Privacy-Preserving Matrix Factorization

Valeria Nikolaenko
Stanford University
valerini@cs.stanford.edu

Stratis Ioannidis
Technicolor
stratis.ioannidis@technicolor.com

Udi Weinsberg
Technicolor
udi.weinsberg@technicolor.com

Marc Joye
Technicolor
marc.joye@technicolor.com

Nina Taft
Technicolor
nina.taft@technicolor.com

Dan Boneh
Stanford University
dabo@cs.stanford.edu

ABSTRACT

Recommender systems typically require users to reveal their ratings to a recommender service, which subsequently uses them to provide relevant recommendations. Revealing ratings has been shown to make users susceptible to a broad set of inference attacks, allowing the recommender to learn private user attributes, such as gender, age, *etc.* In this work, we show that a recommender can profile items without ever learning the ratings users provide, or even which items they have rated. We show this by designing a system that performs matrix factorization, a popular method used in a variety of modern recommendation systems, through a cryptographic technique known as garbled circuits. Our design uses oblivious sorting networks in a novel way to leverage sparsity in the data. This yields an efficient implementation, whose running time is $\Theta(M \log^2 M)$ in the number of ratings M. Crucially, our design is also highly parallelizable, giving a linear speedup with the number of available processors. We further fully implement our system, and demonstrate that even on commodity hardware with 16 cores, our privacy-preserving implementation can factorize a matrix with 10K ratings within a few hours.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Public Policy Issues—*privacy*; H.2.8 [**Database Management**]: Database Applications—*data mining, algorithms, design, performance*; G.1.6 [**Numerical Analysis**]: Optimization—*gradient methods*

Keywords

Garbled circuits; matrix factorization; multi-party computation; privacy; recommender systems

1. INTRODUCTION

A great deal of research and commercial activity in the last decade has led to the wide-spread use of recommendation systems. Such systems offer users personalized recommendations for many kinds of items, such as movies, TV shows, music, books, hotels, restaurants, and more. To receive useful recommendations, users supply substantial personal information about their preferences, trusting that the recommender will manage this data appropriately.

Nevertheless, earlier studies [49, 37, 47, 1, 46] have identified multiple ways in which recommenders can abuse such information or expose the user to privacy threats. Recommenders are often motivated to resell data for a profit [6], but also use it to extract information beyond what is intentionally revealed by the user. For example, even records of user preferences typically not perceived as sensitive, such as movie ratings or a person's TV viewing history, can be used to infer a user's political affiliation, gender, *etc.*, [61]. The private information that can be inferred from the data in a recommendation system is constantly evolving as new data mining and inference methods are developed, for either malicious or benign purposes. In the extreme, records of user preferences can be used to even *uniquely identify a user*: Naranyan and Shmatikov strikingly demonstrated this by de-anonymizing the Neflix dataset [49]. As such, even if the recommender is not malicious, an unintentional leakage of such data makes users susceptible to linkage attacks [46]. Because we cannot always foresee future inference threats, accidental information leakage, or insider threats (purposeful leakage), it is appealing to consider how one might build a recommendation system in which users do not reveal their personal data in the clear.

In this work, we study a widely used collaborative filtering technique known as matrix factorization [31, 5], that was instrumental in winning the Netflix prize competition [35], and is a core component in many real world recommendation systems. It is not a priori clear whether matrix factorization can be performed in a privacy-preserving way; there are several challenges associated with this task. First, to address the privacy concerns raised above, matrix factorization should be performed without *the recommender ever learning the users' ratings, or even which items they have rated.* This requirement is key: earlier studies [61] show that even knowing which movie a user has rated can be used to infer, *e.g.*, her gender. Second, such a privacy-preserving algorithm ought to be *efficient*, and scale gracefully (*e.g.*, linearly) with the number of ratings submitted by users. The privacy requirements imply that our matrix factorization algorithm ought to be *data-oblivious*: its execution ought to not depend on the user input. Moreover, the operations performed by matrix factorization are non-linear; thus, it is not a-priori clear how to implement matrix factorization efficiently under both

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516751.

of these constraints. Finally, in a practical, real-world scenario, users have limited communication and computation resources, and should not be expected to remain online after they have supplied their data. We thus seek a "send-and-forget" type solution, operating in the presence of users that move back and forth between being online and offline from the recommendation service.

We make the following contributions.

- We design a protocol that meets all of the above goals for privacy, efficiency and practicality. Our protocol is hybrid, combining partially homomorphic encryption with Yao's garbled circuits.
- We propose and use in our design a novel data-oblivious algorithm for matrix factorization. Implemented as a garbled circuit, it yields complexity $O(M \log^2 M)$, where M the number of submitted ratings. This is within a $\log^2 M$ factor of matrix factorization complexity in the clear. We achieve this by using Batcher sorting networks, allowing us to leverage sparsity in submitted ratings.
- Crucially, using sorting networks as a core component of our design allows us to take full advantage of the parallelization that such sorting networks enable. We incorporate this and several other optimizations in our design, illustrating that garbled circuits for matrix factorization can be brought into the realm of practicality.
- Finally, we implement our entire system using the FastGC framework [24] and evaluate it with real-world datasets. We modified the FastGC framework in two important ways, by enabling parallelized garbling and computation across multiple processors, and by reducing the memory footprint by partitioning the circuit in layers. Further additional optimizations, including reusing sorting results, and implementing operations via free XOR gates [34], allow us to run matrix factorization over 10^4 ratings within a few hours. Given that recommender systems execute matrix factorization on, *e.g.*, a weekly basis, this is acceptable for most real-life applications.

To the best of our knowledge, we are the first to enable matrix factorization over encrypted data. Although sorting networks have been used before for simple computations, our work is the first to apply sorting networks to leverage matrix sparsity, especially in a numerical task as complex as matrix factorization. Overcoming scalability and performance challenges, our solution is close to practicality for modern day recommendation services.

The remainder of this paper is organized as follows. Section 2 outlines the problem of privacy-preserving matrix factorization. Our solution is presented in Section 3. We discuss extensions in Section 4 and our implementation and experimental results in Sections 5 and 6, respectively.

2. PROBLEM STATEMENT

2.1 Matrix Factorization

In the standard "collaborative filtering" setting [35], n users rate a subset of m possible items (*e.g.*, movies). For $[n] := \{1, \ldots, n\}$ the set of users, and $[m] := \{1, \ldots, m\}$ the set of items, we denote by $\mathcal{M} \subseteq [n] \times [m]$ the user/item pairs for which a rating has been generated, and by $M = |\mathcal{M}|$ the total number of ratings. Finally, for $(i, j) \in \mathcal{M}$, we denote by $r_{ij} \in \mathbb{R}$ the rating generated by user i for item j.

In a practical setting, both n and m are large numbers, typically ranging between 10^4–10^6. In addition, the ratings provided are *sparse*, that is, $M = \Theta(n + m)$, which is much smaller than the total number of potential ratings $n \cdot m$. This is consistent with typical user behavior, as each user may rate only a relatively small number of items (not depending on m, the "catalogue" size).

Given the ratings in $\mathcal{M}$, a recommender system wishes to predict the ratings for user/item pairs in $[n] \times [m] \setminus \mathcal{M}$. Matrix factorization performs this task by fitting a bi-linear model on the existing ratings. In particular, for some small dimension $d \in \mathbb{N}$, it is assumed that there exist vectors $u_i \in \mathbb{R}^d$, $i \in [n]$, and $v_j \in \mathbb{R}^d$, $j \in [m]$, such that

$$r_{ij} = \langle u_i, v_i \rangle + \varepsilon_{ij}$$

where ε_{ij} are i.i.d. Gaussian random variables. The vectors u_i and v_j are called the *user* and *item* profiles, respectively. We will use the notation $U = [u_i^T]_{i \in [n]} \in \mathbb{R}^{n \times d}$ for the $n \times d$ matrix whose i-th row comprises the profile of user i, and $V = [v_j^T]_{i \in [m]} \in \mathbb{R}^{m \times d}$ for the $m \times d$ matrix whose j-th row comprises the profile of item j.

Given the ratings $\{r_{ij} : (i, j) \in \mathcal{M}\}$, the recommender typically computes the profiles U and V by performing the following *regularized least squares minimization*:[1]

$$\min_{U,V} \frac{1}{M} \sum_{(i,j) \in \mathcal{M}} (r_{ij} - \langle u_i, v_j \rangle)^2 + \lambda \sum_{i \in [n]} \|u_i\|_2^2 + \mu \sum_{j \in [m]} \|v_j\|_2^2 \quad (1)$$

for some positive $\lambda, \mu > 0$. The computation of U, V through (1) is a computationally intensive task even in the clear, and is typically performed by recommenders in "batch-mode", *e.g.*, once a week, using ratings collected thus far. These profiles are subsequently used to predict ratings through:

$$\hat{r}_{ij} = \langle u_i, v_j \rangle, \quad i \in [n], j \in [m] \ . \quad (2)$$

The regularized mean square error in (1) is not a convex function; several methods for performing this minimization have been proposed in literature [35, 31, 5]. We focus on gradient descent [35], a popular method used in practice, which we review below. Denoting by $F(U, V)$ the regularized mean square error in (1), gradient descent operates by iteratively adapting the profiles U and V through the adaptation rule

$$\begin{aligned} u_i(t) &= u_i(t-1) - \gamma \nabla_{u_i} F(U(t-1), V(t-1)) \,, \\ v_i(t) &= v_i(t-1) - \gamma \nabla_{v_i} F(U(t-1), V(t-1)) \,, \end{aligned} \quad (3)$$

where $\gamma > 0$ a small gain factor and

$$\begin{aligned} \nabla_{u_i} F(U, V) &= -2 \textstyle\sum_{j:(i,j) \in \mathcal{M}} v_j (r_{ij} - \langle u_i, v_j \rangle) + 2\lambda u_i \,, \\ \nabla_{v_j} F(U, V) &= -2 \textstyle\sum_{i:(i,j) \in \mathcal{M}} u_i (r_{ij} - \langle u_i, v_j \rangle) + 2\mu v_i \,, \end{aligned} \quad (4)$$

where $U(0)$ and $V(0)$ consist of uniformly random norm 1 rows (*i.e.*, profiles are selected u.a.r. from the norm 1 ball).

