abort, under a single malicious corruption.*

4.2.4 Four-Party Oblivious Scatter

During the Scatter operation, the updated data in the vertices are pushed back to their corresponding edges in the graph, replacing the old values stored 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. After updating the edges, the shares are re-randomized to break the correlation between the edges (edges with the same left (or right) id are updated with the same shares during scattering phase. 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 the Shuffle operation of the next round.

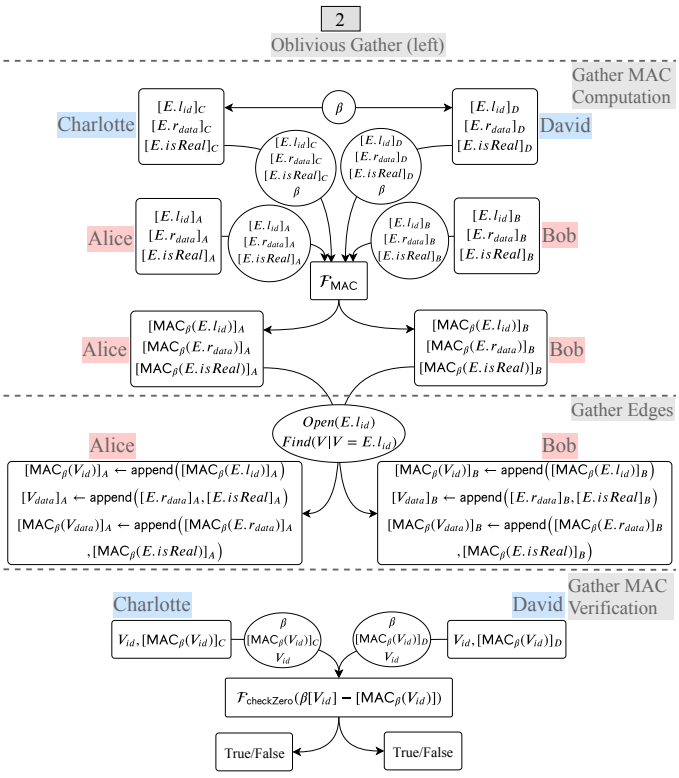


Figure 13: Oblivious Gather Real-World Protocol

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

Inputs: P_1, P_2: $[\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\},...,\{W_{n_1}..W_{n_j}\}]$, $[\beta]$.
P_3, P_4: 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: All parties receive:$[\{V_{1_1}^{(1)}..V_{1_i}^{(1)}\},...,\{V_{n_1}^{(1)}..V_{n_j}^{(1)}\}]$.

Figure 14: Oblivious Apply ideal functionality

4.3 Four-Party Secure GAS computation

In this section, we formally define our overall framework in a hybrid-world model. But first, we define the leakage function $\mathcal{L}(G)$ to be the noisy degree of each vertex in the graph, as was done by Mazloom and Gordon [15] (Definition 7). That is, in the ideal world, after receiving secret shares of the graph description, the functionality creates an array containing the

Π_{apply}

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

Protocol:

1. **Setting up the circuit** Four parties agree on a circuit, C_v, for each vertex. 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 use input $([\{V_{1_1}..V_{1_i}\},...,\{V_{n_1}..V_{n_j}\}]$, $[\{W_{1_1}..W_{1_i}\},...,\{W_{n_1}..W_{n_j}\}]$, $[\beta]$).
     P_3, P_4 use input β.
   - For each vertex, P_1, P_2 receive $(m_V, \lambda'_V, [\lambda_V])$.
     P_3, P_4 receive $(m'_V, \lambda_V, [\lambda'_V])$.
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 5), to obtain the masked values of the updated vertex data.
4. **Secure Mask-Share Conversion** Each party locally converts their masked values to additive shares.
   - P_1 and P_2 computes $[V] \leftarrow (V + \lambda_V) - [\lambda_V]$,
   - P_3 and P_4 computes $[V] \leftarrow (V + \lambda'_V) - [\lambda'_V]$.

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

Figure 15: Protocol for securely computing Apply.

vertex degrees. It then generates an equal length array of integer noise values, each independently sampled from some appropriate distribution. [5] The functionality perturbs the vertex degrees by adding the two arrays, and returns the result to the simulator. Mazloom and Gordon describe a particular distribution that is easy to sample inside a secure computation, and prove that it provides differential privacy. We use the same one in our experiments.

