Proceedings on Privacy Enhancing Technologies ; 2021 (4):30–53

Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry

You May Also Like... Privacy: Recommendation Systems Meet PIR

Abstract: We describe the design, analysis, implementation, and evaluation of PIRSONA, a digital content delivery system that realizes collaborative-filtering recommendations atop private information retrieval (PIR). This combination of seemingly antithetical primitives makes possible—for the first time—the construction of practically efficient e-commerce and digital media delivery systems that can provide personalized content recommendations based on their users' historical consumption patterns while simultaneously keeping said consumption patterns private. In designing PIRSONA, we have opted for the most performant primitives available (at the expense of rather strong non-collusion assumptions); namely, we use the recent computationally 1-private PIR protocol of Hafiz and Henry (PETS 2019.4) together with a carefully optimized 4PC Boolean matrix factorization.

Keywords: Multiparty computation; distributed point functions; private information retrieval; recommendation systems

DOI 10.2478/popets-2021-0059
Received 2021-02-28; revised 2021-06-15; accepted 2021-06-16.

1 Introduction

Over the last two decades, we have witnessed a surge in electronic commerce and digital media streaming, with ubiquitous video streaming services like *Netflix*, *Hulu*, *YouTube*, and *Amazon Instant Video* overtaking cable and DVDs as the *de facto* source for television shows and films; audio streaming services like *Spotify*, *Apple Music*, *Last.fm*, and *Pandora* supplanting CDs and radio as the *de facto* choice for music and commentary; app stores like *Google Play*, Apple's *App Store*, the *Microsoft Store*, and the *Amazon Appstore* becoming the *de facto* way to download software; and e-book marketplaces like the *Amazon Kindle Store*, *Google eBooks*, and Barnes and Noble's *Nook Store* replacing brick-and-mortar bookstores as the *de facto* way to buy books.

With this shift to the digital, on-demand distribution of media comes increased convenience and cost savings for consumers—and an unprecedented capacity for content distributors to amass enormous quantities of fine-grain consumption-pattern data. And there's the rub: While content distributors can (and do) leverage the novel insights gleaned from such data to fuel recommendation engines and inform content creators in ways that benefit consumers, they also can (and do) sell or otherwise exploit said data for uses that are decidedly *not* in the consumers' best interests. Indeed, such data is invaluable to unscrupulous marketers and identity thieves seeking to defraud, law enforcement officers seeking to implicate or entrap, and hostile foreign agents seeking to manipulate and sow discord in democratic processes.

Given this tension between the utility of consumption-patterns on one hand and the sensitivity of consumption-patterns on the other, we ask the following question:

> *Is it possible to construct scalable, practically efficient digital content–distribution services that provide personalized content recommendations based on users' consumption patterns, all the while keeping those consumption patterns private?*

In this paper, we answer the preceding question in the affirmative by constructing PIRSONA, a novel system that uses (semi-honest) secure 4-party computation to implement collaborative-filtering recommendations atop a private information retrieval (PIR) protocol. This is no trivial task, as the privacy guarantees of PIR—allowing users to fetch items from a remote database while hiding from the data holder(s) which items they fetch—appear to be at fundamental odds with the goal of collaborative filtering—making automatic predictions about the interests and preferences of a user based on the interests and preferences of other like-minded users.

Adithya Vadapalli: Indiana University, avadapal@indiana.edu
Fattaneh Bayatbabolghani: University of California, Berkeley, fattaneh.bayat@gmail.com
Ryan Henry: University of Calgary, ryan.henry@ucalgary.ca

1.1 System overview

At a high level, our construction works as follows. Several *content distributors* hold identical copies of some database of fetchable records. To maintain their privacy, the users fetch records of interest using a fast, "computationally 1-private" multiserver PIR protocol (i.e., PIR that protects against a single malicious server). In addition to servicing the users' fetch requests, the servers store per-user, secret-shared consumption histories, which they extract directly from the users' incoming PIR queries. Periodically, the servers transform these consumption histories into (still secret-shared) collaborative filtering item and user profiles using a bespoke 4PC realization of Boolean matrix factorization. The latter secret-shared profiles are thenceforth used to render oblivious, yet personalized content recommendations for PIRSONA users.

1.2 Contributions

Our main contribution is the design, analysis, implementation, and evaluation of PIRSONA, a novel system that realizes collaborative filtering recommendations atop multiserver PIR. PIRSONA is the first system enabling untrusted content distributors to serve users obliviously while maintaining the distributors' ability to train recommender engines with which to provide oblivious, yet personalized content recommendations. Unlike in prior work on privacy-preserving machine learning [36, 37] or targeted advertising [29, 30, 42], users of PIRSONA need not maintain any local state, nor must they disclose any personal information to third parties or actively participate in the training of models; thus, PIRSONA delivers a user experience akin to now-ubiquitous non-private content distribution services—only with strong, provable privacy guarantees for its users. Toward realizing PIRSONA, we introduce some new primitives of independent interest, including (i) 4PC fixed–selection wire multiplexers (MUXs) and demultiplexers (DeMUXs), (ii) fast 3PC vector normalization for secret-shared fixed-point vectors, (iii) 4PC well–formedness tests for $(2, 2)$-distributed point functions, and (iv) one-round 3PC integer comparison.

2 Preliminaries

We begin our technical discussion by introducing the basic technologies underlying our construction: Hafiz–Henry computationally 1-private PIR [24], Boolean matrix factorization–based collaborative filtering, and Du–Atallah 3-party multiplication [18]. Additionally, we give a bird's-eye overview of how PIRSONA uses the latter MPC to "glue" the first two (seemingly antithetical) primitives into a seamless, privacy-preserving content distribution platform.

2.1 Private information retrieval

PIR is a cryptographic primitive using which users can fetch items from *remote* and *untrusted* database servers without disclosing to the servers which items they fetch. We focus on a specific class of PIR protocols—the so-called "1-private" multiserver scheme of Hafiz and Henry [24]—which provide extremely low download and server-side computation costs. These protocols provide the best-available mix of *efficiency* and *expressiveness* for the media delivery applications we target with PIRSONA. Here "expressiveness" refers to the ability of users to request content using human-memorable identifiers (i.e., titles or keywords) instead of physical addresses (i.e., row numbers) as required by most PIR. The remainder of this subsection briefly explains the Hafiz–Henry scheme.

Consider a database $\boldsymbol{D}$ modelled as an r-by-s matrix over a binary field $\mathbf{GF}(2^w)$. Here each of the r rows comprising $\boldsymbol{D}$ is a distinct *record* (e.g., a video, an e-book, or an app) composed of s-many w-bit *words*, for some suitably chosen bitlength w. A complete replica of $\boldsymbol{D}$ resides at each of $s + 1$ *pairwise non-colluding* database servers (i.e., privacy requires that no two servers collude). We denote the servers by $\mathsf{P}_0, \ldots, \mathsf{P}_s$.[1]

2.1.1 Hafiz–Henry "perfectly 1-private" PIR

A user wishing to fetch $\vec{\boldsymbol{D}}_j$, the jth record in $\boldsymbol{D}$, constructs a *query template* by sampling a length-r vector $\vec{\mathbf{q}} \in_{\mathrm{R}} [0\,..\,s]^r$ uniformly at random subject to $\vec{\mathbf{q}}[j] = 0$. Here and throughout, $[0\,..\,s]$ refers to the set of integers between 0 and s (inclusive), while $\vec{\mathbf{q}}[j] \in [0\,..\,s]$ refers to the jth component of the vector $\vec{\mathbf{q}}$. Next, the user selects a permutation $\sigma\colon [0\,..\,s] \to [0\,..\,s]$ uniformly at

1 One should think of the number of available servers $(s + 1)$ and record size as together dictating both the number of words per record (s) and the bit length of each word (w)—not the converse.

random and then, for each $k \in [0\,..\,s]$, it computes and sends to P_k the vector $\mathsf{query}_k := \vec{\mathbf{q}} + \sigma(k) \cdot \vec{e}_j$, where $\vec{e}_j := \langle 0 \;\; 0 \cdots 1 \cdots 0 \rangle \in \mathbb{Z}^r$ denotes the jth length-r standard basis vector. The permutation σ forces independence between the jth component of query_k and the index k.

Upon receiving its vector query_k from the user, P_k computes and responds with $\mathsf{response}_k := \bigoplus_{i=1}^{r} \vec{\boldsymbol{D}}_i(\mathsf{query}_k[i])$, a scalar in $\mathbf{GF}(2^w)$. As above, $\mathsf{query}_k[i]$ refers to the ith component of query_k, whereas $\vec{\boldsymbol{D}}_i(x) := \vec{\boldsymbol{D}}_i[x]$, the xth word of $\vec{\boldsymbol{D}}_i$, unless $x = 0$, in which case $\vec{\boldsymbol{D}}_i(0) := 0^w$.

Finally, for each $k \in [1\,..\,s]$, the user computes the $\sigma(k)$th word of $\vec{\boldsymbol{D}}_j$ from the pair $(\mathsf{response}_k, \mathsf{response}_0)$ using $\vec{\boldsymbol{D}}_j[\sigma(k)] = \mathsf{response}_k \oplus \mathsf{response}_0$.

Hafiz–Henry perfectly 1-private PIR sharply meets concrete lower bounds for expected download [7, Thm 2.5] and server-side computation [4, Thm 6.4] costs; its client-side computation costs are likewise extremely low. This leaves only the upload cost as a target for optimization. The computationally 1-private variant reduces the per-server upload cost from $\Theta(r \lg s)$ to $\Theta\big((\lg r)(\lg s)\big)$, with no effect on the download cost and with only a nominal increase in the client- and server-side computation costs.

2.1.2 Hafiz–Henry "computationally 1-private" PIR

The computational variant uses *distributed point functions* [20] to compactly represent the vectors $\mathsf{query}_0, \ldots, \mathsf{query}_s$ in an otherwise-unmodified instance of the perfectly 1-private protocol. The scheme is most efficient (and easiest to state) when $s+1$ is a power of 2; we describe here—and assume throughout—this special case and refer to Hafiz and Henry's paper [24, §5.3] for details on the case where $s+1$ is not a power of 2.

Distributed point functions. Let $\mathbb{G}$ be an additive group and $\lambda \in \mathbb{N}$ a security parameter. Intuitively, a *2-out-of-2 distributed point function scheme* (*$((2,2)$-DPF*) is an ordered pair $(\mathsf{Gen}, \mathsf{Eval})$ such that $(\mathsf{seed}_0, \mathsf{seed}_1) \leftarrow \mathsf{Gen}(1^\lambda, r;\, x, y)$ and Eval define a $(2,2)$-additive sharing of the *point function* with domain $[1\,..\,r]$ and point $(x, y) \in [1\,..\,r] \times \mathbb{G}$. In particular, such triplets $(\mathsf{Eval};\, \mathsf{seed}_0, \mathsf{seed}_1)$ provide

1. **(perfect) correctness:** if $i \in [1\,..\,r]$, then $\mathsf{Eval}(\mathsf{seed}_0, i) - \mathsf{Eval}(\mathsf{seed}_1, i) = y$ if $i = x$ and 0 otherwise; and
2. **(computational) hiding:** given only one of seed_0 or seed_1, no PPT algorithm can distinguish a $(2,2)$-DPF with point (x, y) from a $(2,2)$-DPF with some other point $(x', y') \in [0\,..\,r] \times \mathbb{G}$, except with a probability negligible in λ.

Hafiz–Henry uses an efficient $(2,2)$-DPF of Boyle, Gilboa, and Ishai [9, Fig. 1] with 1-bit outputs, which is instantiable using any length-doubling pseudorandom generator (PRG). That construction benefits from the existence of a fast *full-domain evaluation* algorithm [9, §3.2.1], which efficiently expands seed_b into the length-r vector

$$\mathsf{EvalFull}(\mathsf{seed}_b) := \langle \mathsf{Eval}(\mathsf{seed}_b, 1), \ldots, \mathsf{Eval}(\mathsf{seed}_b, r) \rangle .$$

In the sequel, we write $[\![(x, y)]\!]$ as shorthand for a seed pair encoding point (x, y) and $[\![(x, y)]\!]_b$ for the seed held by P_b for $b = 0, 1$. (In contexts where the y-coordinate is immaterial, we sometimes write $[\![(x, \cdot)]\!]$ using '$\cdot$' as a placeholder.)