2.2 Setting

Figure 1 depicts the actors in our privacy-preserving matrix factorization system. Each user $i \in [n]$ wants to keep her ratings $\{r_{ij} : (i, j) \in \mathcal{M}\}$ private. The recommender system (RecSys), performs the privacy-preserving matrix factorization, while a *crypto-service provider* (CSP), enables this private computation.

[1] Assuming Gaussian priors on the profiles U and V, the minimization (1) corresponds to maximum likelihood estimation of U and V.

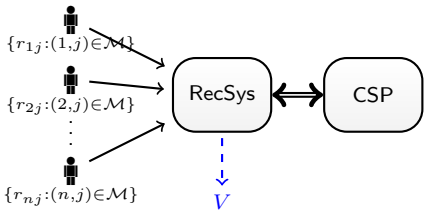


Figure 1: The parties in our first protocol design. The recommender system learns nothing about users' ratings, other than the model V.

Our objective is to design a protocol that allows the RecSys to execute matrix factorization while *neither the RecSys nor the CSP learn anything other than the item profiles*,[2] *i.e.*, V (the sole output of RecSys in Fig. 1). In particular, neither should learn a user's ratings, or even which items she has actually rated. Clearly, a protocol that allows the recommender to learn *both* user *and* item profiles reveals too much: in such a design, the recommender can trivially infer a user's ratings from the inner product (2). As such, our focus is on designing a privacy-preserving protocol in which the recommender learns only the item profiles.

There is a utility in learning the item profiles alone. First, the embedding of items in $\mathbb{R}^d$ through matrix factorization allows the recommender to infer (and encode) similarity: items whose profiles have small Euclidean distance are items that are rated similarly by users. As such, the task of learning the item profiles is of interest to the recommender beyond the actual task of recommendations. Second, having obtained the item profiles, there is a way the recommender can use them to provide relevant recommendations without any additional data revelation by users: the recommender can send V to a user (or release it publicly); knowing her ratings, i can infer her (private) profile u_i by solving (1) w.r.t. u_i through ridge regression [15]. Having u_i and V, she can subsequently predict all ratings through (2).

Both of these scenarios presume that neither the recommender nor the users object to the public release of V. However, in Section 4.1 we show that there is also a way to easily extend our design so that *users learn their predicted ratings* while *the recommender learns nothing, not even* V. For the sake of simplicity, as well as on account of the utility of such a protocol to the recommender, our main focus is on allowing the recommender to learn the item profiles. This design is also the most technically challenging; we treat the second case as an extension to this basic design.

We note that, in general, both output of the profile V or the rating predictions for a user may reveal something about other users' ratings. In pathological cases where there are, *e.g.*, only two users, both revelations may let the users discover each other's ratings. We do not address such cases; when the privacy implications of the revelation of item profiles or individual ratings are not tolerable, techniques such as adding noise can be used, as discussed in Section 7.

Threat Model. Our security guarantees will hold under the *honest but curious* threat model [17]. In other words, the RecSys and CSP follow the protocols we propose as prescribed; however, these interested parties may elect to analyze protocol transcripts, even off-line, in order to infer some additional information. We further assume that the recommender and CSP do not collude; the case of a malicious RecSys is discussed in Section 4.

[2]Technically, our algorithm also leaks the number of items rated by each user, though this can be easily rectified through a simple protocol modification.

3. LEARNING THE ITEM PROFILES

In this section, we present a solution allowing the recommender system to learn only the item profiles V. Our approach is based on Yao's garbled circuits. In short, in a basic architecture, the CSP prepares a garbled circuit that implements matrix factorization, and provides it to the RecSys. Oblivious transfer is used to obtain the garbled inputs from users without revealing them to either the CSP or the RecSys. Our design augments this basic architecture to enable users to be offline during the computation phase, as well as to submit their ratings only once.

We begin by discussing how Yao's protocol for secure multi-party computation applies, and then focus on the challenges that arise in implementing matrix factorization as a circuit. Our solution yields a circuit with a complexity within a polylogarithmic factor of matrix factorization performed in the clear by using *sorting networks*; an additional advantage of this implementation is that the garbling and the execution of our circuit is highly parallelizable.

3.1 A Privacy-Preserving Protocol for Matrix Factorization

Yao's Garbled Circuits. Yao's garbled circuit method, outlined in Appendix A, can be applied to our setting in a manner similar to the privacy-preserving auction and ridge regression settings studied in [48] and [50], respectively. In brief, assume for now that there exists a circuit that implements matrix factorization: this circuit receives as input user's ratings, supplied as tuples of the form

$$(i, j, r_{ij}) \text{ with } (i,j) \in \mathcal{M}\,,$$

where r_{ij} represents the rating r_{ij} of user i on item j, and outputs the item profiles V. Given M, the CSP garbles this circuit and sends it to the RecSys. In turn, through proxy oblivious transfer [48] between the users, the RecSys and the CSP, the RecSys receives the circuit inputs and evaluates V.

As garbled circuits can only be used once, any future computation on the same ratings would require the users to resubmit their data through proxy oblivious transfer. For this reason, we adopt a hybrid approach, combining public-key encryption with garbled circuits, as in [57, 50]. Applied to our setting, in a hybrid approach the CSP advertises a public key. Each user i encrypts her respective inputs (j, r_{ij}) and submits them to the RecSys. Whenever the RecSys wishes to perform matrix factorization over M accumulated ratings, it reports M to the CSP. The CSP provides the RecSys with a garbled circuit that (a) decrypts the inputs and then (b) performs matrix factorization; (Yao's complete protocol can be found in Appendix A). Nikolaenko *et al.* [50] avoid decryption within the circuit by using masks and homomorphic encryption; we adapt this idea to matrix factorization, departing however from [50] by only requiring a *partially* homomorphic encryption scheme.

We note that our protocol, and the ones outlined above, leak beyond V also the number of ratings generated by each user. This can easily be remedied, *e.g.*, by pre-setting the maximum number of ratings the user may provide and

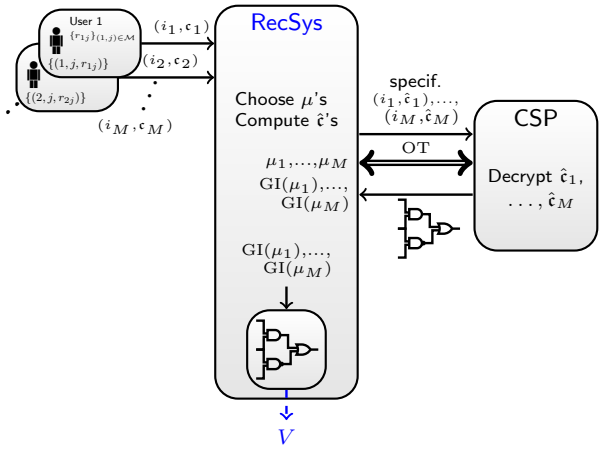


Figure 2: Protocol overview: Learning item profiles V through a garbled circuit. Each user submits encrypted item-rating pairs to the RecSys. The RecSys masks these pairs and forwards them to the CSP. The CSP decrypts them, and embeds them in a garbled circuit sent to the RecSys. The garbled values of the masks (denoted by GI) are obtained by the RecSys through oblivious transfer.

padding submitted ratings with "null" entries; for simplicity, we describe the protocol without this padding operation.

Detailed Description. We use public key encryption as follows. Each user i encrypts her respective inputs (j, r_{ij}) under the CSP's public key, pk_{csp}, with a semantically secure encryption algorithm $\mathcal{E}_{\mathsf{pk}}$, and, for each item j she rated, submits a pair $(i, \mathfrak{c})$ with $\mathfrak{c} = \mathcal{E}_{\mathsf{pk}_{\text{csp}}}(j, r_{ij})$ to the RecSys, where M ratings are submitted in total. A user that submitted her ratings can go off-line.

We require that the CSP's public-key encryption algorithm is *partially* homomorphic: a constant can be applied to an encrypted message without the knowledge of the corresponding decryption key. Clearly, an additively homomorphic scheme such as Paillier [52] or Regev [56] can be used to add a constant, but hash-ElGamal (see, *e.g.*, [7, § 3.1]), which is only partially homomorphic, suffices and can be implemented more efficiently in this case; we review this implementation in Appendix B.