Theorem 7 ([16]) *The randomized algorithm $\mathcal{L}$ is (ε, δ)-approximate differentially private.*

Theorem 8 *The protocol Π_{sgas} (Figure 9) securely computes the ideal functionality $\mathcal{F}_{\mathsf{sgas}}$ (Figure 8) 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.*

5 Implementation and Evaluation

We implemented our four-party secure computation framework in C++. The source code is available at

[5] In addition to proving that the noise distribution provides privacy, we also require that all the noise values are positive, except with probability δ.

https://github.com/sama730/National-Scale-Secure-Parallel-Computation. We measure the performance of our framework on a set of benchmark algorithms in order to evaluate our design. These benchmarks consist of the 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. We compare our results with the closest large-scale secure parallel graph computation schemes, such as GraphSC [18] and OblivGraph [16].

5.1 Implementation

In 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 can 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 choose $k = 40$ 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 shared in the $Z_{2^{80}}$ ring. In our matrix factorization experiments, we factorize the ratings matrix into two matrices, represented by feature vectors that each has dimension 10. We choose these parameters as to be compatible with the GraphSC and OblivGraph representations.

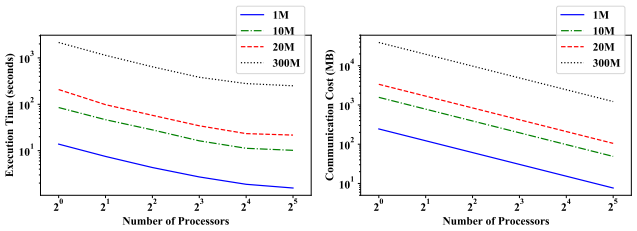
5.2 Evaluation

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 [1]. 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 [11] that contains user ratings on 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, each has 32 processors and 244 GiB RAM, with 10 Gbps network connectivity. For the LAN experiments, all instances were in the same data center (Northern Virginia). For the WAN experiments, they were spread across Northern Virginia (P_1 and P_4) and Oregon data centers (P_2 and P_3). The pairs (P_1, P_4) and (P_2, P_3) each communicate $O(1)$ ring elements in total, thus, we did not bother to separate these pairs in our WAN experiments. We use three metrics in evaluating the performance of our framework: running time in seconds, communication cost in MB, measured by the number of bits transferred between parties, and circuit size, measured by the number of AND Gates/AES operations.

The size of the graphs in all Histogram and MF experiments is as follows: $\langle$ 6K users, 4K items, 1M edges$\rangle$, $\langle$72K users, 10K items, 10M edges$\rangle$, $\langle$138K users, 27K items, 20M edges$\rangle$, and $\langle$300M users, 42K items, 300M edges$\rangle$ for Histogram only. . In all the experiments, the privacy parameters are set as $\varepsilon = 0.3$, $\delta = 2^{-40}$.

Run time and Communication Cost: Figure 16a demonstrates that the run time required to compute the Histogram protocol on a graph with 300 million edges is less than 4.17 mins, using multiprocessor machines in the LAN setting. Table 1 shows the results in more detail. Figure 16b 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.



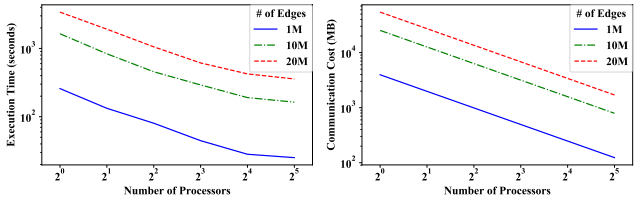
(a) Running Time (b) Communication Cost

Figure 16: Run time(s) and Communication cost(MB) of Histogram on graph sizes 1M, 10M, 20M and 300M edges

Table 1: 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	57.8	643.2
8	2.7	16.2	34.4	382.5
16	1.8	11.2	23.3	279.2
32	1.5	10.1	21.7	250.4

Similarly, Figure 17a shows that computing Matrix Factorization on large scale graph data sets takes less than 6 minutes, using our four-party framework, in our AWS LAN setting. The running time is expected to decrease linearly as we increase the number of processors, however due to some small overhead incurred by parallelization, the run time improvement is slightly sub-linear. Table 2 shows the results in details. Figure 17b shows the communication cost during Matrix Factorization on large data sets. Both graphs are in log-log scale.