Construction for $s+1$ a power of 2. Suppose $s + 1 = 2^L$ for some integer $L \geq 2$. A user seeking $\vec{\boldsymbol{D}}_j$, the jth record in $\boldsymbol{D}$, first samples an L-fold sequence of $(2,2)$-DPF seed pairs, $[\![(j,1)]\!]^{(L-1)}, \ldots, [\![(j,1)]\!]^{(0)}$, each with domain $[1\,..\,r]$ and point $(j, 1) \in [0\,..\,r] \times \mathbf{GF}(2)$. Next, for each $k \in [0\,..\,s]$, it sends the L-tuple

$$\mathbf{seed}_k := ([\![(j,1)]\!]^{(L-1)}_{k_{L-1}}, \ldots, [\![(j,1)]\!]^{(1)}_{k_1}, [\![(j,1)]\!]^{(0)}_{k_0}) \qquad (1)$$

to P_k. Here $(k_{L-1} \cdots k_1\; k_0)_2$ is the 0-padded L-bit binary representation of P_k's index k; i.e., $k = \sum_{i=0}^{L-1} k_i 2^i$.

Upon receiving its L-tuple $\mathbf{seed}_k$ from the user, P_k invokes $\mathsf{EvalFull}$ to expand each of the L seeds into a length-r vector of bits, and then it constructs query_k as the componentwise concatenation of the resulting bit vectors; that is, for each $i \in [1\,..\,r]$, it sets

$$\mathsf{query}_k[i] := \mathsf{Eval}([\![(j,1)]\!]^{(L-1)}_{k_{L-1}}, i) \,\|\, \cdots \,\|\, \mathsf{Eval}([\![(j,1)]\!]^{(0)}_{k_{L-1}}, i) .$$

From here, the protocol proceeds identically to its perfectly 1-private counterpart.

2.2 Collaborative filtering

Collaborative filtering is among the most widely deployed recommender engines. Broadly speaking, collaborative filtering renders predictions about the interests and preferences of a user using observations about the interests and preferences of many other users, based

(very roughly) on the assumption that users who exhibit similar preferences about some things will likely exhibit similar preferences about other related things. PIRSONA leverages a well-known *latent-factor collaborative filtering* algorithm based on Boolean matrix factorization [33]. Latent-factor collaborative filtering characterizes observed preferences with respect to some (parameterized) number of latent features; similarities with respect to these latent features form the basis for predictions about users' as-yet-unobserved preferences. Here the qualifier "latent" references the fact that the features are not necessarily tangible attributes like *artist* or *genre*.

Collaborative filtering via matrix factorization. Consider a system in which there are m distinct users fetching items from a database comprising r distinct records and let $\boldsymbol{M} \in \{0,1\}^{m \times r}$ denote the $(0,1)$-matrix having one row per user and one column per record and having $\boldsymbol{M}_{ij} = 1$ if and only if the ith user has fetched the jth record.

Given $\boldsymbol{M}$, the goal of the recommender system is to recommend (i,j) pairs for which $\boldsymbol{M}_{ij} = 0$ and yet the ith user "should" enjoy the jth record. Toward this goal, consider a *user-profile* matrix $\boldsymbol{U} \coloneqq \left[\vec{\boldsymbol{u}}_1; \cdots; \vec{\boldsymbol{u}}_m\right] \in \mathbb{R}^{m \times d}$ and *item-profile* matrix $\boldsymbol{V} \coloneqq \left[\vec{\mathbf{v}}_1; \cdots; \vec{\mathbf{v}}_r\right] \in \mathbb{R}^{r \times d}$, respectively associating a *profile* from $(\mathbb{R} \cap [0,1])^d$ to each of the m users and to each of the r records reflected in $\boldsymbol{M}$. Here $d \in \mathbb{N}$ is a dimensionality parameter indicating the number of latent features needed to effectively capture preferences in the system. Initially, both $\boldsymbol{U}$ and $\boldsymbol{V}$ are selected uniformly subject to $\|\vec{\boldsymbol{u}}_i\|_2 = 1$ and $\|\vec{\mathbf{v}}_j\|_2 = 1$ for all $i \in [1..m]$ and $j \in [1..r]$; the goal is then to "evolve" $\boldsymbol{U}$ and $\boldsymbol{V}$ until $\boldsymbol{U}\boldsymbol{V}^{\mathsf{T}} \in (\mathbb{R} \cap [0,1])^{m \times r}$ is a "reasonable approximation" to $\boldsymbol{M}$, and then to use the differences between $\boldsymbol{U}\boldsymbol{V}^{\mathsf{T}}$ and $\boldsymbol{M}$ as the basis for making recommendations.

To this end, we seek an approximate solution to the minimization problem

$$\begin{aligned}\underset{\boldsymbol{U},\boldsymbol{V}}{\arg\min} \Big\{ \tfrac{1}{\|\boldsymbol{M}\|_1} \sum_{\boldsymbol{M}_{ij}=1} \left(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle\right)^2 \\ + \mu_u \sum_{i=1}^{m} \|\vec{\boldsymbol{u}}_i\|_2^2 + \mu_v \sum_{j=1}^{r} \|\vec{\mathbf{v}}_j\|_2^2 \Big\},\end{aligned}$$

for some convergence parameters $\mu_u, \mu_v \in \mathbb{R}^+$.

There are several known methods for solving such minimization problems. Perhaps the best-known method—and the method employed by PIRSONA—is *gradient descent*, a first-order iterative algorithm for approximating (local) minima of any differentiable function. Each gradient-descent iteration produces progressively "better" approximations (by normalizing the vectors obtained) via the *adaptation rules*

$$\vec{\boldsymbol{u}}_i \leftarrow \hat{\mu}_u \vec{\boldsymbol{u}}_i + \gamma \sum_{\boldsymbol{M}_{ij}=1} \vec{\mathbf{v}}_j \left(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle\right) \tag{2}$$

and

$$\vec{\mathbf{v}}_j \leftarrow \hat{\mu}_v \vec{\mathbf{v}}_j + \gamma \sum_{\boldsymbol{M}_{ij}=1} \vec{\boldsymbol{u}}_i \left(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle\right), \tag{3}$$

for $\hat{\mu}_u = 2\mu_u - 1$, $\hat{\mu}_v = 2\mu_v - 1$, and step size γ.

After several gradient-descent iterations, the resulting $\boldsymbol{U}$ and $\boldsymbol{V}$ are used to render recommendations for the ith user as follows: Compute the length-r vector $\vec{\mathbf{R}}_i \coloneqq \vec{\boldsymbol{u}}_i \boldsymbol{V}^{\mathsf{T}} - \vec{\boldsymbol{M}}_i$, where $\vec{\boldsymbol{M}}_i$ denotes the ith row of $\boldsymbol{M}$; the largest components of $\vec{\mathbf{R}}_i$ are the "top" recommendations for the ith user.

2.3 Du–Atallah 3-party multiplication

In principle, any secure MPC could serve as the substrate through which PIRSONA performs collaborative filtering over PIR queries. However, as our design already uses 1-private PIR among 2^L pairwise non-colluding servers, we focus our attention on 4PC protocols secure when all parties are semi-honest. This special subclass of MPC admits computations that are highly efficient relative to, e.g., 2PC from garbled circuits or general MPC secure against threshold collusion or Byzantine computation parties. In the following and throughout, we use $[x]$ to denote a $(2,2)$-additive sharing of x and $[x]_b$ to denote the portion of $[x]$ held by P_b for $b = 0, 1$.

We employ Du–Atallah multiplication [18], an efficient 1-private 3-party variant of Beaver triples [3] that introduces a semi-trusted third party to eliminate expensive oblivious transfers (OTs)—the bottleneck in traditional 2PC from Beaver triples—without increasing other costs. Let R be a finite ring and suppose that P_0 and P_1 hold a $(2,2)$-additive sharing $([x],[y])$ of the pair $(x,y) \in R \times R$; the goal is for them to compute a sharing $[z]$ of the product $z = xy$. A semi-trusted third party, P_2, facilitates this computation. At a high level, the idea is for P_0 and P_1 to exchange blinded copies of their respective shares so that both parties effectively hold (x,y), albeit with additional machinery to maintain secrecy. P_2 chooses *blinding factors* and *cancellation terms* to help P_0 and P_1 arrive at the final product; specifically, P_2 samples $X_0, X_1, Y_0, Y_1, \alpha \in_{\mathrm{R}} R$, and then it sends $(X_0, Y_0, X_0 Y_1 + \alpha)$ to P_0 and $(X_1, Y_1, X_1 Y_0 - \alpha)$ to P_1.

P_1 then sends $([x]_1 + X_1, [y]_1 + Y_1)$ to P_0, who outputs

$$[z]_0 \coloneqq [x]_0([y]_0 + ([y]_1 + Y_1)) - Y_0([x]_1 + X_1) + (X_0 Y_1 + \alpha),$$

while P_0 sends $([x]_0 + X_0, [y]_0 + Y_0)$ to P_1, who outputs

$$[z]_1 \coloneqq [x]_1([y]_1 + ([y]_0 + Y_0)) - Y_1([x]_0 + X_0) + (X_1 Y_0 - \alpha).$$

It is easy to verify that Du–Atallah multiplication is correct—i.e., $[z]_0 + [z]_1 = ([x]_0 + [x]_1)([y]_0 + [y]_1) = xy$—and that the view of each party to the computation is simulatable. Also notice that all contributions of P_2 can be computed offline before P_0 and P_1 ever receive their private shares. Finally, we remark that Du–Atallah generalizes naturally to inner products [17] and efficient scalar–vector multiplication.

2.4 Secure 4PC as "garbled glue"

Whenever the ith user submits a PIR query over the database, some designated subset of the servers stores a tuple $(i, [\![(j,1)]\!]_b)$, where $[\![(j,1)]\!]_b$ is one of the L seeds constituting that query. Periodically, the designated servers perform a 4-party computation that takes as input (i) these $(i, [\![(j,1)]\!]_b)$ pairs and (ii) the previously computed user- and item-profile shares, and then outputs updated user- and item-profile shares. The updated profile shares can be multiplied to produce a personalized content recommendation vector upon which to base user-specific recommendations.

3 Threat model

As PIRSONA composes *computationally 1-private* PIR with MPC secure against a *single passive corruption*, three assumptions are immediate and unavoidable; namely, that

1. the adversary is computationally bounded,
2. the adversary is honest-but-curious, and
3. the adversary controls *at most one* (of 2^L) servers.

PIRSONA's reliance on computational assumptions follows from its use of $(2,2)$-DPFs that are instantiable from any length-doubling PRG; thus, beyond the (admittedly quite strong) assumption of non-collusion, PIRSONA relies only on rather conservative "minicrypt" assumptions [28]. For instance, our implementation assumes only that AES-128 is hard to distinguish from an ideal random permutation.

PIRSONA employs a novel 4PC sanity-check protocol for the DPFs to ensure that users cannot corrupt the user- and item-profile shares by submitting malformed queries; thus, we can safely assume that the adversary controls—in addition to one server—an arbitrary number of Byzantine users. This is an important aspect of the model, as contributing strategically chosen queries may enable the adversary to influence the collaborative filtering profiles in a way that causes the model to leak information about the consumption histories of honest users [10]. We stress that PIRSONA does not attempt to limit such collaborative filtering model leakage (though doing so would be an interesting direction for future work); rather, we merely insist that the only source of information leakage is that which is inherent to the collaborative filtering recommendations.

Finally, we assume that all user–server and server–server communication occurs over point-to-point confidential and mutually authenticated channels. We do not concern ourselves with how these channels are realized. (Our implementation uses TLS; a real deployment would presumably couple this with password- or key-based authentication.)