Upon receiving M ratings from users—recalling that the encryption is partially homomorphic—the RecSys obscures them with random masks $\hat{\mathfrak{c}} = \mathfrak{c} \oplus \mu$, and sends them to the CSP together with the complete specifications needed to build a garbled circuit. In particular, the RecSys specifies the dimension of the user and item profiles (*i.e.*, parameter d), the total number of ratings (*i.e.*, parameter M), the total number of users and items (*i.e.*, n and m), as well as the number of bits used to represent the integer and fractional parts of a real number in the garbled circuit.

Upon receiving the encryptions, the CSP decrypts them and gets the masked values: $(i, (j, r_{ij}) \oplus \mu)$. Then, using the matrix factorization circuit as a blueprint, the CSP prepares a Yao's garbled circuit that

(a) takes as input the garbled values corresponding to the masks—this is denoted by $\text{GI}(\mu)$ on Figure 2;
(b) removes the mask μ to recover the corresponding tuple (i, j, r_{ij});
(c) performs matrix factorization; and
(d) outputs the item profiles V.

The CSP subsequently makes the garbled circuit available to the RecSys. Then, it engages in an oblivious transfer protocol with the RecSys so that the RecSys obtains garbled values of the masks: $\text{GI}(\mu)$. Finally, the RecSys evaluates the circuit, whose final (ungarbled) output comprises the requested profiles V.

We note that, in contrast to the solution presented in Appendix A, the circuit recovers (i, j, r_{ij}) by simply removing the mask through the XOR operation, rather than using decryption. Most importantly, as discussed in Section 5, XOR operations can be performed very efficiently in a garbled circuit implementation [34].

To complete the above protocol, we need to provide a circuit that implements matrix factorization. Before we discuss our design, we first describe a naïve solution below.

3.2 A Naïve Design

The gradient descent operations outlined in Eqs. (3)–(4) involve additions, subtractions and multiplications of real numbers. These operations can be efficiently implemented in a circuit [50]. The K iterations of gradient descent (3) correspond to K circuit "layers", each computing the new values of U and V from values in the preceding layer. The final output of the circuit are the item profiles V outputted the last layer, while the user profiles are discarded.

Observe that the time complexity of computing each iteration of gradient descent is $\Theta(M)$, when operations are performed in the clear, *e.g.*, in the RAM model: each gradient computation (4) involves adding $2M$ terms, and profile updates (3) can be performed in $\Theta(n + m) = \Theta(M)$ time.

The main challenge in implementing gradient descent as a circuit lies in doing so efficiently. To illustrate this, consider the following naïve implementation:

1. For each pair $(i, j) \in [n] \times [m]$, generate a circuit that computes from input the indicators $\delta_{ij} = \mathbb{1}_{(i,j)\in\mathcal{M}}$, which is 1 if i rated j and 0 otherwise.
2. At each iteration, using the outputs of these circuits, compute each item and user gradient as a summation over m and n products, respectively, where:

$$\nabla_{u_i} F(U, V) = -2 \textstyle\sum_{j\in[m]} \delta_{ij} \cdot v_j (r_{ij} - \langle u_i, v_j \rangle) + 2\lambda u_i,$$
$$\nabla_{v_j} F(U, V) = -2 \textstyle\sum_{i\in[n]} \delta_{ij} \cdot u_i (r_{ij} - \langle u_i, v_j \rangle) + 2\mu v_i.$$

Unfortunately, this implementation is inefficient: every iteration of the gradient descent algorithm has a circuit complexity of $\Theta(nm)$. When $M \ll nm$, as is usually the case in practice, the above circuit is drastically less efficient than gradient descent in the clear; in fact, the quadratic cost $\Theta(nm)$ is prohibitive for most datasets.

3.3 A Simple Counting Circuit

The inefficiency of the naïve implementation arises from the inability to identify which users rate an item and which items are rated by a user at the time of the circuit design, mitigating the ability to leverage the inherent sparsity in the data. The question that thus naturally arises is how to perform such a matching efficiently within a circuit.

We illustrate our main idea for performing this matching through a simple counting circuit. Let $c_j = |\{i : (i, j) \in \mathcal{M}\}|$

be the number of ratings item $j \in [m]$ received. Suppose that we wish to design a circuit that takes as input the set $\mathcal{M}$ and outputs the counts $\{c_j\}_{j\in[m]}$. This task's complexity in the RAM model is $\Theta(m+M)$, as all c_j can be computed simultaneously by a single pass over $\mathcal{M}$. In contrast, a naïve circuit implementation using "indicators", as in the previous section, yields a circuit complexity $\Theta(nm)$. Nevertheless, we show it is possible to construct a circuit that returns $\{c_j\}_{j\in[m]}$ in $\Theta\big((m+M)\log^2(m+M)\big)$ steps using a *sorting network* (see Appendix C).

We first describe the algorithm that performs this operation, and then discuss how we implement it as a circuit.

1. Given $\mathcal{M}$ as input, construct an array $\mathbf{S}$ of $m+M$ tuples. First, for each $j \in [m]$, create a tuple of the form $(j, \perp, 0)$, where the "null" symbol $\perp$ is a placeholder. Second, for each $(i,j) \in \mathcal{M}$, create a tuple of the form $(j, 1, 1)$, yielding:
$$\mathbf{S} = \begin{pmatrix} 1 & 2 & \dots & m & j_1 & j_2 & \dots & j_M \\ \perp & \perp & \dots & \perp & 1 & 1 & \dots & 1 \\ 0 & 0 & \dots & 0 & 1 & 1 & \dots & 1 \end{pmatrix} .$$
Intuitively, the first m tuples will serve as "counters", storing the number of ratings per item. The remaining M tuples contain the "input" to be counted. The third element in each tuple serves as a binary flag, separating counters from input.
2. Sort the tuples in increasing order w.r.t. the item ids, *i.e.*, the 1st element in each tuple. If two ids are equal, break ties by comparing tuple flags, *i.e.*, the 3rd elements in each tuple. Hence, after sorting, each "counter" tuple is succeeded by "input" tuples with the same id:
$$\mathbf{S} = \begin{pmatrix} 1 & 1 & \dots 1 & \dots & m & m & \dots & m \\ \perp & 1 & \dots 1 & \dots & \perp & 1 & \dots & 1 \\ 0 & 1 & \dots 1 & \dots & 0 & 1 & \dots & 1 \end{pmatrix} .$$
3. Starting from the right-most tuple, move from right to left, adding the values of the second entries in each tuple; if a counter tuple (*i.e.*, a zero flag) is reached, store the computed value at the $\perp$ entry, and restart the counting. More formally, denote by $s_{\ell,k}$ the ℓ-th element of the k-th tuple. This "right-to-left" pass amounts to the following assignments:
$$s_{2,k} \leftarrow s_{3,k} + s_{3,k+1} \cdot s_{2,k+1} \,, \tag{5}$$
for k ranging from $M+m-1$ down to 1.
4. Sort the array again in increasing order, this time w.r.t. the flags $s_{3,k}$. The resulting array's first m tuples contain the counters, which are released as output.

The above algorithm can be readily implemented as a circuit that takes as input $\mathcal{M}$ and outputs (j, c_j) for every item $j \in [m]$. Step 1 can be implemented as a circuit with input the tuples $(i,j) \in \mathcal{M}$ and output the initial array S, using $\Theta(m+M)$ gates. The sorting operations can be performed using, *e.g.*, Batcher's sorting network (*cf.* Appendix C) which takes as input the initial array and outputs the sorted array, requiring $\Theta((m+M)\log^2(m+M))$ gates. Finally, the right-to-left pass can be implemented as a circuit that performs (5) on each tuple, also with $\Theta(m+M)$ gates. Crucially, the pass is data-oblivious: (5) discriminates "counter" from "input" tuples through flags $s_{3,k}$ and $s_{3,k+1}$, but the same operation is performed on all k.

3.4 Our Efficient Design

Algorithm 1 Matrix Factorization Circuit

Input: Tuples (i, j, r_{ij})
Output: V

1: Initialize matrix $\mathbf{S}$
2: **Sort** tuples with respect to rows 1 and 3
3: **Copy** user profiles (left pass): for $k = 2 \dots M+n$
$$s_{\mathbf{5},k} \leftarrow s_{3,k} \cdot s_{\mathbf{5},k-1} + (1 - s_{3,k}) \cdot s_{\mathbf{5},k}$$
4: **Sort** tuples with respect to rows 2 and 3
5: **Copy** item profiles (left pass): for $k = 2 \dots M+m$
$$s_{\mathbf{6},k} \leftarrow s_{3,k} \cdot s_{\mathbf{6},k-1} + (1 - s_{3,k}) \cdot s_{\mathbf{6},k}$$
6: Compute the gradient contributions: $\forall k < M+m$
$$\begin{bmatrix} s_{\mathbf{5},k} \\ s_{\mathbf{6},k} \end{bmatrix} \leftarrow \begin{bmatrix} s_{3,k} \cdot 2\gamma s_{\mathbf{6},k}(s_{4,k} - \langle s_{\mathbf{5},k}, s_{\mathbf{6},k} \rangle) + (1 - s_{3,k}) \cdot s_{\mathbf{5},k} \\ s_{3,k} \cdot 2\gamma s_{\mathbf{5},k}(s_{4,k} - \langle s_{\mathbf{5},k}, s_{\mathbf{6},k} \rangle) + (1 - s_{3,k}) \cdot s_{\mathbf{6},k} \end{bmatrix}$$
7: **Update** item profiles (right pass): for $k = M+m-1 \dots 1$
$$s_{\mathbf{6},k} \leftarrow s_{\mathbf{6},k} + s_{3,k+1} \cdot s_{\mathbf{6},k+1} + (1 - s_{3,k}) \cdot 2\gamma\mu s_{\mathbf{6},k}$$
8: **Sort** tuples with respect to rows 1 and 3
9: **Update** user profiles (right pass): for $k = M+n-1 \dots 1$
$$s_{\mathbf{5},k} \leftarrow s_{\mathbf{5},k} + s_{3,k+1} \cdot s_{\mathbf{5},k+1} + (1 - s_{3,k}) \cdot 2\gamma\lambda s_{\mathbf{5},k}$$
10: **If** # of iterations is less than K, **goto** 3
11: **Sort** tuples with respect to rows 3 and 2
12: **Output** item profiles $s_{\mathbf{6},k}$, $k = 1, \dots, m$