(a) Running Time (b) Communication Cost

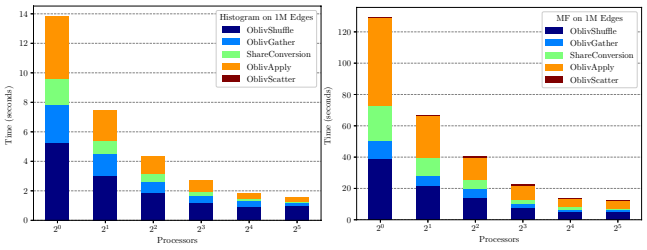
Figure 17: Run time(s) and Communication cost(MB) of Matrix Factorization on graph sizes 1M, 10M and 20M edges

Table 2: 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	455.6	1055.9
8	44.6	292.2	613.1
16	28.2	190.6	423.7
32	25.1	163.4	357.2

We measure the run time for each of the graph oblivious operations in our framework, to understand the effect of each step in the performance of the framework as a whole. Figure 18a and 18b 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 3 and 4, we demonstrated both the gain in our four party oblivious shuffle against the two party shuffle [21]



(a) Histogram (b) Matrix factorization

Figure 18: Run time for each operation in Histogram and Matrix Factorization on graph size 1M edges (LAN)

used in OblivGraph and the gain in the Apply phase with the use of arithmetic circuits in the four party setting.

Table 3: Estimated number of AES operations per party for a single iteration of matrix factorization: $|E|$ is total number of edges (real and dummies), |V| number of vertices.

	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: Estimated total communication cost for all parties(bits), for a single iteration of matrix factorization: κ is the number of bits per ciphertext, s = 40, $|E|$ is total number of edges (real and dummies), |V| number of vertices. The length of the fixed point numbers used is k = 40 bits

	OblivGraph	This work
Oblivious Shuffle	$4752\kappa(\|E\|\log\|E\|-\|E\|+1)$	$432(k+s)\|E\|$
Oblivious Gather	$32\kappa\|E\|$	$160(k+s)\|E\|$
Share Conversion	-	$(192\|E\|+120\|V\|)(k+s)$
Oblivious Apply	$186032\kappa\|E\|+2960\kappa\|V\|$	$(212\|E\|+120\|V\|)(k+s)$
Oblivious Scatter	0	0
Total	$4752\kappa\|E\|\log\|E\|+181312\kappa\|E\|+2960\kappa\|V\|+4752$	$(996\|E\|+240\|V\|)(k+s)$

Table 5, compares our running time with those of GraphSC [18] and OblivGraph [16], while computing matrix factorization on the real-world, MovieLens dataset, with 6040 users, 3883 movies, 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 using multiprocessor machines in the LAN setting, Figure 19. We also provide the number of dummy edges required for different value of ε and δ in Table 6. Note that the stated number of dummy edges are for each right node in the graph. For example, in a movie recommendation system based on our framework, we require 118 dummy edges

	GraphSC [18]	OblivGraph [16]	This work
Time	13*hrs*	2*hrs*	**25s**

Table 5: Run time comparison on this work vs. OblivGraph vs. GraphSC. Single iteration of Matrix Factorization on real-world dataset, MovieLens with 6K users ranked 4K movies with 1M ratings

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

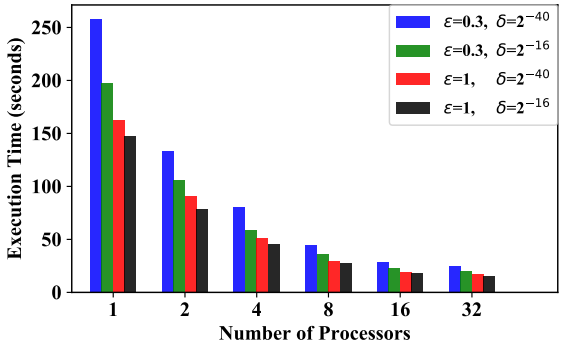


Figure 19: Effect of differential privacy parameters, ε and δ on run time in Matrix Factorization with graph size 1M edges

Table 6: 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 Figure 20 shows a dramatic slow-down in the run time when we deployed the computation servers across data centers, rather than having them in the same geographic region. Nevertheless, even in the WAN setting, we still greatly out-perform the LAN implementations of GraphSC and OblivGraph.

6 Conclusion

In this work, we combine the best results of secure multi-party computation with low-communication cost, and a security relaxation that allows the computation servers to learn some differentially private leakage about user inputs, and construct a new framework which can compute the histogram problem on 300 million users in almost 4 mins and the Matrix Factorization problem on 20 million records in about 6 mins. It reduces the overall runtime of the state of the art solution by 320X, and its communication cost by 200X. Furthermore, in contrast to prior work, our system is secure against a malicious adversary that corrupts one of the computing servers.