Appendix D presents a formal definition and analysis in the ideal-/real-world simulation paradigm. In the ideal world, all PIRSONA users hand their requests directly to a benevolent trusted party who provides answers and updates profiles in response to incoming requests. In the real world, we replace the trusted party with the MPC protocols described in the sequel. The adversary $\mathcal{A}$ seeks to draw inferences about the consumption patterns of non-corrupted users. Informally, we prove that operating in the real world provides $\mathcal{A}$ with no tangible advantage over operating in the ideal world, where all inferences about non-corrupted users necessarily result from model leakage inherent to the underlying collaborative filtering recommendations—not from the MPC components PIRSONA uses to realize those recommendations.

4 Overview of the 4PC for gradient descent

We now describe, at a high level, the 4PC protocol PIRSONA uses to evaluate Equations (2) and (3), the so-called adaptation rules for gradient descent–based Boolean matrix factorization. Given an incoming query from user i for item j, a designated subset of servers must cooperatively

1. locate the user profile $\vec{u}_i$ and item profile $\vec{v}_j$;

2. compute $\Delta_{\mathbf{u}_i} \coloneqq \vec{\mathbf{v}}_j(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle)$ and $\Delta_{\mathbf{v}_j} \coloneqq \vec{\boldsymbol{u}}_i(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle)$;
3. update $\vec{\boldsymbol{u}}_i$ with $\Delta_{\mathbf{v}_j}$ and $\vec{\mathbf{v}}_j$ with $\Delta_{\mathbf{u}_i}$; and, ultimately,
4. normalize the resulting updated profiles.

In the following, we elaborate on each step in turn.

Step 1a: Locating the user profile. This step is made trivial by virtue of all user–server communication occurring over mutually authenticated channels; that is, because each server knows with which user it is communicating, it can easily retrieve the correct entry from its list of user-profile shares.

Step 1b: Locating the item profile. Recall that users fetch items via Hafiz–Henry computationally 1-private PIR. In that protocol, the user submits an L-fold sequence $(\llbracket (j,1) \rrbracket^{(L-1)}, \ldots, \llbracket (j,1) \rrbracket^{(1)}, \llbracket (j,1) \rrbracket^{(0)})$ of DPF seeds to each server (see §2.1.2). Assume, without loss of generality, that servers P_0 and P_2 receive $\llbracket (j,1) \rrbracket_0^{(0)}$ while P_1 and P_3 receive $\llbracket (j,1) \rrbracket_1^{(0)}$. The four parties use their respective seeds to fetch shares of $\vec{\mathbf{v}}_j$ using the 4-party MUX construction detailed in Section 6.2.2. For technical reasons (explained in Section 6.2), the 4-party MUX ultimately results in P_0 and P_1 holding $(2,2)$-additive shares of $((-1)^b \vec{\mathbf{v}}_j, (-1)^b)$, where b is a uniformly distributed sign bit.

Step 2: Computing $\Delta_{\mathbf{u}_i}$ and $\Delta_{\mathbf{v}_j}$. The computation of $\Delta_{\mathbf{u}_i}$ and $\Delta_{\mathbf{v}_j}$ proceeds via a 4-move 3PC protocol comprising three sequential rounds of generalized Du–Atallah multiplications (see §2.3). Note that after Step 1, P_0 and P_1 hold a $(2,2)$-additive sharing $([\vec{\boldsymbol{u}}_i], [(-1)^b \vec{\mathbf{v}}_j], [(-1)^b])$. P_2 makes the first move, sending a sequence of Du–Atallah blinding factors and cancellation terms for P_0 and P_1 to use in the subsequent moves. In the second move, P_0 and P_1 use $([\vec{\boldsymbol{u}}_i], [(-1)^b \vec{\mathbf{v}}_j])$ to compute $[(-1)^b(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle)]$. In the third move, P_0 and P_1 use $([(-1)^b], [(-1)^b(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle)])$ to compute $[1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle]$. In the fourth and final move, P_0 and P_1 use $([(-1)^b \vec{\mathbf{v}}_j], [(-1)^b(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle)])$ to compute $[\Delta_{\mathbf{u}_i}]$, and they use $([\vec{\boldsymbol{u}}_i], [1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle])$ to compute $[\Delta_{\mathbf{v}_j}]$.

Step 3a: Updating $\vec{\boldsymbol{u}}_i$ with $\Delta_{\mathbf{v}_j}$. Analogous to Step 1a, this step is made trivial by virtue of all user–server communication occurring over mutually authenticated channels.

Step 3b: Updating $\vec{\mathbf{v}}_j$ with $\Delta_{\mathbf{u}_i}$. This step is roughly converse to Step 2b; thus, whereas Step 2b used seeds from the user's query to instantiate a 4-party MUX, this step uses them to instantiate a 4-party DeMUX, as detailed in §6.3.

Step 4: Normalizing the updated profiles. At this point, P_0 and P_1 hold $(2,2)$-additive shares of the updated (but non-normalized) user profile $\vec{\boldsymbol{u}}_i$ and item-profile matrix $\boldsymbol{V}$; their goal is to normalize each of the shared profiles. We describe here only the normalization of $\vec{\boldsymbol{u}}_i$; normalization of the profiles comprising $\boldsymbol{V}$ is similar. The idea is for P_0 and P_1 (with help from P_2) to use Du–Atallah to compute a sharing of $\|\vec{\boldsymbol{u}}_i\|_2^2 = \langle \vec{\boldsymbol{u}}_i, \vec{\boldsymbol{u}}_i \rangle$, and then to enlist the help of P_2 to convert these into a sharing of $1/\|\vec{\boldsymbol{u}}_i\|_2$. At a high level, the three parties do this by running a modified PIR protocol to jointly fetch shares of the coefficients for a high-fidelity linear approximation to the inverse square-root function.

Section 6 presents complete details and analyses of the 4-party MUX used in Step 1b (§6.2), the 4-party DeMUX used in Step 3b (§6.3), and the 3-party vector-normalization protocol used in Step 4 (§6.4). Following that, Section 7 describes one potential way to *use* the resulting item and user profiles—an efficient 3PC that takes as input (i) a uniformly sampled number $u \in ([0,1] \cap \mathbb{R})$ and (ii) shares of the resulting user profile $\vec{\boldsymbol{u}}_i$ and item-profile matrix $\boldsymbol{V}$, and outputs a personalized recommendation for user i.

5 System architecture

Before presenting the details of our MPC, we elaborate on the high-level PIRSONA architecture sketched out in the preceding sections. Recall that the system comprises $s + 1 = 2^L$ pairwise non-colluding servers, $\mathsf{P}_0, \ldots, \mathsf{P}_s$, for some $L \geq 2$. Of these, only four are involved in collaborative filtering profile updation. Specifically, P_0 and P_1 play a central role in all aspects of collaborative filtering, while P_2 participates in all 3PC sub-protocols and P_3 joins only for the occasional 4PC sub-protocol. Any additional servers merely respond to users' PIR queries.[2] In particular, P_0 and P_1 hold $(2,2)$-additive sharings $([\boldsymbol{U}], [\boldsymbol{V}])$ of the user- and item-profile matrices; P_2 and P_3 hold $[\boldsymbol{V}]_1$ and $[\boldsymbol{V}]_0$ in common with P_0 and P_1, respectively. Upon receiving a query from user i, P_0 and P_2 store the pair $(i, \llbracket (j,1) \rrbracket_0^{(0)})$ consisting of the user's identity and the "least-significant–bit" DPF seed

2 The best-possible download cost and (per-server) computation costs of PIR are each decreasing functions of the number of servers. Being sharply optimal in both metrics, Hafiz–Henry grows increasingly efficient as the number of servers is allowed to grow large. This potential for higher efficiency is why we entertain the possibility of setting $L > 2$.

from her query (see Equation (1)); likewise, P_1 and P_3 store the pair $(i, [\![(j,1)]\!]_1^{(0)})$. Periodically, the four parties input these shares to a 4PC to update the collaborative filtering profiles (see §4).

5.1 Query representation

By default, Hafiz–Henry assumes users fetch records via *positional queries* [23]; that is, via their row numbers within $\boldsymbol{D}$. We remark that (i) the DPFs naturally support keyword-based queries [12] as a more user-friendly alternative and (ii) this extension "plays well" with Hafiz–Henry queries. Beyond the preceding remark, we do not concern ourselves with how users specify which records they seek (though we *do* consider a candidate user interface in Section 9); however, we do prescribe one minor change to Hafiz–Henry query generation in support of PIRSONA functionality: The DeMUX assumes a so-called *leafless seed* for the "least-significant-bit" DPF (see §6.3) and, consequently, that the servers substitute the bit vector $\mathsf{flags}([\![(j,\cdot)]\!]_b^{(0)})$, as defined in §6.1, in lieu of $\mathsf{EvalFull}([\![(j,1)]\!]_b^{(0)})$ when constructing query_k.

5.2 Profile representation

Recall that collaborative filtering user and item profiles are length-d unit vectors of positive reals (see §2.2), whereas Du–Atallah multiplication works over an arbitrary finite ring R (see §2.3). For concreteness—and to facilitate efficient implementation on 64-bit computers—PIRSONA fixes R as the ring of integers modulo 2^{64}; profiles are then vectors of 64-bit unsigned integers representing fixed-point approximations to reals. Specifically, the fixed-point approximation to x is $\lfloor x\, 2^p \rfloor$, where $p \in \mathbb{N}$ is a *fractional precision* parameter indicating how many bits are used to represent the fractional part of x. Since profile components are non-negative and bounded above by 1, we require high-fidelity fractional approximations. Yet setting p too large can cause intermediate values to overflow, as the product of fixed-point numbers with fractional precisions p_1 and p_2 has fractional precision $p_1 + p_2$ (and summing n such numbers grows the integer part by up to $\lg n$ bits).

When $\vec{\boldsymbol{u}}_i$ and $\vec{\mathbf{v}}_j$ are represented by vectors of fixed-point numbers with fractional precision p, the gradient-descent adaptation rules yield vectors $\Delta_{\mathbf{u}_i} = \vec{\mathbf{v}}_j(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle)$ and $\Delta_{\mathbf{v}_j} = \vec{\boldsymbol{u}}_i(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle)$ with fractional precision $3p$. For technical reasons (see §6.4.1), we must ensure that the most-significant bit is unset in every vector component; hence, we set $p = 64 - 3p - N - 1$ with $N \coloneqq \lceil (\lg(n+1))/(1+\epsilon) \rceil$, where n is a tuning parameter determining the number of adaptation values the servers can accumulate without risk of overflows and $\epsilon \geq 0$ captures the worst-case approximation error in our 3-party vector normalization protocol (see §6.4).

We emphasize that our MPC protocols are agnostic to the fixed-point representation up until the *normalization* step (see §6.4), which includes *precision-reduction* substeps (§6.4.1) to "reset" the fractional precision as needed.

6 Primitives for gradient descent

We begin with an overview of the Boyle–Gilboa–Ishai $(2,2)$-DPF construction [9, Fig. 1], which serves as the scaffolding around which we construct our 4PC gradient descent.

6.1 Boyle–Gilboa–Ishai (2,2)-DPFs

The presentation herein attempts to strike a balance between focusing on the intuition behind the Boyle–Gilboa–Ishai construction and clearly articulating specific low-level details upon which our MUXs and DeMUXs rely.

The Goldreich–Goldwasser–Micali PRF. At its core, Boyle et al.'s construction is a clever generalization of the celebrated Goldreich–Goldwasser–Micali (GGM) construction [21] of a *pseudorandom function (PRF)* from any length-doubling PRG. The GGM construction represents a PRF taking h-bit inputs by a height-h binary tree in which (i) the root node is labelled by the λ-bit PRF key K (i.e., a PRG seed); and (ii) the left and right children of each non-leaf node are respectively labelled by the left and right halves of the image of the PRG applied to the label on the parent. Notice that there is a natural bijection between the set of h-bit binary strings and the leaves of a tree so defined, with the sequence of left (meaning the next bit is 0) or right (meaning the next bit is 1) traversals from the root to any leaf associating that leaf with a specific h-bit input string. GGM defines the image $F_K(i)$ of the PRF keyed by $K \in \{0,1\}^\lambda$ and evaluated at $i \in \{0,1\}^h$ as the *label* on the leaf node associated with i when this natural bijection is applied to the height-h tree with root labelled K.