Motivated by the above approach, we design a circuit for matrix factorization based on sorting, whose complexity is $\Theta((n+m+M)\log^2(n+m+M))$, *i.e.*, within a polylogarithmic factor of the implementation in the clear. The circuit operations are described in Algorithm 1. In summary, as in the simple counting example above, both the input data (the tuples (i,j,r_{ij})) and placeholders for both user and item profiles are stored together in an array. Through appropriate sorting operations, user or item profiles can be placed close to the input with which they share an identifier; linear passes through the data allow the computation of gradients, as well as updates of the profiles.

We again first describe the algorithm in detail and then discuss its implementation as a circuit. As before, the null symbol $\perp$ indicates a placeholder; when sorting, it is treated as $+\infty$, *i.e.*, larger than any other number.

Initialization. The algorithm receives as input the sets $L_i = \{(j, r_{ij}) : (i,j) \in \mathcal{M}\}$, and constructs an $n+m+M$ array of tuples $\mathbf{S}$. The first n and m tuples of $\mathbf{S}$ serve as placeholders for the user and item profiles, respectively, while the remaining M tuples store the inputs L_i. More specifically, for each user $i \in [n]$, the algorithm constructs a tuple $(i, \perp, 0, \perp, u_i, \perp)$, where $u_i \in \mathbb{R}^d$ is the initial profile of user i, selected at random from the unit ball. For each item $j \in [m]$, the algorithm constructs the tuple $(\perp, j, 0, \perp, \perp, v_j)$, where $v_j \in \mathbb{R}^d$ is the initial profile of item j, also selected at random from the unit ball. Finally, for each pair $(i,j) \in \mathcal{M}$, the corresponding tuple $(i, j, 1, r_{ij}, \perp, \perp)$, where r_{ij} is i's rating to j. The resulting array is shown in Figure 3(a).

$$\begin{pmatrix} 1: & 1 & \cdots & n & \bot & \cdots & \bot & i_1 & \cdots & i_M \\ 2: & \bot & \cdots & \bot & 1 & \cdots & m & j_1 & \cdots & j_M \\ 3: & 0 & \cdots & 0 & 0 & \cdots & 0 & 1 & \cdots & 1 \\ 4: & \bot & \cdots & \bot & \bot & \cdots & \bot & r_{i_1 j_1} & \cdots & r_{i_M j_M} \\ \mathbf{5}: & u_1 & \cdots & u_n & \bot & \cdots & \bot & \bot & \cdots & \bot \\ \mathbf{6}: & \bot & \cdots & \bot & v_1 & \cdots & v_m & \bot & \cdots & \bot \end{pmatrix}$$

(a) Initial state

$$\begin{pmatrix} 1: & 1 & 1\cdots 1 & \cdots & n & n\cdots n & \bot & \cdots & \bot \\ 2: & \bot & j_1\cdots j_{k_1} & \cdots & \bot & j_1\cdots j_{k_n} & 1 & \cdots & m \\ 3: & 0 & 1\cdots 1 & \cdots & 0 & 1\cdots 1 & 0 & \cdots & 0 \\ 4: & \bot & r_{1j_1}\cdots r_{1j_{k_1}} & \cdots & \bot & r_{nj_1}\cdots r_{nj_{k_n}} & \bot & \cdots & \bot \\ \mathbf{5}: & u_1 & \bot\cdots\bot & \cdots & u_n & \bot\cdots\bot & \bot & \cdots & \bot \\ \mathbf{6}: & \bot & \bot\cdots\bot & \cdots & \bot & \bot\cdots\bot & v_1 & \cdots & v_m \end{pmatrix}$$

(b) After sorting w.r.t. user ids

Figure 3: Data structure S used by Alg. 1. Fig. (a) indicates the initial state, and (b) shows the result after sorting w.r.t. the user ids, breaking ties through flags, as in Line 2 of Alg. 1. Bold rows 5, 6 correspond to d-dimensional (rather than scalar) values. Note that a left pass as in line 3 of Alg. 1 will copy user profiles to their immediately adjucent tuples.

We again denote by $s_{\ell,k}$ the ℓ-th element of the k-th tuple. Intuitively, these elements serve the following roles:

- $s_{1,k}$: user identifiers in $[n]$
- $s_{2,k}$: item identifiers in $[m]$
- $s_{3,k}$: a binary flag indicating if the tuple is a "profile" or "input" tuple
- $s_{4,k}$: ratings in "input" tuples
- $s_{\mathbf{5},k}$: user profiles in $\mathbb{R}^d$
- $s_{\mathbf{6},k}$: item profiles in $\mathbb{R}^d$

Gradient Descent. In brief, gradient descent iterations comprise of the following three steps:

1. **Copy profiles.** At each iteration, the profiles u_i, v_j of each user i and each item j are copied to the corresponding elements $s_{\mathbf{5},k}$ and $s_{\mathbf{6},k}$ of each "input" tuple in which i and j appear. This is implemented in Lines 2 to 5 of Algorithm 1. To copy, *e.g.*, the user profiles, **S** is sorted using the user id (*i.e.*, $s_{1,k}$) as a primary index and the flag (*i.e.*, $s_{3,k}$) as a secondary index. An example of such a sorting applied to the initial state of **S** can be found in Figure 3(b). Subsequently, the user ids are copied by traversing the array from left to right (a "left" pass), as described formally in Line 3. This copies $s_{\mathbf{5},k}$ from each "profile" tuple to its adjacent "input" tuples; item profiles are copied similarly.
2. **Compute gradient contributions.** After profiles are copied, each "input" tuple corresponding to, *e.g.*, (i, j) stores the rating r_{ij} (in $s_{4,k}$) as well as the profiles u_i and v_j (in $s_{\mathbf{5},k}$ and $s_{\mathbf{6},k}$, respectively), as computed in the last iteration. From these, the following are computed:

$$v_j(r_{ij} - \langle u_i, v_j\rangle), \text{ and } u_i(r_{ij} - \langle u_i, v_j\rangle)\,,$$

   which amount to the "contribution" of the tuple in the gradients w.r.t. u_i and v_j, as given by (4). These replace the $s_{\mathbf{5},k}$ and $s_{\mathbf{6},k}$ elements of the tuple, as indicated by Line 6. Through appropriate use of flags, this operation affects "input" tuples, leaving "profile" tuples unchanged.
3. **Update profiles.** Finally, the user and item profiles are updated, as shown in Lines 7 to 9. Through appropriate sorting, "profile" tuples are made again adjacent to the "input" tuples with which they share ids. The updated profiles are computed through a right-to-left traversing of the array (a "right pass"). This operation adds the contributions of the gradients as it traverses "input" tuples. Upon encountering a "profile" tuple, the summed gradient contributions are added to the profile, scaled appropriately. After passing a profile, the summation of gradient contributions restarts from zero, through appropriate use of the flags $s_{3,k}, s_{3,k+1}$.

The above operations are repeated K times, the number of desirable iterations of gradient descent.

Output. Finally, at the termination of the last iteration, the array is sorted w.r.t. the flags (*i.e.*, $s_{3,k}$) as a primary index, and the item ids (*i.e.*, $s_{2,k}$) as a secondary index. This brings all item profile tuples in the first m positions in the array, from which the item profiles can be extracted.

Each of the above operations is data-oblivious, and can be implemented as a circuit. Copying and updating profiles requires $\Theta(n+m+M)$ gates, so the overall complexity is determined by sorting, which yields a $\Theta((n+m+M)\log^2(n+m+M))$ cost when using Batcher's circuit. As we will see in Section 6, sorting and the gradient computation in Line 6 are the most computationally intensive operations; fortunately, both are highly paralellizable. In addition, sorting can be further optimized by reusing previously computed comparisons at each iteration. We discuss these and other optimizations in Section 5.3.

4. EXTENSIONS

4.1 Privacy-Preserving Recommendations

We now extend our design to a system that enables a user to learn her predicted ratings r_{ij}, for all $j \in [m]$, as given by (2). However, *neither the RecSys nor the CSP learn anything about the users* beyond how many ratings they generated; in particular, neither learns V. Again, these guarantees hold under the *honest-but-curious* threat model.

To implement this functionality, at the beginning of the protocol, each user i chooses a random mask ϑ_i (this mask will be used to hide user's i profile u_i), encrypts it under the CSP's public key using any semantically secure encryption scheme $\mathcal{E}$ and sends it to the RecSys. We denote by $\mathsf{t}_i = \mathcal{E}_{\mathsf{pk}_{\mathsf{csp}}}(\vartheta_i)$ the encrypted value.