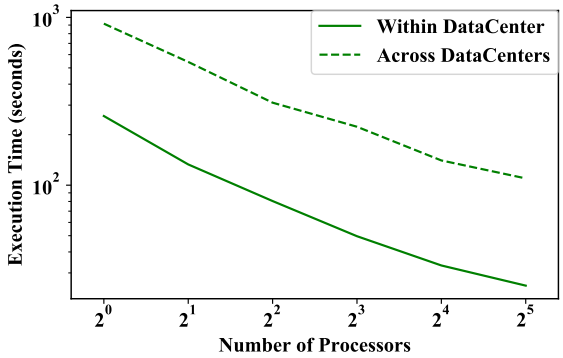


Figure 20: Run time of Matrix Factorization on graphs size 1M, showing the effect of network delay in LAN vs WAN.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-15-C-4070. It is also supported by NSF award #1564088.

References

[1] The 2020 united states census. https://2020census.gov.

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

[3] T-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *Symposium on Discrete Algorithms*, SODA '19, pages 2448–2467, 2019.

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

[5] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. pages 769–798, 2018.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

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

[8] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge University Press, 2009.

[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos 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)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[10] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. pages 59–85, 2018.

[11] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

[12] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. pages 1389–1406, 2017.

[13] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1408.2041, 2014.

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] Sahar Mazloom and S. Dov Gordon. Differentially private access patterns in secure computation. Cryptology ePrint Archive, Report 2017/1016, 2017. http://eprint.iacr.org/2017/1016.

[16] Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. pages 490–507, 2018.

[17] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. pages 35–52, 2018.

[18] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. pages 377–394, 2015.

[19] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. pages 801–812, 2013.

[20] Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private oblivious RAM. *PoPETs*, 2018(4):64–84, 2018.

[21] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968.

[22] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. pages 21–37, 2017.

A Assumed Protocols

We assume that we have access to the following oracles: $\mathcal{F}_{\mathsf{coin}}$ (Figure 21), $\mathcal{F}_{\mathsf{rerand}}$ (Figure 22), $\mathcal{F}_{\mathsf{checkZero}}$ (Figure 23), $\mathcal{F}_{\mathsf{Triple}}$ (Figure 25).

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

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

Figure 21: Sample a random ring element

FUNCTIONALITY $\mathcal{F}_{\mathbf{rerand}}$ - Rerandomize additive shares

Input Two parties P_1, P_2 hold shares of $[X]$.
Functionality

- The ideal functionality waits for shares $[X]$ from the parties, reconstruct X.
- The ideal functionality samples random values Δ, sends $[X^{(1)}]_1 = \Delta$ to P_1 and $[X^{(1)}]_2 = X - \Delta$ to P_2.

Output The parties receive $[X^{(1)}]$

Figure 22: Rerandomize additive shares

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

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

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

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 uniformly at random.
- Compute $\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}$ 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 24: Triple Generation

Π_{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_3 and P_4 both provide λ_A,λ_B.
Protocol:

- P_1, P_3, and P_4 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}]_1$ and shares $[\lfloor\lambda_c/2^d\rfloor]_1$
- P_2, P_3, and P_4 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}]_2$
- P_3 and P_4 reconstruct $\lambda_c+\sum_{i=1}^{n}\lambda_{a_i}\lambda_{b_i}$ and compute $[\lfloor\lambda_c/2^d\rfloor]_2$. P_3 sends $[\lfloor\lambda_c/2^d\rfloor]_2$ to P_2, while P_4 sends its hash to P_2. P_2 verifies shares sent from P_3 and P_4.
- P_3, P_1, and P_2 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda'_c+\sum_{i=1}^{n}\lambda_{a'_i}\lambda_{b'_i}]_1$ and shares $[\lfloor\lambda'_c/2^d\rfloor]_1$
- P_4, P_1, and P_2 query $\mathcal{F}_{\mathsf{coin}}$ to sample shares $[\lambda'_c+\sum_{i=1}^{n}\lambda_{a'_i}\lambda_{b'_i}]_2$
- P_1 and P_2 reconstruct $\lambda'_c+\sum_{i=1}^{n}\lambda_{a'_i}\lambda_{b'_i}$ and compute $[\lfloor\lambda'_c/2^d\rfloor]_2$. P_1 sends $[\lfloor\lambda_c/2^d\rfloor]_2$ to P_4, while P_2 sends its hash to P_4. P_4 verifies shares sent from P_1 and P_2.

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 25: Triple Generation

$\mathcal{F}_{\mathbf{Mult}}$ **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 shares $[\alpha]$ and $[Z]$ to P_3 and P_4.

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

Figure 26: Multiplication up to an Attack

$\Pi_{\mathbf{Mult}}$ **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 $\mathcal{F}_{\mathsf{coin}}$ 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 27: Multiplication up to an Attack