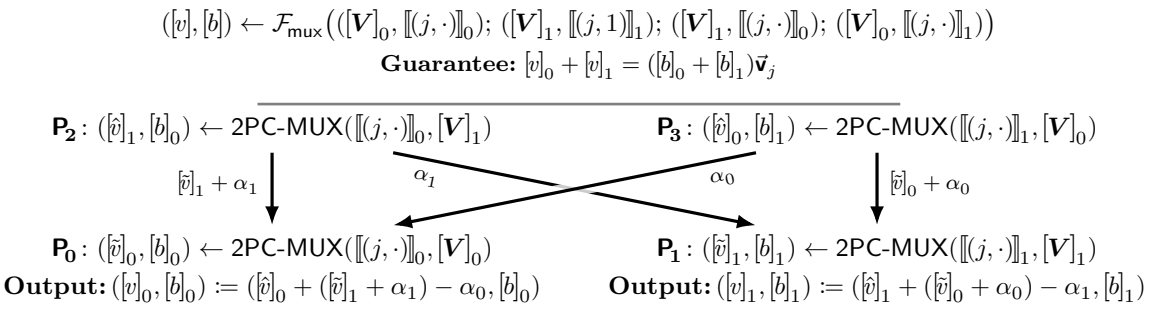


Fig. 1. 4-party fixed–selection wire MUX with secret-shared input wires, as described in Section 6.2.2.

(2,2)-DPFs from pairs of GGM-like trees. Boyle et al.'s construction maintains the basic structure of GGM, while adding some additional machinery in the form of so-called *correction words (CWs)*; that is, traversing the "DPF tree share" for a given input follows the same basic process as in GGM: Begin at the root and repeatedly traverse either left or right, depending on successive bits of the input, until arriving at a leaf node. The main difference is in how DPF-tree traversal incorporates the CWs into the computation of labels. At a high level, the strategy is to single out the path from the root to a specific *target leaf node* (the x-coordinate of the point function's "point"), and then to specially craft the CWs to force the labels on all *other* paths in a given pair of GGM-like trees to converge, thereby assigning identical labels to all non-target leaves. To this end, the construction associates with each layer of the tree shares one λ-bit CW, the two least-significant bits of which are special *toggle bits* that help the evaluator decide to which nodes it should "apply" the CW. The precise details of how to generate and apply CWs (and toggle bits) goes beyond what is needed to understand our MUX and DeMUX constructions; for brevity, we omit those details here and direct interested readers to Gilboa and Ishai [20] and Boyle et al. [9]. For our purposes, it suffices to know that

1. each node (in each tree share) is associated with its own *flag bit* that is determined by a combination of the toggle bits and node labels produced by the PRG output together with the CW for the preceding layer;
2. the distribution of flag bits within each tree share is *pseudorandom* (and *independent of the target leaf* node); and
3. the flag bits in corresponding nodes of the two DPF tree shares are *pairwise identical*—except along the path to the target leaf, where they are *pairwise complementary*.

In the following, we denote by $\mathsf{flags}([\![(x,y)]\!]_0)$ and $\mathsf{flags}([\![(x,y)]\!]_1)$ the vectors of leaf-layer flag bits for the trees induced by $[\![(x,y)]\!]$.

Taken together, the preceding facts yield the next observation, upon which we base our MUX constructions.

Observation 1. *If* $[\![(x,y)]\!] \leftarrow \mathsf{Gen}(1^\lambda, r; x, y)$*, then the vectors* $\mathsf{flags}([\![(x,y)]\!]_0)$ *and* $\mathsf{flags}([\![(x,y)]\!]_1)$ *define a computationally private* $(2,2)$*-additive sharing of the* x*th standard basis vector of length* r *over* $\mathbf{GF}(2)$*; i.e.,* $\mathsf{flags}([\![(x,y)]\!]_0) \oplus \mathsf{flags}([\![(x,y)]\!]_1) = \vec{\boldsymbol{e}}_x$.

6.2 DPFs as fixed–selection wire MUXs

An r-to-1 *multiplexer (MUX)* is a combinational logic circuit that, based on the signal it receives on a *selection wire*, forwards the value from one *input wire* (out of r) to a single *output wire*. That 2-party MUXs are instantiable from $(2,2)$-DPFs is hardly surprising: Privately selecting one-out-of-r input wires is substantially equivalent to 2-server PIR over r items—the veritable "bread and butter" of $(2,2)$-DPFs.

The above-noted parallel between MUXs and PIR notwithstanding, two important caveats differentiate DPFs in our MPC setting from their use in the more familiar PIR setting; specifically, existing DPF-based PIR protocols

1. operate on the leaf-node *labels*—indeed, the ubiquitous early-termination optimization [9, §3.2.3] packs an output into each bit of each leaf-node label, effectively lopping $\lfloor \lg \lambda \rfloor$ layers off the DPF trees; and they
2. operate in a binary field, using the DPF outputs only to decide which records to include in a big exclusive-OR.

Notably, working in the characteristic-2 setting of binary fields eliminates the notion of positive and negative *signs*. Even in settings where the characteristic is greater than 2, the querier can typically (i) flip the sign at reconstruction time or (ii) induce the desired sign via the leaf-layer CW.

In our setting, the MUX outputs $(2,2)$-additive shares in $\mathbb{Z}_{2^{64}}$, which are input to some larger 4PC protocol; thus, *signs matter*. Yet, jumping ahead, we reserve the leaf-layer labels for use in demultiplexing (see §6.3) and use only the leaf-layer flag bits for multiplexing. (The alternative would be two DPFs—one for multiplexing and one for demultiplexing—plus a consistency check to ensure they share a target input; we opt to use one DPF for efficiency: It halves storage cost and calls to EvalFull while obviating the need for an additional consistency check.) Using the flag bits precludes inducing the desired sign via the leaf-layer CW; instead, our MUX outputs (shares of) the sign bit so that subsequent MPC can flip the sign as needed.

6.2.1 2-party MUX with public input wires

The 2-party MUX is fast and simple, but it requires cleartext values on all input wires;[3] the 4-party MUX adds support for secret-shared input wires.

Suppose P_0 and P_1 hold $[\![(j,\cdot)]\!]$ and $\boldsymbol{V} = \left[\vec{\mathbf{v}}_1; \cdots; \vec{\mathbf{v}}_r\right]$; the goal is for P_0 and P_1 to compute $([(-1)^b\vec{\mathbf{v}}_j], [(-1)^b])$. To this end, P_0 computes and outputs

$$([v]_0, [b]_0) \coloneqq \left(\sum_{\mathsf{flags}([\![(j,\cdot)]\!]_0)[i]=1} \vec{\mathbf{v}}_i,\ \|\mathsf{flags}([\![(j,\cdot)]\!]_0)\|_1\right)$$

while P_1 computes and outputs

$$([v]_1, [b]_1) \coloneqq \left(-\sum_{\mathsf{flags}([\![(j,\cdot)]\!]_1)[i]=1} \vec{\mathbf{v}}_i, -\|\mathsf{flags}([\![(j,\cdot)]\!]_1)\|_1\right).$$

Correctness of the above 2-party MUX construction follows immediately from Observation 1. We write $([(-1)^b\vec{\mathbf{v}}_j], [(-1)^b]) \leftarrow \mathsf{2PC\text{-}MUX}([\![(j,\cdot)]\!], \boldsymbol{V})$.

6.2.2 4-party MUX with secret-shared input wires

The 4-party MUX is a 2-move protocol that supports $(2,2)$-additively shared input wires. It follows a simple *mix-and-match* pattern that we will see again in the context of DPF sanity checks for our 4-party DeMUX (§6.3).

Let $[\boldsymbol{V}]$ be a $(2,2)$-additive sharing of $\boldsymbol{V} = \left[\vec{\mathbf{v}}_1; \cdots; \vec{\mathbf{v}}_r\right]$. Suppose P_0 and P_2 both hold $[\![(j,\cdot)]\!]_0$, while P_1 and P_3 both hold $[\![(j,\cdot)]\!]_1$; likewise, suppose that P_0 and P_3 both hold $[\boldsymbol{V}]_0$, while P_1 and P_2 both hold $[\boldsymbol{V}]_1$. As in the 2-party MUX, the goal is to compute $([(-1)^b\vec{\mathbf{v}}_j], [(-1)^b])$. For the first move:

1a) P_2 selects an output-wire blinding factor α_1, computes $([\hat{v}]_1, [b]_0) \leftarrow \mathsf{2PC\text{-}MUX}([\![(j,\cdot)]\!]_0, [\boldsymbol{V}]_1)$, and then it sends α_1 to P_1 and $[\hat{v}]_1 + \alpha_1$ to P_0; meanwhile,

1b) P_3 selects an output-wire blinding factor α_0, computes $([\hat{v}]_0, [b]_1) \leftarrow \mathsf{2PC\text{-}MUX}([\![(j,\cdot)]\!]_1, [\boldsymbol{V}]_0)$, and then sends α_0 to P_0 and $[\hat{v}]_0 + \alpha_0$ to P_1.

For the second move:

2a) P_0 computes $([\tilde{v}]_0, [b]_0) \leftarrow \mathsf{2PC\text{-}MUX}([\![(j,\cdot)]\!]_0, [\boldsymbol{V}]_0)$ and $[v]_0 \coloneqq [\tilde{v}]_0 + ([\hat{v}]_1 + \alpha_1) - \alpha_0$, and then it outputs $([v]_0, [b]_0)$; meanwhile,

2b) P_1 computes $([\tilde{v}]_1, [b]_1) \leftarrow \mathsf{2PC\text{-}MUX}([\![(j,\cdot)]\!]_1, [\boldsymbol{V}]_1)$ and $[v]_1 \coloneqq [\tilde{v}]_1 + ([\hat{v}]_0 + \alpha_0) - \alpha_1$, and then it outputs $([v]_1, [b]_1)$.

An easy derivation confirms that $[v]_0 + [v]_1 = ([b]_0 + [b]_1)\vec{\mathbf{v}}_j$, as desired. Figure 1 illustrates the above 4-party MUX construction and its "mix-and-match" structure.

6.3 DPFs as fixed–selection wire DeMUXs

A 1-to-r *demultiplexer (DeMUX)* is a logic circuit that implements the converse of an r-to-1 MUX; that is, based on the signal it receives on a *selection wire*, it forwards the value from a single *input wire* to one *output wire* (out of r). Oblivious demultiplexing bears a striking resemblance to so-called PIR-Writing [40], a task for which DPFs have been fruitfully deployed in the contexts of, e.g., oblivious RAM [16, 35] and anonymous messaging [14].

Yet one crucial aspect of our DeMUXs differentiates them from standard PIR-Writing: in PIR-Writing, the user fixes values on *both* the selection wire *and* the input wire, whereas our DeMUX effectively *unfixes* the input wire, enabling the *servers*—as opposed to the user—to forward any (secret-shared) value of their choosing to an output wire determined by the (still fixed by the user) selection wire. All other output wires receive $(2,2)$-additive sharings of zero. We remark that Kushilevitz and Mour propose a superficially similar DPF-based 4-party PIR-Writing construction in the context of oblivious RAM [35, Figure 1]; however, unlike with our DeMUXs, the *user* in Kushilevitz–Mour must choose the

3 For PIRSONA, this would require P_0 and P_1 to hold cleartext item profiles, thus leaking information beyond what is inherently leaked by collaborative filtering.

value for the input wire (and it must do so at DPF generation time). This distinction makes our DeMUXs significantly more powerful and is essential for PIRSONA functionality, where the DPF generator (i.e., a PIRSONA user) is not privy to the value (i.e., a scalar multiple of $\Delta_{\mathbf{v}_j}$) to appear on the input wire.