The protocol then proceeds as described in Section 3 but with the following modifications. Initially, the RecSys forwards to the CSP the encrypted masks t_i ($i \in [n]$), which are then decrypted by the CSP. Hence the CSP knowns the plain value of the masks ϑ_i ($i \in [n]$). Likewise, on its side, the CSP chooses random masks ϱ_j for $j \in [m]$ (mask ϱ_j will be used to hide item profile v_j). The circuit built by the CSP again performs matrix factorization, as described in Section 3; however, rather than outputting $V = (v_j^T)_{j\in[m]}$, the circuit now outputs the item profiles masked with ϱ_j *and* the user profiles masked with ϑ_i:

$$\hat{v}_j = v_j + \varrho_j \quad \text{and} \quad \hat{u}_i = u_i + \vartheta_i$$

for $j \in [m]$ and $i \in [n]$. At the end of the protocol, the RecSys sends the respective $\hat{u}_i$ to each user i, who can then recover her profile u_i by removing the mask: $u_i = \hat{u}_i - \vartheta_i$.

The above execution, which is as computationally intensive as learning V in Section 3, can be performed as frequently as matrix factorization in real-life systems, *e.g.*, once a week.

In between such computations, whenever a user i wishes to get recommendations, she encrypts her profile under her own public key pk_i with an additively homomorphic encryption scheme $\mathfrak{E}_{\mathsf{pk}_i}$ (like Paillier's cryptosystem [52])[3] and sends the resulting value $\mathfrak{E}_{\mathsf{pk}_i}(u_i)$ to the RecSys. The RecSys forwards $\mathfrak{E}_{\mathsf{pk}_i}(u_i)$ to the CSP and also computes, for $j \in [m]$, $\mathfrak{E}_{\mathsf{pk}_i}(\langle u_i, \hat{v}_j \rangle)$. The CSP in turn computes for $j \in [m]$, $\mathfrak{E}_{\mathsf{pk}_i}(\langle u_i, \varrho_j \rangle)$, and returns this value to the RecSys. The RecSys subtracts $\mathfrak{E}_{\mathsf{pk}_i}(\langle u_i, \varrho_j \rangle)$ from $\mathfrak{E}_{\mathsf{pk}_i}(\langle u_i, \hat{v}_j \rangle) = \mathfrak{E}_{\mathsf{pk}_i}(\langle u_i, v_j + \varrho_j \rangle)$ to obtain the encryption of the predicted ratings $\hat{r}_{ij} = \mathfrak{E}_{\mathsf{pk}_i}(\langle u_i, v_j \rangle)$ for all items $j \in [m]$ and sends them to the user i. The user uses her private decryption key to obtain in the clear the predictions $r_{i1}, \ldots, r_{im}$.

4.2 Malicious RecSys

Our basic protocol, as described in Section 3, operates under the honest-but-curious model. However, a malicious RecSys can alter, duplicate or drop user ratings as a means to have the output model leak information about individual ratings. For example, the RecSys can provide inputs to the circuit from only a single user and feed dummy ratings for the remaining ones. It would then learn from the output model the set of items that were rated by this victim user: the RecSys simply observes which of the learned item profiles has a non-unit norm. This would clearly violate user privacy.

In order to prevent the RecSys from misbehaving, we need the CSP to build a circuit that, beyond outputting V, also verifies that the input to the circuit contains *all* user ratings, that the ratings were not changed, and that no dummy ratings were input. To do so we require users to obtain one-time MAC keys (one key per rating) from the CSP which they then use to sign their ratings. The CSP builds a circuit that verifies these MACs, making sure that the ratings were not altered, and that the exact number of ratings submitted by each user are provided as input to the circuit.

Our approach is to have each user first communicate with the CSP, reporting the number of ratings that she will send to the RecSys. The CSP in return sends the user a set of one-time MAC keys for each rating tuple, which the user uses for signing each tuple (j, r_{ij}). The garbled circuit the CSP builds verifies each input tuple with a specific MAC key, requiring the RecSys to provide the exact inputs as reported by the users to the CSP, sorted w.r.t. i and j. Any deviation from this order or the introduction of dummy ratings will result in a verification failure. Assume that the output of the verification circuit for each signed tuple is a bit that is set to 1 if the verification succeeds and 0 otherwise. The verification bits of all input tuples are fed into an AND gate, so that if at least one verification fails, the output of the AND gate is 0. If the outputs of this AND gate is 0 then the circuit sets its overall output to 0. This way, if at least one verification failed the circuit simply outputs 0. We chose to use fast one-time MACs based on pairwise independent hash functions so that the number of gates needed to verify these MACs is relatively small.

[3]Specifically, it is required that the encryptions of two messages $\mathfrak{E}_{\mathsf{pk}}(m_1)$ and $\mathfrak{E}_{\mathsf{pk}}(m_2)$ satisfy $\mathfrak{E}_{\mathsf{pk}}(m_1) \star \mathfrak{E}_{\mathsf{pk}}(m_2) = \mathfrak{E}_{\mathsf{pk}}(m_1 + m_2)$ for some binary operator $\star$. To ease notation, we use multiplication for $\star$ as is the case for Paillier encryption.

5. IMPLEMENTATION

We implemented our system to assess its practicality. Our garbled circuit construction was based on FastGC, a publicly available garbled circuit framework [24].

5.1 FastGC

FastGC [24] is a Java-based open-source framework, which enables circuit definition using elementary XOR, OR and AND gates. Once circuits are constructed, FastGC handles garbling, oblivious transfer and garbled circuit evaluation.

FastGC incorporates several known optimizations that aim to improve its memory foot-print and garbling and execution times. These optimizations include so-called "free" XOR gates [34], in which XOR evaluation is decryption-free and thereby of negligible cost, and garbled-row reduction for non-XOR gates [53], which reduces communication by 25%. FastGC also provides an "addition of 3 bits" circuit [33], enabling additions with 4 "free" XOR gates and one AND gate. OT extensions [27] are also implemented: these significantly increase the number of transfers made during OT, reducing communication overhead and lowering execution time.

Finally, FastGC enables the garbling and evaluation to take place concurrently on two separate machines. The CSP processes gates into garbled tables and transmits them to the RecSys in the order defined by circuit structure. Once a gate was evaluated its corresponding table is immediately discarded, which brings memory consumption caused by the garbled circuit to a constant.

5.2 Extensions to the Framework

Before garbling and executing the circuit, FastGC represents *the entire ungarbled circuit* in memory as a set of Java objects. These objects incur a significant memory overhead relative to the memory footprint of the ungarbled circuit, as only a subset of the gates is garbled and/or executed at any point in time. Moreover, although FastGC performs garbling in parallel to the execution process as described above, both operations occur in a sequential fashion: gates are processed one at a time, once their inputs are ready. Clearly, this implementation is not amenable to parallelization.

We modified the framework to address these issues, reducing the memory footprint of FastGC but also enabling paralellized garbling and computation across multiple processors. In particular, we introduced the ability to partition a circuit horizontally into sequential "layers", each one comprising a set of vertical "slices" that can be executed in parallel. A layer is created in memory only when all its inputs are ready. Once it is garbled and evaluated, the entire layer is removed from memory, and the following layer can be constructed, thus limiting the memory footprint to the size of the largest layer. The execution of a layer uses a scheduler that assigns its slices to threads, which run in parallel. Although we implemented parallelization on a single machine with multiple cores, our implementation can be extended to run across different machines in a straightforward manner since no shared state between slices is assumed.

Finally, to implement the numerical operations outlined in Algorithm 1, we extended FastGC to support addition and multiplications over the reals with fixed-point number representation, as well as sorting. For sorting, we used Batcher's sorting network [3]. Fixed-point representation introduces a tradeoff between the accuracy loss resulting from truncation and the size of circuit, which we explore in Section 6.

5.3 Optimizing Algorithm 1

We optimize the implementation of Algorithm 1 in multiple ways. In particular, we (a) reduce the cost of sorting by reusing comparisons computed in the beginning of the circuit's execution, (b) reduce the size of array S, (c) optimize swap operations by using XORs, and (d) parallelize computations. We describe these optimizations in detail below.

Comparison Reuse. As described in Appendix C, the basic building block of a sorting network is a compare-and-swap circuit, that compares two items and swaps them if necessary, so that the output pair is ordered. Observe that the sorting operations (Lines 4 and 8) of Algorithm 1 perform identical comparisons between tuples at each of the K gradient descent iterations, using exactly the same inputs per iteration. In fact, each sorting permutes the tuples in array S in exactly the same manner, at each iteration.

We exploit this property by performing the comparison operations for each of these sortings *only once*. In particular, we perform sortings of tuples of the form $(i, j, \mathit{flag}, \mathit{rating})$ in the beginning of our computation (without the payload of user or item profiles), *e.g.*, w.r.t. i and the flag first, j and the flag, and back to i and the flag. Subsequently, we reuse the outputs of the comparison circuits in each of these sortings as input to the swap circuits used during gradient descent. As a result, the "sorting" network applied at each iteration does not perform any comparisons, but simply permutes tuples (*i.e.*, it is a "permutation" network).

Row Reduction. Precomputing all comparisons allows us to also drastically reduce the size of tuples in S. To begin with, observe that the rows corresponding to user or item ids are only used in Algorithm 1 as input to comparisons during sorting. Flags and ratings are used during copy and update phases, but their relative positions are identical at each iteration. Moreover, these positions are produced as outputs when sorting the tuples $(i, j, \mathit{flag}, \mathit{rating})$ at the beginning of our computation. As such, "permutation" operations performed at each iteration need only be applied to user and item profiles; all other rows are removed from S.

One more trick reduces the cost of permutations by an additional factor of 2. We fix one set of profiles, *e.g.*, users, and permute only item profiles. Then, item profiles rotate between two states, each one reachable from the other through permutation: one in which they are aligned with user profiles and partial gradients are computed, and one in which item profiles are updated and copied.

XOR Optimization. Given that XOR operations can be executed for "free", we optimize comparison, swap, update and copying operations by using XORs wherever possible. For comparisons, we reduce the use of AND and OR gates using a technique by Kolesnikov *et al.* [33]. Swap operations are implemented as follows: for $b \leftarrow x > y$ the comparison bit between tuples x and y (which, by the above optimizations, is pre-computed at the beginning of circuit execution), a swap is performed as:

$$x' \leftarrow [b \wedge (x \oplus y)] \oplus x, \text{ and } y' \leftarrow x' \oplus (x \oplus y)$$

Finally, copy operations are also optimized to use XOR's. Observe that the copy operation takes two elements x and y and a flag s and outputs a new element y' which is equals y if $s = 0$ and x, otherwise. This is performed as follows:

$$x' \leftarrow y \oplus [s \wedge (x \oplus y)]$$

Parallelization. As discussed in Section 3.4, sorting and gradient computations constitute the bulk of the computation in our circuit (copying and updating contribute no more than 3% of the execution time and 0.4% of the non-XOR gates); we parallelize these operations through our extension of FastGC. Gradient computations are clearly parallelizable; sorting networks are also highly parallelizable (parallelization is the main motivation behind their development). Moreover, since many of the parallel slices in each sort are identical, we reused the same FastGC objects defining the circuit slices with different inputs, significantly reducing the need to repeatedly create and destroy objects in memory.

6. EXPERIMENTS

We now assess the performance of our implementation. We use two commodity servers, 1.9GHz 16-cores 128GB RAM each, one acting as the RecSys and the other as the CSP. We use both real and synthetic datasets. For the real dataset we use MovieLens, a movie rating dataset that is commonly used for recommender systems research, that consists of 943 users that submit 100K ratings to 1682 movies.

We use the following evaluation metrics. Our solution introduces inaccuracies due to our use of fixed-point representation of real numbers. Thus, our goal here is to understand the relative error of our approach compared to a system that operates in the clear with floating point representation. Let $E(U, V)$ denote the squared error for a given user's profile U and items profiles V, $E(U, V) = \sum_{(i,j)\in\mathcal{M}}(r_{ij} - u_i^T v_j)^2$; we define the *relative error* as

$$|E(U^*, V^*) - E(U, V)|/E(U, V)$$

where U^* and V^* are computed using our solution and U and V are computed using gradient descent executed in the clear over floating point arithmetic, *i.e.*, with minimal precision loss. Our *time* metric captures the execution time needed to garble and evaluate the circuit. We note that we exclude the encryption and decryption times performed by the users and the CSP, since these are short in duration compared to the circuit processing time. The *communication* metric is defined by the number of bytes that are transmitted between the CSP and RecSys; it captures the size of the circuit, namely the number of non-XOR gates, but unlike the time metric, communication is not affected by parallelization.

The relative error of our solution using the complete MovieLens dataset is shown in Figure 4. We study the relative error over a range of parameters, varying the number of bits allocated to the fractional part of the fixed point representation and the number of iterations of gradient descent. Overall we see that for more than 20 bits, the relative errors are very low. When the number of bits is small, the gradient descent method may converge to a different local minimum of (1) than the one reached in the clear. Beyond 20 bits allocated for the fractional part, our solution converges to the same local minimum, and errors decrease exponentially with additional bits. The relative error increases with the number of iterations because the errors introduced by the fixed point representations accumulate across the iterations, however this increase is very small. We note that this should not be confused with the regularized least square error (1), which actually decreases when using more iterations.

In the following experiments we used synthetic data with 100 users, 100 items, a dimension of 10 for the user and item profiles, 20 bits for the fractional part of the fixed-point

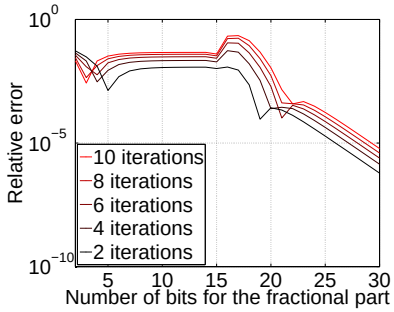


Figure 4: Relative errors due to fixed point representation

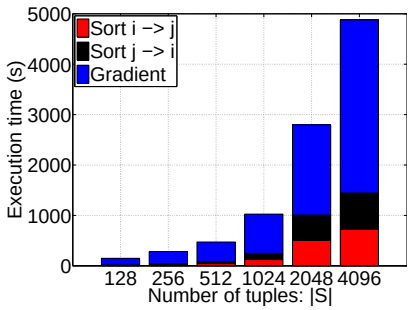


Figure 5: Execution time per iteration w.r.t. no. of tuples

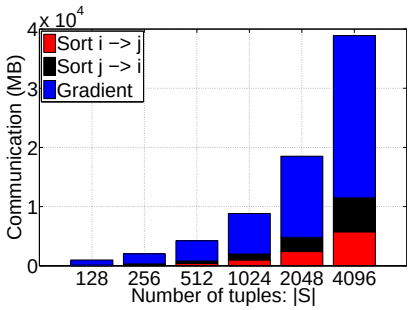


Figure 6: Communication cost per iteration w.r.t. no. of tuples

representation (36 bits overall). We measure the time and communication it takes to perform (garble and evaluate) a single iteration of gradient descent. Clearly with T iterations execution and communication grow by a factor T.

Figure 5 shows the increase in time per iteration as we increase the number of ratings in the dataset (the logarithmic x-axis corresponds to the number of tuples in $\mathbf{S}$ that grows with $\mathcal{M}$ since n and m are fixed). The plot also illustrates the proportion of time spent in various sections of our algorithm. We note that, in all executions, the time spent on update and copy phases, which are more difficult to parallelize, never exceeded 3%, and thus is omitted from the plots as it is not visible.

The plot confirms that the growth is almost linear with the number of ratings, $\Theta(M \log^2 M)$. Furthermore, we observe that more than 2/3 of the execution time is spent on gradient computations (mainly due to vector multiplication operations), while the remaining 1/3 is due to sorting operations. As both operations are highly parallelizable, this illustrates that the execution time can be significantly reduced through parallelization.

Similarly, Figure 6 plots the amount of bytes communicated between the CSP and RecSys for garbling and evaluating a single iteration. The plot shows the same properties as the execution time, namely that the size scales almost linearly with the number of ratings and that the majority of the circuit is devoted to gradient computations. In all implementations, copy and update operations did not contribute more that 0.4% of the gates in the circuit.

As an example for the time and communication performance for a real dataset, we limited our MovieLens dataset to the 40 most popular movies. This corresponds to 14683 ratings generated by 940 users. One iteration of gradient descent with parameters set to achieve error of 10^{-4} took 2.9hr. These experiments were performed on a machine with 16 cores; real-life systems use much more powerful hardware (*e.g.*, hundreds of Amazon EC2 servers). Moreover, operations are highly parallelizable. As such, with access to industry-level equipment this timing can be brought to the realm of practicallity, especially given that recommender systems run matrix factorization on, *e.g.*, a weekly basis.

7. RELATED WORK

Secure multiparty computation (MPC) was initially proposed by Yao [62, 63]. There are presently many frameworks that implement Yao garbled circuits [45, 23, 24, 44, 53, 25, 36]. A different approach to general purpose MPC is based on secret-sharing schemes and another is based on fully-homomorphic encryption (FHE). Secret-sharing schemes have been proposed for a variety of linear algebra operations, such as solving a linear system [51], linear regression [29, 30, 21], and auctions [10]. Secret-sharing requires at least three non-colluding online authorities that equally share the workload of the computation, and communicate over multiple rounds; the computation is secure as long as no two of them collude. Garbled circuits assumes only *two* non-colluding authorities and far less communication which is better suited to the scenario where the RecSys is a cloud service and the CSP is implemented in a trusted hardware component. Non-linear computation through fully homomorphic encryption [16] may be used to reduce the workload on the CSP compared to garbled circuits, but current FHE schemes [39, 20] for simpler algebraic computations are not as efficient as garbled circuit approaches [50].

Centralized garbled-circuit computation of a function over a large number of individual inputs was introduced by Naor *et al.* in the context of auctions [48]. Our approach is closest to the privacy-preserving regression computation in [50], though implementing matrix factorization efficiently as a circuit introduces challenges not present in regression. Beyond [50], hybrid approaches combining garbled circuits with other methods (such as HE or secret-sharing) have been used for, *e.g.*, face and fingerprints recognition [57, 26], and learning a decision tree [41]; such discrete function evaluations differ considerably from matrix factorization.