Leafless DPF seeds. The "input-wire" of a Boyle–Gilboa–Ishai DPF is determined by the leaf-layer CW; thus, we "unfix" the input-wire value by deferring selection of this CW. To this end, we provide a modified Gen that emits $(2,2)$-additive shares of a leaf-layer CW for the target point $(j,0) \in [1..r] \times R$. With the latter shares, P_0 and P_1 can compute a leaf-layer CW corresponding to any $(2,2)$-additively shared input wire value from R.

While DPF seeds without a leaf-layer CW are effectively *inevaluable*, the missing CW presents no obstacle to generating the interior of the DPF tree, up to and including the leaf-layer flag bits—it is only the (labels on) the leaf nodes that are not yet determined. Thus, such "defoliated" DPFs remain suitable for use in our 2- or 4-party MUXs (see §6.2). In a nod to arboriculture, we call the eventual computation of the leaf-layer CW for such leafless seeds *vernalization*. The modified seed-generation algorithm, $\mathsf{G\overline{en}}$, uses the default Gen as a black box; specifically, on input $(1^\lambda, r;\, j)$, it

1. samples $[\![(j,0)]\!] \leftarrow \mathsf{Gen}(1^\lambda, r;\, j, 0)$;
2. parses each $[\![(j,0)]\!]_b$ as $(K_b;\, cw_0 \| \cdots \| cw_{h-1} \| \overline{cw}_h)$, with $\overline{cw}_h$ being the leaf-layer CW; and then
3. outputs $[\![(j,\star)]\!]_0 := (K_0;\, cw_0 \| \cdots \| cw_{h-1})$ and $[\![(j,\star)]\!]_1 := (K_1;\, cw_0 \| \cdots \| cw_{h-1})$ and $[\overline{cw}_h]$.

Vernalization is a 2-party interaction comprising a single move. Let $y \in R$ be a secret and suppose P_0 and P_1 hold sharings $([\![(j,\star)]\!], [\overline{cw}_h], [y])$. P_0 and P_1 compute $[cw_h] := [\overline{cw}_h] + [y]$ and then reconstruct to the leaf-layer CW

$$\begin{aligned} cw_h &:= [cw_h]_0 + [cw_h]_1 \\ &= ([\overline{cw}_h]_0 + [y]_0) + ([\overline{cw}_h]_1 + [y]_1) \\ &= ([\overline{cw}_h]_0 + [\overline{cw}_h]_1) + ([y]_0 + [y]_1) \\ &= \overline{cw}_h + y\,. \end{aligned}$$

Finally, P_0 outputs $[\![(j,y)]\!]_0 := (K_0;\, cw_0 \| \cdots \| cw_{h-1} \| cw_h)$ and P_1 outputs $[\![(j,y)]\!]_1 := (K_1;\, cw_0 \| \cdots \| cw_{h-1} \| cw_h)$.

6.3.1 2-party DeMUX construction

Suppose P_0 and P_1 hold sharings $([\![(j,\star)]\!], [\overline{cw}_h], [y])$; the goal is for them to compute $[y \cdot \vec{e}_j]$. To this end, (i) P_0 and P_1 use 2-party vernalization to convert $([\![(j,\star)]\!], [\overline{cw}_h], [y])$ into $[\![(j,y)]\!]$, and then (ii) they respectively output $[y \cdot \vec{e}_j]_0 \leftarrow \mathsf{EvalFull}([\![(j,y)]\!]_0)$ and $[y \cdot \vec{e}_j]_1 \leftarrow \mathsf{EvalFull}([\![(j,y)]\!]_1)$.

6.4 3PC fixed-point vector normalization

Recall that user profiles $\vec{u}_i$ and item profiles $\vec{v}_j$ are length-d unit vectors of fixed-point numbers with fractional precision p; thus, the adaptation values $\Delta_{\mathbf{u}_i} = \vec{v}_j(1 - \langle \vec{u}_i, \vec{v}_j \rangle)$ and $\Delta_{\mathbf{v}_j} = \vec{u}_j(1 - \langle \vec{u}_i, \vec{v}_j \rangle)$ are themselves at-most-unit magnitude vectors[4] having fractional precision $3p$. Profile updation consists of summing one or more such adaptation values into a given profile. As detailed in Section 5.2, we choose p to maximize fixed-point approximation accuracy subject to an "overflow quota" allowing the accumulation of some tuneable number of adaptation values without risk of overflowing into the most-significant bits of profile components.

For user-profile updation, the servers know precisely how many adaptation values are accumulated; for item-profile updation, the DeMUX hides this number and we must count every adaptation value against the overflow quota of *all* item profiles. In both cases, upon hitting the overflow quota, the servers must perform a precision reduction, as described in Section 6.4.1, to allocate additional bits for the integer parts of the profile components.

6.4.1 3-party precision reduction

Suppose P_0 and P_1 hold a $(2,2)$-additive sharing $[x]$ of a fixed-point number x with precision p. We now describe a protocol using which P_0 and P_1 (with some help from P_2) can produce shares of the largest fixed-point number less than or equal to x having precision $p - s$, where $s \in [1..p]$ is a parameter. If x were given in cleartext, such a precision reduction would be trivially realized by an arithmetic shift to the right by s bits; in the $(2,2)$-additive sharing setting, by contrast, we must account for any implicit reductions that arise when reconstructing $x = [x]_0 + [x]_1$ using 64-bit integer arithmetic.

Our approach insists that the most-significant bit of x is always 0, and then it examines the most-significant bits of $[x]_0$ and $[x]_1$ to identify three cases:

4 That both $\|\Delta_{\mathbf{u}_i}\|_2 \le 1$ and $\|\Delta_{\mathbf{v}_j}\|_2 \le 1$ follows by the Cauchy–Schwarz inequality.

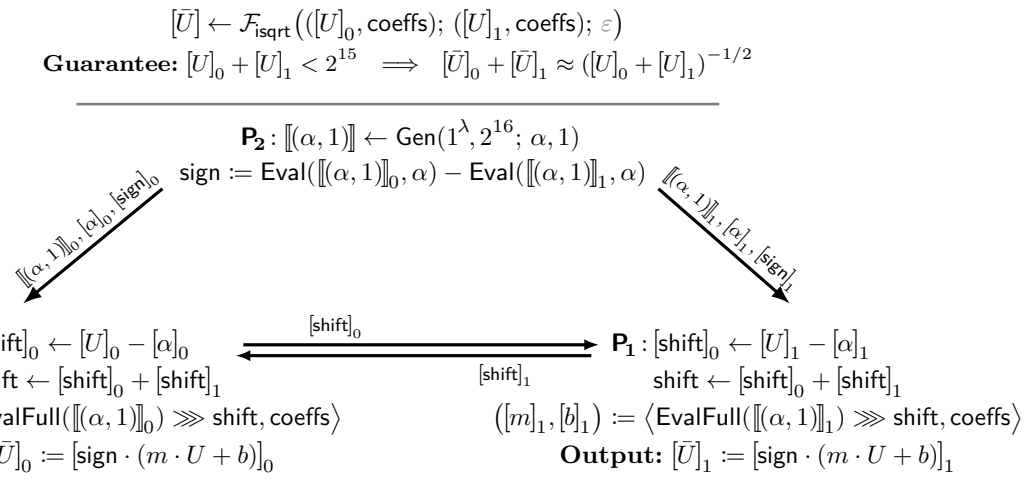


Fig. 2. 3-party inverse square-root protocol, as described in Section 6.4.2. In the diagram, '$\ggg$' is the cyclic right shift operator. P_0 and P_1 compute $[\bar{U}]$ from $([\mathsf{sign}], [m], [b])$ using Du-Atallah multiplication; the diagram omits the Du-Atallah steps for clarity.

1. $\mathrm{msb}([x]_0) = \mathrm{msb}([x]_1) = 0$ so that $x = [x]_0 + [x]_1$ over $\mathbb{Z}$, in which case no implicit reduction occurs and no corrective action is needed;
2. $\mathrm{msb}([x]_0) \neq \mathrm{msb}([x]_1)$ so that $x = [x]_0 + [x]_1 + 2^{64}$ over $\mathbb{Z}$, in which case the arithmetic shifts of $[x]_0$ and $[x]_1$ preserve a carry chain that results in the desired implicit reduction and, again, no corrective action is needed; and
3. $\mathrm{msb}([x]_0) = \mathrm{msb}([x]_1) = 1$ so that $x = [x]_0 + [x]_1 - 2^{64}$ over $\mathbb{Z}$, in which case the arithmetic shifts reconstruct to a secret having its s most-significant bits (incorrectly) set.

Given these three cases, it suffices to add $[2^{64-s}]$ to the arithmetic shifts of $[x]$ if and only if $\mathrm{msb}([x]_0) = \mathrm{msb}([x]_1) = 1$, a procedure we can implement obliviously by computing the product of $\mathrm{msb}([x]_0)$ and $\mathrm{msb}([x]_1)$ using a 1-bit Du–Atallah multiplication and the help of P_2, and then scaling the resulting shares by 2^{64-s}.

6.4.2 Normalizing the precision-reduced vectors

Suppose P_0 and P_1 hold a $(2,2)$-additive sharing $[\vec{\boldsymbol{u}}_i']$ of an updated, but not-yet-normalized profile with fractional precision p. Their goal is to arrive at a sharing $[\vec{\boldsymbol{u}}_i'']$, where $\vec{\boldsymbol{u}}_i'' \approx \vec{\boldsymbol{u}}_i' / \|\vec{\boldsymbol{u}}_i'\|_2$. To do this, they enlist the help of P_2 in a 3-party protocol. All three parties hold in common an associative array coeffs mapping ranges of possible values of $\|\vec{\boldsymbol{u}}_i'\|_2^2$ to coefficients for a good linear approximation to *inverse (reciprocal) square roots* within that range. (Our implementation employs a simple greedy algorithm to find ranges and coefficients exhibiting low approximation errors.)

At the outset, P_0 and P_1 compute $[\|\vec{\boldsymbol{u}}_i'\|_2^2]$ from $[\vec{\boldsymbol{u}}_i']$ using generalized Du–Atallah (and help from P_2). From here, the three parties employ a modified PIR protocol to fetch suitable coefficients from coeffs, as illustrated in Figure 2: P_2 samples α and shares $[\alpha]$ and $[\![(\alpha, 1)]\!]$ among P_0 and P_1, who use $[\|\vec{\boldsymbol{u}}_i'\|_2^2]$ and $[\alpha]$ to reconstruct $\|\vec{\boldsymbol{u}}_i'\|_2^2 - \alpha$. Next, P_0 and P_1 *cyclically shift* the full-domain evaluation of $[\![(\alpha, 1)]\!]$ by $\|\vec{\boldsymbol{u}}_i'\|_2^2 - \alpha$, effectively moving the target point of the DPF from $(\alpha, 1)$ to $(\|\vec{\boldsymbol{u}}_i'\|_2^2, 1)$. Finally, both parties use their shifted full-domain evaluations as CGKS-like queries against coeffs, yielding shares of $\pm(m, b)$ such that $m\|\vec{\boldsymbol{u}}_i'\|_2^2 + b$ is a good linear approximation for $1/\|\vec{\boldsymbol{u}}_i'\|_2$.[5]

To improve efficiency of the DPF steps (at the cost of a nominal loss of accuracy), our implementation truncates low-order bits from $[\|\vec{\boldsymbol{u}}_i'\|_2^2]$ to yield shares having just 6 bits of fractional precision (and we support at most 9 bits for the integer part), yielding shares over $\mathbb{Z}_{2^{16}}$ for use in choosing the liner approximation; we still compute $[m][\|\vec{\boldsymbol{u}}_i'\|_2^2] + [b]$ using all 64 bits of $[\|\vec{\boldsymbol{u}}_i'\|_2^2]$. Even with this truncate-to-16-bits optimization, we can always approximate inverse square roots with relative error less than 10^{-5}.

5 Astute readers might notice a similarity between this technique and the works of Krips and Willemson [34] and Kerik, Laud, and Randmets [31]. Those works also approximate elementary functions via piecewise-linear approximations; however, they rely on expensive, multi-round bit-decomposition protocols that are significantly more costly than our DPF-based approach.