Irrespective of the cryptographic primitive used, the main challenge in building an efficient algorithm for secure multiparty computation is in implementing the algorithm in a *data-oblivious* fashion, *i.e.*, so that the execution path does not depend on the input. In general, any RAM program executable in bounded time T can be converted to a $O(T^3)$ Turing machine [8], and any bounded T-time TM can be converted to a circuit of size $O(T \log T)$ [54], which is data-oblivious. This results in a $O(T^3 \log T)$ complexity, which is prohibitive in most applications. A survey of algorithms for which efficient data-oblivious implementations are unknown can be found in [11]: matrix factorization broadly falls into the category of Data Mining summarization problems.

Sorting networks were originally developed to enable sorting parallelization as well as an efficient hardware implementation. Several recent works exploit the data-obliviousness of sorting networks for cryptographic purposes which, in turn, has lead to renewed interest in oblivious sorting protocols beyond sorting networks (*e.g.*, [18, 22]). There are many recent data-oblivious algorithms using sorting as a building

block, including compaction and selection [60, 19], the computation of a convex hull and all-nearest neighbors [13], as well as weighted set intersection [28]; the simple counting protocol in Section 3.3 is a variation upon these schemes. Nevertheless, these operations are much simpler than matrix factorization; to the best of our knowledge, we are the first to apply oblivious sorting on such a numerical task.

Privacy in recommender systems has been studied under several contexts, including the use of trusted hardware [1] as well as the susceptibility of a system to shilling attacks (*i.e.*, the injection of false ratings to manipulate the recommendation outcome) [38, 47]. An approach orthogonal to ours that introduces privacy in recommender systems is differential privacy [12, 46]. By adding noise, differential privacy guarantees that the distribution of the system's output is insensitive to any individual's record, preventing the inference of any single user's data from the output. However, differential privacy does not protect data from the recommender system itself. Crucially, differential privacy can be combined with secure computation [58], in our case by incorporating noise addition within the garbled circuit factorizing the input matrix. Differential privacy can thus be used to enhance the privacy properties of our protocol, ensuring not only that the data remains private during computation, but also the final result does not expose individual user data.

8. CONCLUSIONS AND FUTURE WORK

We presented a protocol for matrix factorization on user ratings that remain encrypted at all times. This critical building block allows a recommender to learn item profiles without learns *anything* about users' ratings, providing users protection from inference threats and accidental information leakage. Our hybrid approach combines partially homomorphic encryption and Yao's garbled circuits. To the best of our knowledge, we are the first to apply oblivious sorting to a numerical task as complex as matrix factorization. Through this key idea, that also enables us to highly parallelize our implementation, we overcome scalability and performance needs, and bring matrix factorization on encrypted data into the realm of practicality.

There are several future directions for this work. First, we hope to deploy our system over a cloud compute service (*e.g.*, using Hadoop on Amazon EC2), which will enable an increase in the range of datasets that we can process. A second direction is to investigate the application of our approach to other equally intensive machine learning tasks, especially ones that exhibit an underlying bipartite structure in computations; we could thus leverage sorting networks again to achieve performance scalability.

A third direction is to extend our protocol to work under different security models, *e.g.*, a malicious CSP. A malicious CSP can create an incorrect circuit, which can be handled with standard techniques for verifying garbled circuits [43, 40]. Moreover, it can feed the wrong inputs to the circuit, *e.g.*, maliciously altered masked values as described in Section 3.1. The latter attack reveals no additional information to the CSP, but it may corrupt the result of the computation. Therefore additional techniques should be designed to ensure that either the CSP provided the correct inputs to the circuit or that the output of the recommendation circuit closely approximates the ratings provided by users.

9. REFERENCES

[1] E. Aïmeur, G. Brassard, J. M. Fernandez, and F. S. M. Onana. ALAMBIC: A privacy-preserving recommender system for electronic commerce. *Int. J. Inf. Sec.*, 7(5), 2008.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, 1983.

[3] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, 1968.

[4] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *CRYPTO*, 1990.

[5] E. J. Candès and B. Recht. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6), 2009.

[6] J. F. Canny. Collaborative filtering with privacy. In *IEEE S&P*, 2002.

[7] B. Chevallier-Mames, P. Paillier, and D. Pointcheval. Encoding-free ElGamal encryption without random oracles. In *PKC*, 2006.

[8] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *J. Computer and System Sciences*, 1973.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[10] I. Damgård and T. Toft. Trading sugar beet quotas - secure multiparty computation in practice. *ERCIM News*, 2008.

[11] W. Du and M. J. Atallah. Secure multi-party computation problems and their applications: A review and open problems. In *New Security Paradigms Workshop*, 2001.

[12] C. Dwork. Differential privacy. In *ICALP*, 2006.

[13] D. Eppstein, M. T. Goodrich, and R. Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *18th SIGSPATIAL*, 2010.

[14] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6), 1985.

[15] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, 2nd edition, 2009.

[16] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[17] S. Goldwasser and M. Bellare. *Lecture Notes on Cryptography*. MIT, 2001.

[18] M. T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *SODA*, 2010.

[19] M. T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, 2011.

[20] T. Graepel, K. Lauter, and M. Naehrig. ML confidential: Machine learning on encrypted data. Cryptology ePrint Archive, Report 2012/323, 2012.

[21] R. Hall, S. E. Fienberg, and Y. Nardi. Secure multiple linear regression based on homomorphic encryption. *J. Official Statistics*, 2011.

[22] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, 2013.

[23] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for automating secure two-party computations. In *CCS*, 2010.

[24] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[25] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE S&P*, 2012.

[26] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[27] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, 2003.

[28] K. V. Jónsson, G. Kreitz, and M. Uddin. Secure multi-party sorting and applications. Cryptology ePrint Archive, Report 2011/122, 2011.

[29] A. F. Karr, W. J. Fulp, F. Vera, S. S. Young, X. Lin, and J. P. Reiter. Secure, privacy-preserving analysis of distributed databases. *Technometrics*, 2007.
[30] A. F. Karr, X. Lin, A. P. Sanil, and J. P. Reiter. Privacy-preserving analysis of vertically partitioned data using secure matrix products. *J. Official Statistics*, 2009.
[31] R. H. Keshavan, A. Montanari, and S. Oh. Learning low rank matrices from $O(n)$ entries. In *Allerton*, 2008.
[32] D. E. Knuth. *The Art Of Computer Programming — Volume 3 / Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.
[33] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, 2009.
[34] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.
[35] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 2009.
[36] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, 2012.
[37] S. K. Lam, D. Frankowski, and J. Riedl. Do you trust your recommendations? An exploration of security and privacy issues in recommender systems. In *ETRICS 2006*, 2006.
[38] S. K. Lam and J. Riedl. Shilling recommender systems for fun and profit. In *WWW*, 2004.
[39] K. Lauter, M. Naehrig, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, 2011.
[40] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. *IACR Cryptology ePrint Archive*, 2013.
[41] Y. Lindell and B. Pinkas. Privacy preserving data mining. *J. Cryptology*, 2002.
[42] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 2009.
[43] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *J. Cryptology*, 2012.
[44] Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN*, 2008.
[45] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – Secure two-party computation system. In *USENIX Security*, 2004.
[46] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the Netflix prize contenders. In *KDD*, 2009.
[47] B. Mobasher, R. Burke, R. Bhaumik, and C. Williams. Toward trustworthy recommender systems: An analysis of attack models and algorithm robustness. *ACM Trans. Internet Techn.*, 7(4), 2007.
[48] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *1st ACM Conference on Electronic Commerce*, 1999.
[49] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE S&P*, 2008.
[50] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P*, 2013.
[51] K. Nissim and E. Weinreb. Communication efficient secure linear algebra. In *TCC*, 2006.
[52] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.
[53] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, 2009.
[54] N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. ACM*, 26(2), 1979.
[55] M. O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Aiken Computation Laboratory, Harvard University, 1981.
[56] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), 2009.
[57] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, 2009.
[58] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *NDSS*, 2011.
[59] Y. Tsiounis and M. Yung. On the security of ElGamal based encryption. In *PKC*, 1998.
[60] G. Wang, T. Luo, M. T. Goodrich, W. Du, and Z. Zhu. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In *CCS*, 2010.
[61] U. Weinsberg, S. Bhagat, S. Ioannidis, and N. Taft. BlurMe: Inferring and obfuscating user gender based on ratings. In *RecSys*, 2012.
[62] A. C.-C. Yao. Protocols for secure computations. In *FOCS*, 1982.
[63] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

APPENDIX

A. YAO'S GARBLED CIRCUITS

Yao's protocol (a.k.a. *garbled circuits*) [63] (see also [42]) is a generic method for secure multi-party computation. In a variant thereof (adapted from [48, 50]), the protocol is run between a set of n input owners, where a_i denotes the private input of user i, $1 \leq i \leq n$, an evaluator, that wishes to evaluate $f(a_1, \ldots, a_n)$, and a third party, the crypto-service provider or CSP in short. At the end of the protocol, the evaluator learns the value of $f(a_1, a_2, \ldots, a_n)$ but no party learns more than what is revealed from this output value. The protocol requires the function f can be expressed as a Boolean circuit, *e.g.* as a graph of OR, AND, NOT and XOR gates, and that the evaluator and the CSP do not collude.

Oblivious Transfer. *Oblivious transfer* (OT) [55, 14] is an important building block of Yao's protocol. OT is two-party protocol between a chooser and a sender. The sender has two ℓ-bit strings σ_0 and σ_1. The chooser selects a bit b and *exactly* obtains from the sender the string σ_b, *without the sender learning the value of* b. In addition, the chooser *learns nothing about* σ_{1-b} (beyond its length).

Oblivious transfer protocols can be constructed from many cryptographic assumptions. We describe below a protocol based on the Decision Diffie-Hellman assumption [4].

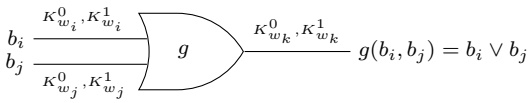
Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of order q in which the decisional Diffie-Hellman (DDH) assumption holds. Let also Ω be an encoding map from $\{0,1\}^\ell$ onto $\mathbb{G}$. Finally, let $c \in \mathbb{G}$ whose discrete logarithm is unknown. The chooser chooses $x \in_R \mathbb{Z}_q$ and computes $y_b = g^x$ and $y_{1-b} = c/g^x$. She sends y_0 to the sender. The sender represents σ_0 and σ_1 as elements in $\mathbb{G}$: $\omega_0 = \Omega(\sigma_0)$ and $\omega_1 = \Omega(\sigma_1)$. She chooses $r_0, r_1 \in_R \mathbb{Z}_q$, recovers $y_1 = c/y_0$, and computes $C_0 = (g^{r_0}, \omega_0\, {y_0}^{r_0})$ and $C_1 = (g^{r_1}, \omega_1\, {y_1}^{r_1})$. The sender sends C_0, C_1 to the chooser. Upon receiving C_0, C_1, the chooser computes ω_b as $\omega_b\, {y_b}^{r_b}/(g^{r_b})^x$ using secret value x and obtains $\sigma_b = \Omega^{-1}(\omega_b)$.

Circuit Garbling. The key idea behind Yao's protocol resides in the circuit encoding. To each wire w_i of the circuit, the CSP associates two random cryptographic keys, $K^0_{w_i}$ and $K^1_{w_i}$, that respectively correspond to the bit-values $b_i = 0$ and $b_i = 1$. Next, for each binary gate g (*e.g.*, an OR-gate) with input wires (w_i, w_j) and output wire w_k, the CSP

computes the four ciphertexts

$$\mathrm{Enc}_{(K_{w_i}^{b_i}, K_{w_j}^{b_j})}(K_{w_k}^{g(b_i,b_j)}) \quad \text{for } b_i, b_j \in \{0,1\} \ . \tag{6}$$

The set of these four *randomly ordered* ciphertexts defines the garbled gate. See [45] for an efficient implementation. For example, as illustrated on Fig. 7, given the pair of keys $(K_{w_i}^0, K_{w_j}^1)$ it is possible to recover the key $K_{w_k}^1$ by decrypting $\mathrm{Enc}_{(K_{w_i}^0, K_{w_j}^1)}(K_{w_k}^1)$. However, the other key, namely $K_{w_k}^0$, cannot be recovered. More generally, it is worth noting that the knowledge of $(K_{w_i}^{b_i}, K_{w_j}^{b_j})$ yields only the value of $K_{w_k}^{g(b_i,b_j)}$ and that no other output values can be recovered for the corresponding gate.



b_i	b_j	$g(b_i,b_j)$	Garbled value
0	0	0	$\mathrm{Enc}_{(K_{w_i}^0, K_{w_j}^0)}(K_{w_k}^0)$
0	1	1	$\mathrm{Enc}_{(K_{w_i}^0, K_{w_j}^1)}(K_{w_k}^1)$
1	0	1	$\mathrm{Enc}_{(K_{w_i}^1, K_{w_j}^0)}(K_{w_k}^1)$
1	1	1	$\mathrm{Enc}_{(K_{w_i}^1, K_{w_j}^1)}(K_{w_k}^1)$

Figure 7: Example of a garbled OR-gate.

Circuit Evaluation. We are now ready to present the complete protocol for evaluating f. The CSP generates a private and public key, and makes the latter available to the users. Each user i encrypts her private input a_i under the CSP's public key to get $\mathfrak{c}_i$, and sends $\mathfrak{c}_i$ to the evaluator. Upon receiving all encrypted inputs, the evaluator contacts the CSP to build a garbled circuit performing the steps of (a) decrypting the encrypted input values, using the CSP's private key and (b) evaluating function f.

The CSP provides the evaluator with the garbled gates of this circuit, each comprising a random permutation of the ciphertexts (6), as well as the graph representing how these connect. It also provides the correspondence between the garbled value and the real bit-value for the circuit-output wires (the outcome of the computation): if w_k is an circuit-output wire, the pairs $(K_{w_k}^0, 0)$ and $(K_{w_k}^1, 1)$ are given to the evaluator. To transfer the garbled values of the input wires, the CSP engages in an oblivious transfer with the evaluator, so that evaluator obliviously obtains the garbled-circuit input values corresponding to the $\mathfrak{c}_i$'s; this ensures that the CSP does not learn the user inputs and that the evaluator can only compute the function on these inputs alone. Having the garbled inputs, the evaluator can "evaluate" each gate sequentially, by decrypring gate and obtaining the keys necessary to decrypt the output of the gates it connects to.

B. HASH-ELGAMAL ENCRYPTION

Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of order q and $H : \mathbb{G} \to \{0,1\}^\ell$ be a cryptographic hash function. The public key is $\mathsf{pk} = (g, y)$ where $y = g^x$ for some random $x \in \mathbb{Z}_q$, and the private key is $\mathsf{sk} = x$. A message $m \in \{0,1\}^\ell$ is encrypted using

$$\mathcal{E}_{\mathsf{pk}} : \{0,1\}^\ell \to \mathbb{G} \times \{0,1\}^\ell, m \mapsto \mathfrak{c} = \big(g^\rho, m \oplus H(y^\rho)\big)$$

for some random $\rho \in \mathbb{Z}_q$. Letting $\mathfrak{c} = (\mathfrak{c}^{(1)}, \mathfrak{c}^{(2)})$, it is worth remarking that one can publicly mask the ciphertext $\mathfrak{c}$ with any chosen random mask $\mu \in \{0,1\}^\ell$ as

$$\hat{\mathfrak{c}} = (\mathfrak{c}^{(1)}, \mathfrak{c}^{(2)} \oplus \mu) \ .$$

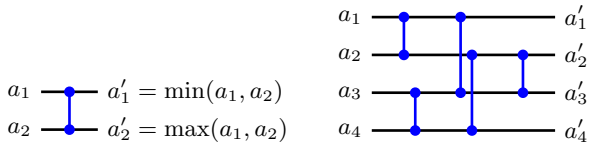
Decrypting $\hat{\mathfrak{c}} = (\hat{\mathfrak{c}}^{(1)}, \hat{\mathfrak{c}}^{(2)})$ then yields the masked message $\hat{m} = m \oplus \mu$. Indeed, we have

$$\begin{aligned} \hat{\mathfrak{c}}^{(2)} \oplus H\big((\hat{\mathfrak{c}}^{(1)})^x\big) &= (\mathfrak{c}^{(2)} \oplus \mu) \oplus H\big((g^\rho)^x\big) \\ &= \big((m \oplus H(y^\rho)) \oplus \mu\big) \oplus H(y^\rho) \\ &= m \oplus \mu \ . \end{aligned}$$

The scheme can be shown to be semantically secure in the random oracle model, under the Decision Diffie-Hellman assumption [59].

C. SORTING NETWORKS

Sorting networks [9, 32] are circuits that sort an input sequence $(a_1, a_2, \ldots, a_n)$ into a monotonically increasing sequence $(a'_1, a'_2, \ldots, a'_n)$. They are constructed by wiring together *compare-and-swap* circuits, their main building block. A *compare-and-swap* circuit is a binary operator taking on input a pair (a_1, a_2), and returning the sorted pair (a'_1, a'_2) where $a'_1 = \min(a_1, a_2)$ and $a'_2 = \max(a_1, a_2)$. For graphical convenience, a comparator is usually represented as a vertical line, as illustrated in Figure 8(a). Note that elements are swapped if and only if the first element is larger than the second one. Figure 8(b) shows a sorting network example.



(a) Compare-and-Swap circuit (b) Sorting network

Figure 8: Networks of compare-and-swap elements.

Sorting networks were specifically designed to admit an efficient hardware implementation, but also to be highly parallelizable. The efficiency of the sorting network can be measured by its size (the total number of comparisons) or depth (the maximum number of stages, where each stage comprises comparisons that can be executed in parallel). The depth of the network reflects the parallel running time of the sorting.

For example, in Figure 8(b) the comparisons (a_1, a_2) and (a_3, a_4) can be executed in parallel; so can comparisons (a_1, a_3) and (a_2, a_4). As such, the depth of this network is 3, which is the maximum number of compare-and-swaps along each "line". The network can be computed in 3 timesteps with 2 processors or in 5 timesteps with just one processor.

The best known (and asymptotically optimal) sorting network is the AKS network [2] that achieves size $O(n \log n)$ and depth $O(\log n)$. Being an important theoretical discovery, the AKS network has no pratical application because of a large constant. Efficient networks that are often used in practice achieve depth $O(\log^2 n)$ and size $O(n \log^2 n)$. These include Batcher, odd-even merge sort, bitonic sort, and Shell sort networks [32]. In the presence of p processors, the running time of these networks is $O(n(\log^2 n/p)$. Empirical studies indicate that, in practice, Batcher has better average performance than most widely used algorithms [60].