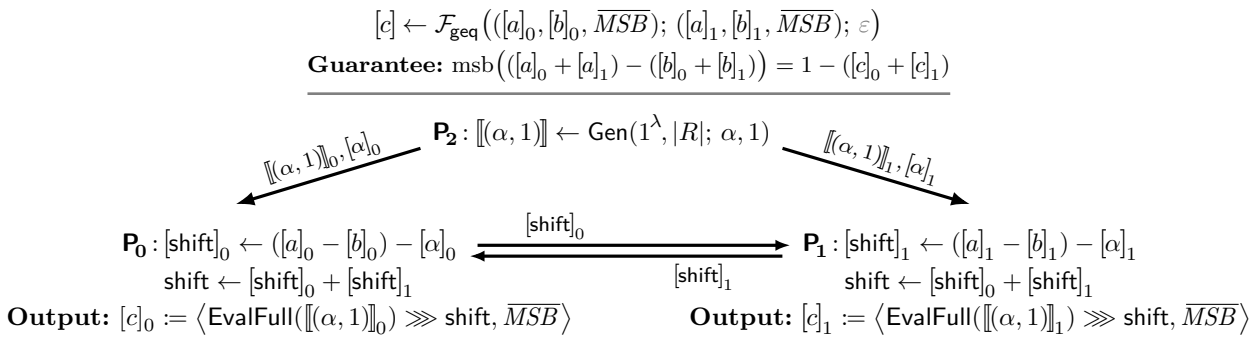


Fig. 3. 3-party comparison protocol, as described in Section 7.1. In the diagram, '$\ggg$' is the cyclic right shift operator.

7 Providing recommendations

We now describe our 3PC for providing recommendations based on user profile $\vec{\boldsymbol{u}}_i$ and item-profile matrix $\boldsymbol{V} = \left[\vec{\mathbf{v}}_1; \cdots; \vec{\mathbf{v}}_r\right]$. The strategy, at a high level, is to sample recommendations via the discrete probability distribution whose *probability mass function* is given by (normalizing) the vector $pmf_i \coloneqq \langle p_{i,1}, \ldots, p_{i,r} \rangle$ in which each $p_{i,j} \in [0,1]$ is the inner product between $\vec{\boldsymbol{u}}_i$ and $\vec{\mathbf{v}}_j$.

To this end, P_0 and P_1 enlist the help of P_2 to compute a sharing $[pmf_i]$ using r parallel generalized Du-Atallah invocations. Next, they non-interactively transform $[pmf_i]$ into shares of a (scaled) *cumulative distribution function* by taking partial sums; that is, they compute shares of

$$cdf_i \coloneqq \left\langle p_{i,1}, p_{i,1} + p_{i,2}, \ldots, \textstyle\sum_{j=1}^{r} p_{i,j} \right\rangle.$$

To sample from the distribution, it now suffices for the user to choose and submit a uniform random $u \in_{\mathrm{R}} [0,1] \cap \mathbb{R}$ to P_0 and P_1 (in cleartext). P_0 and P_1 scale the given u by $\sum_{j=1}^{r} p_{i,j}$, and then they run a 3-party protocol to find and return the index j of the smallest number in cdf_i that is greater than or equal to this scaled u. For this last step, they employ a novel 3PC method to evaluate the greater-than-or-equal-to operator (over $\mathbb{Z}$) on $(2,2)$-additive shares.

7.1 3-party integer comparison

The task of MPC-based integer comparison has received considerable attention. Most prior works (e.g., [8, 15, 22, 38]) implement comparison using (inherently costly) *bit decomposition* as a subroutine. One notable exception is the constant-round construction of Nishide and Ohta [38]; however, their construction is specific to secret sharing over *odd-order* rings and is, consequently, not applicable to our setting.

Our new 3PC comparison protocol is reminiscent of our 3-party fixed-point vector normalization protocol (§6.4.2). Specifically, our method checks whether $a \geq b$ over $\mathbb{Z}$ by using a modified PIR protocol to map the difference $a - b$ to its most-significant bit, which will equal 1 if and only if $a - b$ overflows (which happens if and only if $a < b$).

Suppose P_0 and P_1 hold $(2,2)$-additive sharings $[a]$ and $[b]$; the goal is for P_0 and P_1 to output $[1]$ if $a \geq b$ and $[0]$ otherwise. To this end, consider the "database" $\overline{MSB} = 1^{|R|/2} \| 0^{|R|/2} \in \mathbf{GF}(2)^{|R|}$ mapping each possible value of $a - b \in R$ to $1 - \mathrm{msb}(a-b)$. From here, the three parties employ a modified PIR protocol to fetch the answer from $\overline{MSB}$, as illustrated in Figure 3: P_2 samples $\alpha \in_{\mathrm{R}} R$ and shares $[\alpha]$ and $[\![(\alpha,1)]\!]$ among P_0 and P_1. Next, P_0 and P_1 use their respective shares to reconstruct the value $\mathsf{shift} \coloneqq (a-b) - \alpha$.

Finally, P_0 and P_1 compute the full-domain evaluations of $[\![(\alpha,1)]\!]_0$ and $[\![(\alpha,1)]\!]_1$, cyclically shift them by shift, and then output their respective inner products with $\overline{MSB}$. Similar to the normalization protocol, the cyclic shift effectively moves the DPF target from α to $a - b$, while the inner product maps the result to 1 if the most-significant bit of this new target is 0, and to 0 otherwise.

7.2 Avoiding duplicate recommendations

Using the above strategy, PIRSONA will frequently recommend "duplicate" items—that is, items that the user has previously fetched. To avoid such unhelpful recommendations, PIRSONA servers can "zero out" the $\mathtt{j}$th element of pmf_i for each item j previously downloaded by user i. The procedure is as follows: P_0 and P_1 hold (i) a $(2,2)$-additive sharing $[pmf_i]$, and (ii) a running sum $\vec{\mathsf{e}} \in \big(\mathbf{GF}(2)\big)^r$ of flag vectors derived from the queries of user i. They use Du-Atallah to compute the product of each component of pmf_i with the corresponding component of $\langle 1, 1, \ldots, 1 \rangle - \vec{\mathsf{e}}$. This effectively "zeros out" the

components of pmf_i corresponding to elements that user i has previously fetched, while leaving all other components unchanged. The resulting vector is then used to compute cdf_i and render recommendations as described above.

8 Implementation & evaluation

In order to validate the practicality of PIRSONA, we wrote a proof-of-concept implementation in C++ using the open-source library dpf++ [27] for $(2, 2)$-DPFs and Asio for asynchronous TLS communication [32].[6]

Experimental setup. We ran PIRSONA on Amazon EC2 servers running in four geographically distant regions; namely, we ran P_0 in the us-west-2 region (Oregon), P_1 in the us-east-1 region (North Virgina), P_2 in the eu-west-1 region (Ireland), and P_3 in the us-east-2 region (Ohio). Each party ran on its own m5.2xlarge instance (8 vCPUs and 32GiB of RAM) running the standard Ubuntu 18.04 LTS AMI.

Most experiments use *MovieLens* [25], a standard dataset collection used as a benchmark by the collaborative filtering research community. Specifically, we used two datasets from *MovieLens*, each consisting of $q = 100\,000$ queries:

1. **ML-100K** with $m = 943$ users and $r = 1\,700$ items; and
2. **ML-Latest** with $m = 610$ users and $r = 9\,000$ items.

The first dataset is the same one used by Nikolaenko, Ioannidis, Weinsberg, Joye, Taft, and Boneh [36] in the context of garbled circuit–based collaborative filtering, which is perhaps the most closely related work to PIRSONA in the literature. The second dataset is similar, but it includes over a factor $5\times$ more items, which greatly affects the performance of our DPF-based MUX and DeMUX constructions[7]. We remark that data in the *MovieLens* datasets contain explicit *numerical ratings*; however, as we deal exclusively with collaborative filtering based on Boolean matrix factorization, we replace all numeric ratings with the value 1. To measure how PIRSONA scales with a varying number of queries, profile dimensions, or items in the database, we also ran a handful of experiments over synthetic (random) data of varying sizes.

Table 1. Communication cost (in bytes) for q queries and x normalizations when profiles have dimension d.

Sender \ Receiver	P_0	P_1	P_2	P_3
P_0	—	$q(24d+24)+2^9\lambda x$	$2x$	$8q+2^9\lambda x$
P_1	$q(24d+24)$ $+2^9\lambda x$	—	$2^9\lambda x$	$8q+2x$
P_2	$2\lambda + q(32d+$ $24)+16x$	$4\lambda + q(16d+$ $24)+16x$	—	—
P_3	$4\lambda + 16x$	$\lambda + 16qd + 16x$	—	—

Each of our experiments measures the (wall-clock) training times for a given dataset, which includes both processing time and network latency. We repeated all experiments for 100 trials and our plots and tables report the sample mean running timings across those 100 trials, with error bars indicating one standard deviation from the sample mean. In tables and when discussing selected statistics in the text, we report all figures to one digit of precision in the sample standard deviation. Except where otherwise specified, all experiments were run with $p = 18$ bits of precision for the fractional part of user- and item-profile components, leaving $64 - 3 \cdot 18 - 1 = 9$ bits for integer parts when accumulating adaptation values.

8.1 Bandwidth consumption

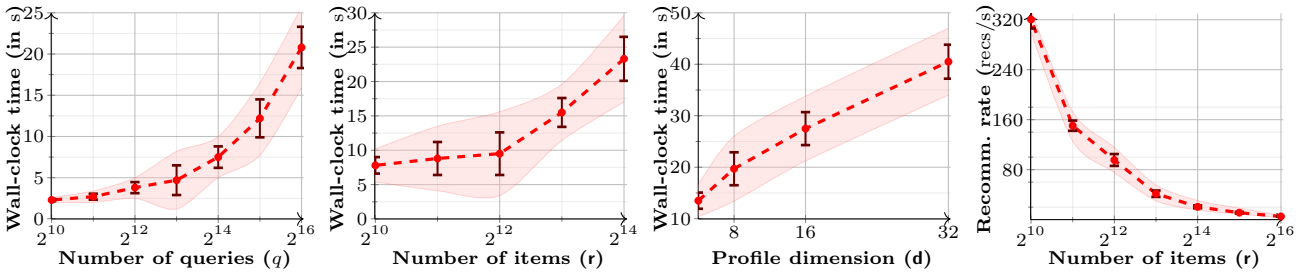
Before presenting the results of our running-time experiments, we briefly analyze the bandwidth consumption of PIRSONA. Recall that P_2 sends mostly random blinding factors and cancellation terms to P_0 and P_1; as the DPFs already assume the existence of PRGs, PIRSONA reduces the communication cost by having P_2 send λ-bit PRG seeds to P_0 and P_1 so that they can generate blinding factors locally. The total server-to-server bandwidth consumption, accounting for this optimization, is summarized in Table 1.

8.2 Scalability microbenchmarks

Our first running-time experiments measure the training time of PIRSONA over synthetic (random) data, with the goal of isolating the effect of varying a single parameter when all others are fixed. To this end, we ran three sets of experiments respectively measuring the impact

6 Our GPLv3-licensed source code is available via https://pr.iva.cy/git/pirsona

7 Note that PIRSONA's performance is agnostic to the actual data and is only dependent on the number of items and the dimensions of the profiles.



(a) Varying the number of queries from $m = 943$ users over $r = 2^{12}$ items using profiles of dimension $d = 8$.

(b) Varying the number of items with $q = 2^{14}$ queries from $m = 943$ users using profiles of dimension $d = 8$.

(c) Varying the profile dimension with $q = 2^{16}$ queries from $m = 943$ users over $r = 2^{10}$ items.

(d) Recommendation rate (per s) as the number of items varies using profiles of dimension $d = 8$.

Fig. 4. Wall-clock training time and recommendation rate for scalability microbenchmarks. Error bars show ± one standard deviation; the shaded bands are 95% confidence intervals ($z = 1.96$). Notice that the x-axes of Figures 4a, 4b, and 4d use a log scale with base 2.

on training time of varying (i) the number of queries being trained on, (ii) the number of items r in the database, and (iii) the dimension d of the profiles. We also measured the rate at which PIRSONA can render recommendations while varying the number of items r.

Varying the number of queries. Figure 4a plots the (wall-clock) training time for $m = 943$ users, $r = 2^{12}$ items, and profiles of dimension $d = 8$ as the number of queries ranges from $q = 2^{10}$ up to 2^{16}. As expected, the measured training times scale linearly (linear regression gives $R^2 = 0.9972$) from about 2.3 ± 0.1s for $q = 2^{10}$ queries up to 21 ± 3s for $q = 2^{16}$ queries (an effective rate of about 3500 queries/s).

Varying the number of items. Figure 4b plots the (wall-clock) training time for $q = 2^{14}$ queries, $m = 943$ users, and profiles of dimension $d = 8$ as the number of items ranges from $r = 2^8$ up to 2^{14}. Again we observe the expected linear scaling (linear regression gives $R^2 = 0.9889$) from about 8 ± 1s for $r = 2^8$ items through 28 ± 3s for $r = 2^{14}$ items ($\approx$ 1s slowdown per 1000 items).

Varying the profile dimensions. Figure 4c plots the (wall-clock) training time for $q = 2^{16}$ queries, $m = 943$ users, and $r = 2^{10}$ items as the dimension of user and item profiles ranges from $d = 4$ up to 32. Again, the measured training times (after $d = 8$) scale linearly (linear regression gives $R^2 = 0.9978$) from about 20 ± 3s for $d = 8$ up to 41 ± 3s for $d = 32$ ($\approx$ 0.9s slowdown per profile component). The initially steeper slope between $d = 4$ and 8 results from the former profiles occupying a single 256-bit AVX2 register and the latter spanning multiple such registers.

Recommendation rate. Figure 4d plots the number of recommendations rendered per second for profiles of dimension $d = 8$ as the number of items ranges from $r = 2^{10}$ up to 2^{16}. We observe slightly superlinear scaling as the number of items grows from about 320 ± 20 recommendations/s for $r = 2^{10}$ items down to 5.1 ± 0.2 recommendations/s for $r = 2^{16}$ items.

8.3 MovieLens experiments

Table 2 displays the training and recommendation times for PIRSONA on the two *MovieLens* datasets. Recall that both datasets include 100000 queries; thus, even the slowest experiment (**ML-Latest** with profiles of dimension $d = 8$) had mean training throughput exceeding 1400 queries/s. We also compared the training time of PIRSONA against that of our own (lightly optimized) non-private collaborative filtering implementation and found that PIRSONA ran around 150× slower. We forgo a detailed comparison because significant speedups to the non-private code are surely possible.

The rightmost column in Table 2 displays the mean time for providing one recommendation to *each* user, of which there are $m = 943$ in **ML-100K** and $m = 610$ in **ML-Latest**. For the **ML-100K** dataset, this works out to 84 ± 4 recommendations/s when profiles have dimension $d = 4$ and 71 ± 5 recommendations/s when they have dimension $d = 8$; for the **ML-Latest** dataset, these rates decrease to 32 ± 4 recommendations/s when profiles have dimension $d = 4$ and 31 ± 5 recommendations/s

when they have dimension $d = 8$, due to the comparatively large number of items ($r = 9000$ versus $r = 1700$).

Recall from Section 8.2 that the time to do gradient descent scales linearly with the profile dimension. Therefore, although we report our results only for $d = 4$ and $d = 8$, one can reliably extrapolate to any required dimension. The recommended number of the dimension is dependent on the complexity of the domain.[8]

Mean squared error. PIRSONA's use of fixed-point arithmetic introduces approximation errors relative to a non-private collaborative filtering implementation using floating-point arithmetic. To quantify the effects of fixed-point approximations, we compared the *mean squared error* between the user and item profiles produced by PIRSONA on the **ML-100K** dataset relative to those produced by our non-private collaborative filtering implementation, which uses IEEE double-precision floating-point arithmetic. A similar metric was used to measure fixed-point approximation errors in the related literature [36, 37]. We also measured what effect varying the fractional precision had on training times, since higher fractional precision necessitates more frequent precision reductions; fortunately, this difference was statistically insignificant with only $p = 18$ exhibiting a *slightly* higher sample mean. The mean squared error was minuscule with $p = 14$ or more bits (whereas below $p = 14$ bits, it rapidly approached 1).

Table 2. Wall-clock running times for *MovieLens*. The rightmost column is time to make one recommendation per user.

Data Set	Profile dims.	Grad. descent	User prof. normalize	Item prof. normalize	Recomm. m items
ML-100K	4	22 ± 4s	0.7 ± 0.4 s	1.8 ± 0.1s	11 ± 4s
	8	32 ± 6s	0.95 ± 0.09s	2.0 ± 0.2s	13 ± 3s
ML-Latest	4	49 ± 4s	0.53 ± 0.08s	3.5 ± 0.4s	21 ± 3s
	8	65 ± 6s	0.7 ± 0.1 s	4.2 ± 0.7s	22 ± 4s

8 For reference, Bell and Koren [5] showed good results for dimension $d = 20$ in their winning Netflix Prize Challenge submission.

Table 3. Mean squared error and training time for select fractional precisions on the **ML-100K** dataset with profile dimension $d = 8$.

Fractional precision	Running time	User prof. MSE	Item prof. MSE
18	35 ± 8s	4×10^{-5}	4.1×10^{-5}
16	32 ± 6s	2×10^{-5}	1.8×10^{-5}
14	32 ± 6s	1.5×10^{-4}	1.7×10^{-4}
12	32 ± 6s	1.4×10^{-2}	1.5×10^{-2}

8.4 Comparison with garbled circuits

The most closely related prior work is detailed in a manuscript by Nikolaenko, Ioannidis, Weinsberg, Joye, Taft, and Boneh [36]. They use *garbled circuits* to train similar collaborative filtering models as PIRSONA. Fundamental model differences render an apples-to-apples comparison impossible, yet it is instructive to note that Nikolaenko et al. report 2.9 hours to train on a subset of 14683 queries (out of 100000) from the **ML-100K** dataset—a full *five orders of magnitude* slower than PIRSONA.

9 A hypothetical deployment

The evaluation in Section 8 suggests that PIRSONA may be sufficiently performant to support recommendations for a large-scale streaming service. This section considers one hypothetical such deployment modeled after Netflix. We stress that this paper focuses on training and providing recommendations from collaborative filtering models in a PIR-based content-distribution system—not on the design of a specific content-distribution system. We consider a Netflix-like system because (i) video streaming is exceedingly popular and is growing more so each day, and (ii) the bandwidth and latency demands of video streaming make it an exceedingly ambitious target for a PIR-based realization. In other words, practicality for video streaming implies practicality for audio streaming, e-book downloads, software repositories, and other related services.

When the user launches our hypothetical video streaming service, their client software (perhaps browser-based or perhaps a Smart TV app) presents a landing page that lists all titles available for streaming. The client software automatically requests a batch of

recommendations (produced via the machinery developed in Section 7) to display under a "Based on Your Viewing History" banner, much like how Netflix currently provides its recommendations. Upon selecting a video to stream, the client software automatically samples a Hafiz–Henry query to fetch that video, and the content distributor transparently updates the user and item profiles behind the scenes.

Thus, the user experience with such a service would be virtually indistinguishable from that of Netflix—*provided the PIR-based video streaming is sufficiently performant*. The bottleneck in Hafiz–Henry PIR is I/O throughput; thus, for optimal performance, we want the entire PIR database to fit in *physical* memory. To reconcile this requirement with the massive size of video streaming libraries, we consider a "striped" layout in which the first N_1 minutes of each title is served from a dedicated PIR instance; if the user is still watching when the N_1-minute mark draws near, the first instance hands the query off to a second dedicated instance serving the next N_2 minutes of each title, and so on.[9]

According to Netflix, at the time of writing, HD video streaming uses "up to 3GB" per hour; thus, we assume an upper bound of $3\text{GB}/60 = 50\text{MB}$ per minute of video. This means that we could store the first $N_1 = 0.5$ minutes of each title comprising the **ML-Latest** dataset on servers with 225GB of RAM. Assuming the bit rates and "striping" strategy above, we performed microbenchmarks to estimate the number of concurrent viewers that could stream from the **ML-Latest** dataset. Specifically, we assume one 4-server Hafiz–Henry instance running on four geographically dispersed `m5.16xlarge` instances (each sporting 64 vCPUs and 256GiB of RAM) per 30 second stripe of video. In order to hide I/O latency, the servers process incoming queries in parallel batches of size 16. With this setup, we were able to process $58\,000 \pm 1\,100$ queries per minute. This equates to about $58000/(60 \cdot 16) \approx 60$ batches per second, suggesting that the PIR adds an expected delay of only about 17ms beyond normal Internet latency.

Monetary cost of deployment

We conclude this section with some back-of-the-envelope calculations to estimate the cost of deployment. Our calculations assume the measurements above with EC2 pricing (currently US$3.072/hour for an m5.16xlarge instance). For simplicity, we assume that each video is 1 hour (60 minutes) so that a total of $4 \cdot 2 \cdot 60 = 480$ servers suffice for every 58000 viewers that commences streaming a video in any given minute (or up to 3.48 million concurrent viewers). This equates to about US$0.0002 per viewer per hour, giving an upper bound of about US$0.022 per subscriber per month, assuming all subscribers streams HD video 24 hours a day, 7 days a week. (A typical Netflix account purportedly streams about 2 hours of video per day, on average.)

10 Concluding remarks

We presented PIRSONA, a digital content delivery system that realizes collaborative-filtering recommendations atop private information retrieval (PIR). This combination of seemingly antithetical primitives makes possible–for the first time–the construction of e-commerce and digital media delivery systems that can provide personalized content recommendations based on their users' historical consumption patterns while simultaneously keeping said consumption patterns private. A series of experiments in our proof-of-concept implementation suggest that PIRSONA can easily scale to systems serving thousands of users per second.

Toward building PIRSONA, we proposed several 4PC primitives of independent interest, including (i) efficient 2- and 4PC fixed–selection wire multiplexers (MUXs) and demultiplexers (DeMUXs) from $(2, 2)$-distributed point functions, (ii) fast 3PC normalization for secret-shared fixed-point vectors, (iii) 4PC well-formedness tests for $(2, 2)$-distributed point functions, and (iv) a novel, constant-round 3PC integer ("less than") comparison protocol.

Acknowledgements. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Science Foundation under Grant No. 1718475.

9 We note that the "striped" layout allows the servers to infer approximately how long a user watches a given stream, which might leak much information about what they were watching. This seems to be inherent: Either users download something as long as the longest video—which is wasteful—or they accept some leakage.

References

[1] Esma Aïmeur, Gilles Brassard, José Manuel Fernandez, and Flavien Serge Mani Onana. Privacy-preserving demographic filtering. In *Proceedings of ACM SAC 2006*, pages 872–878, Dijon, France (April 2006).

[2] Esma Aïmeur, Gilles Brassard, José Manuel Fernandez, and Flavien Serge Mani Onana. Alambic: A privacy-preserving recommender system for electronic commerce. *International Journal of Information Security (IJIS)*, 7(5):307–334, October 2008.

[3] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology: Proceedings of CRYPTO 1991*, volume 576 of *LNCS*, pages 420–432, Santa Barbara, CA, USA (August 1991).

[4] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151, March 2004.

[5] Robert M. Bell, Yehuda Koren, and Chris Volinsky. The BellKor 2008 solution to the Netflix Prize. Available from: https://www.netflixprize.com/assets/ProgressPrize2008_BellKor.pdf, 2008.

[6] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Advances in Cryptology: Proceedings of EUROCRYPT 1998*, volume 1403 of *LNCS*, pages 236–250, Espoo, Finland (June 1998).

[7] Simon R. Blackburn, Tuvi Etzion, and Maura B. Paterson. PIR schemes with small download complexity and low storage requirements. In *Proceedings of ISIT 2017*, pages 146–150, Aachen, Germany (June 2017).

[8] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206, Málaga, Spain (October 2008).

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of CCS 2016*, pages 1292–1303, Vienna, Austria (October 2016).

[10] Joseph A. Calandrino, Ann Kilzer, Arvind Narayanan, Edward W. Felten, and Vitaly Shmatikov. "You might also like:" Privacy risks of collaborative filtering. In *Proceedings of IEEE S&P 2011*, pages 231–246, Berkeley, CA, USA (May 2011).

[11] J. Canny. Collaborative filtering with privacy. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 45–57, May 2002.

[12] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Technical Report CS 0917, Technion-Israel Institute of Technology, Haifa, Israel, February 1997.

[13] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of FOCS 1995*, pages 41–50, Milwaukee, WI, USA (October 1995).

[14] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of IEEE S&P 2015*, pages 321–338, San Jose, CA, USA (May 2015).

[15] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proceedings of TCC 2006*, volume 3876 of *LNCS*, pages 285–304, New York, NY, USA (March 2006).

[16] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In *Proceedings of CCS 2017*, pages 523–535, Dallas, TX, USA (October–November 2017). ACM.

[17] Rafael Dowsley, Jeroen van de Graaf, Davidson Marques, and Anderson C. A. Nascimento. A two-party protocol with trusted initializer for computing the inner product. In *Proceedings of WISA 2010*, volume 6513 of *LNCS*, pages 337–350, Jeju Island, Korea (August 2010).

[18] Wenliang Du and Mikhail J. Atallah. Protocols for secure remote database access with approximate matching. In *E-Commerce Security and Privacy (Part II)*, volume 2 of *Advances in Information Security*, pages 87–111, February 2001.

[19] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *Proceedings of USENIX Security 2021*, Vancouver, BC, Canada (August 2021).

[20] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology: Proceedings of EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658, Copenhagen, Denmark (May 2014).

[21] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, October 1986.

[22] Ken Goss and Wei Jiang. Efficient and constant-rounds secure comparison through dynamic groups and asymmetric computations. *IACR Cryptology ePrint Archive*, Report 2018/179, February 2018.

[23] Syed Mahbub Hafiz and Ryan Henry. Querying for queries: Indexes of queries for efficient and expressive IT-PIR. In *Proceedings of CCS 2017*, pages 1361–1373, Dallas, TX, USA (October–November 2017).

[24] Syed Mahbub Hafiz and Ryan Henry. A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server PIR. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2019(4):112–131, January 2019.

[25] F. Maxwell Harper and Joseph A. Konstan. The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):Article No. 19, December 2015.

[26] Ryan Henry, Femi Olumofin, and Ian Goldberg. Practical PIR for electronic commerce. In *Proceedings of CCS 2011*, pages 677–690, Chicago, IL, USA (October 2011).

[27] Ryan Henry and Adithya Vadapalli. dpf++; version 0.0.1 [computer software]. Available from: https://www.github.com/rh3nry/dpfplusplus, July 2019.

[28] Russell Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structures 1995*, pages 134–147, Minneapolis, MN, USA (June 1995).

[29] Ari Juels. Targeted advertising... and privacy too. In *Proceedings of CT-RSA 2001*, volume 2020 of *LNCS*, pages 408–424, San Francisco, CA, USA (April 2001).

[30] Przemysław Kazienko and Michał Adamski. AdROSA—Adaptive personalization of web advertising. *Information Sciences*, 177(11):2269–2295, June 2007.

[31] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, volume 9604 of *Lecture Notes in Computer Science*, pages 271–287. Springer, 2016.

[32] Christopher M. Kohlhoff. Asio; version 1.14.1 [computer software]. Available from: http://think-async.com/Asio/, August 2019.

[33] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, August 2009.

[34] Toomas Krips and Jan Willemson. Hybrid model of fixed and floating point numbers in secure multiparty computations. *IACR Cryptology ePrint Archive*, 2014:221, 2014.

[35] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. In *Proceedings of PKC 2019 (Part I)*, volume 11442 of *LNCS*, pages 3–33, Beijing, China (April 2019).

[36] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *Proceedings of CCS 2013*, pages 801–812, Berlin, Germany (November 2013).

[37] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Proceedings of IEEE S&P 2013*, pages 334–348, Berkeley, CA, USA (May 2013).

[38] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Proceedings of PKC 2007*, volume 4450 of *LNCS*, pages 343–360, Beijing, China (April 2007).

[39] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *Proceedings of USENIX Security 2016*, pages 619–636, Austin, TX, USA (August 2016).

[40] Rafail Ostrovsky and Victor Shoup. Private information storage (Extended abstract). In *Proceedings of STOC 1997*, pages 294–303, El Paso, TX, USA (May 1997).

[41] Rupa Parameswaran and Douglas M. Blough. Privacy preserving data obfuscation for inherently clustered data. *Int. J. Inf. Comput. Secur.*, 2(1):4–26, 2008.

[42] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of NDSS 2010*, San Diego, CA, USA (February 2010).

A Related literature

There does not appear to be any prior work specifically addressing the problem recommendations in the context of PIR-based systems; however, past efforts have addressed other related problems. The fundamental difference between PIRSONA and all the prior work that we have come across in this area lies in the system design. The previous work in this area assumes that the system already has the data on which it runs the recommendation algorithm. Once the system has the data, there are a host of techniques like Garbled Circuits, MPC on secret shares, and fully homomorphic encryption that have been used in the prior work to provide recommendations. PIRSONA, on the other hand, relies only upon the PIR queries by the users to do the CF. The work by Nikolaenko, Ioannidis, Weinsberg, Joye, Taft, and Boneh [36] is the closest to PIRSONA. They also do CF to provide recommendations. While the underlying algorithm to compute the recommendations is the same, there are some key differences. Firstly, in their system, the users submit their encrypted ratings of the items they have downloaded to the system. Secondly, they use Garbled circuits on these encrypted ratings to do the CF. PIRSONA relieves the users of any such responsibility and does MPC on the repurposed PIR queries to compute the recommendations. This change in the system design comes with the following benefits: (i) The user experience in PIRSONA is more akin to a non-private system, where they simply download the desired items and have no additional responsibility, and (ii) Secondly, the switch from Garbled Circuits to MPC on secret shares gives a massive speedup.

Nikolaenko, Weinsberg, Ioannidis, Weinsberg, Joye, Boneh, and Taft also present a Garbled Circuit based ridge regression [37], which can be used to provide recommendations. The work by Parameswaran and Blough [41] uses CF as the underlying algorithm to compute recommendations. They too assume that the data on which the CF algorithm needs to be run is available to the system. Their main idea is to do a data-obfuscation in such a manner that, it preserves the clusters and they can do the CF on the obfuscated data.

Another approach to CF is taken by Canny [11]. They run the CF algorithm on the user side. A group of users aggregates their data using a Fully Homomorphic Encryption scheme to get personalized recommendations.

There is a line of work in recommender systems that is dependent on trusted hardware. Trusted hardware like

Intel's *Software Guard eXtensions* (SGX) has been used to build recommender systems; for example, in a series of papers by Aïmeur, Brassard, Fernandez [1, 2]. Ohrimenko, Schuster, Fournet, Mehta, Nowozin, Vaswani, and Costa [39] perform various machine learning algorithms like support vector machines, neural networks, k-means clustering using trusted SGX-processors.

B 4-party DeMUX w/ sanity checks

The 2-party DeMUX assumes that the seeds held by P_0 and P_1 constitute a well-formed leafless DPF; that is, it assumes that, $\mathsf{flags}(\llbracket(j,\star)\rrbracket_0) \oplus \mathsf{flags}(\llbracket(j,\star)\rrbracket_1) = \vec{e}_j$ and $\overline{cw}_h$ is indeed the leaf-layer correction word for $\llbracket(j,0)\rrbracket$. If not, the resulting output wires may take on arbitrary values, corrupting the entire item-profile matrix. The 4-party DeMUX augments its 2-party counterpart with a probabilistic *sanity check* consisting of two main components that respectively check that:

1. $\exists j \in [1..r]$, $\mathsf{flags}(\llbracket(j,\star)\rrbracket_0) \oplus \mathsf{flags}(\llbracket(j,\star)\rrbracket_1) = \vec{e}_j$, and
2. the leaf-layer CW sharing $[\overline{cw}_h]$ is well formed.

Both components borrow well-known ideas from batch testing [6], although the way they implement these ideas as 4-party interactive protocols appears to be new. The components of the sanity check run in tandem; however, for clarity and ease of exposition, we treat each step in isolation.

B.1 Testing the flag bits

The first component of the sanity check tests if $\mathsf{flags}(\llbracket(j,\star)\rrbracket_0) \oplus \mathsf{flags}(\llbracket(j,\star)\rrbracket_1) = \vec{e}_j$ for some $j \in [1..r]$; in other words, it checks if $\llbracket(j,\star)\rrbracket$ defines a valid 2-party MUX (see §6.2.1). Intuitively, the idea is to have P_3 select a vector $\vec{R} \in_R [0..2^\mu - 1]^r$, which it sends to P_0 and P_1; then, treating $\vec{R}$ a "random database", P_0 and P_1 respectively use $\mathsf{flags}(\llbracket(j,\star)\rrbracket_0)$ and $\mathsf{flags}(\llbracket(j,\star)\rrbracket_1)$ as queries in a 2-server instance of the celebrated Chor–Gilboa–Kushilevitz–Sudan (CGKS) PIR protocol [13], forwarding the results for P_3 to reconstruct. If the CGKS responses provided to P_3 reconstruct to a component of $\vec{R}$, then P_3 instructs P_0 and P_1 to accept $\llbracket(j,\star)\rrbracket$ as valid.

The problem with this simplistic protocol is that it leaks to P_3 the index j of the standard basis vector $\vec{e}_j$. To prevent this, P_0 and P_1 enlist the help of P_2, who selects a blinding factor γ and a random cyclic permutation $\pi\colon [1..r] \to [1..r]$, and then sends both of these values to P_0 and P_1—but not to P_3. P_0 and P_1 use π to *cyclically shift* the components of $\mathsf{flags}(\mathsf{seed}_0)$ and $\mathsf{flags}(\mathsf{seed}_1)$ before using them as CGKS queries, and then they blind the CGKS responses with γ before sending them to P_3. Intuitively, the permutation π severs the link between the index j and the matching component $\vec{R}[\pi(j)]$, while γ prevents P_3 from inferring π using $\llbracket(j,\star)\rrbracket_1$—which it also knows—in conjunction with $\vec{R}$ and the response it receives from P_1. Figure 5 illustrates this component of the sanity check. Note that the same batch-testing vector $\vec{R}$ can be used to check arbitrarily many DPFs, provided (i) $\vec{R}$ is sampled *after* the DPFs are generated and (ii) each run of the test uses fresh choices for π, γ.

A simple derivation confirms that P_3 accepts whenever $\llbracket(j,\star)\rrbracket$ is indeed a well-formed (leafless) DPF, as output by $\mathsf{G}\overline{\mathsf{en}}$. The next observation follows from a standard counting argument [26, Appendix A].

Observation 2. *Let* $\llbracket(j,\star)\rrbracket_0$ *and* $\llbracket(j,\star)\rrbracket_1$ *be fixed, and let* $\vec{R} \in_R [0..2^\mu - 1]^r$. *If* $\mathsf{flags}(\llbracket(j,\star)\rrbracket_0) \oplus \mathsf{flags}(\llbracket(j,\star)\rrbracket_1) \neq \vec{e}_j$, *then* $\Pr\big[\langle \mathsf{flags}(\llbracket(j,\star)\rrbracket_0), \vec{R}\rangle \oplus \langle \mathsf{flags}(\llbracket(j,\star)\rrbracket_1), \vec{R}\rangle = \vec{R}[j]\big] = 2^{-\mu}$.

Corollary 1. *If* $\mathsf{flags}(\llbracket(j,\star)\rrbracket_0) \oplus \mathsf{flags}(\llbracket(j,\star)\rrbracket_1)$ *is not a standard basis vector, then* P_3 *rejects with probability at least* $1 - r/2^\mu$.

We remark that Boyle et al. proposed a 2-party sanity check [9, §4] based on linear sketching and 2PC; recent work by Eskandarian, Corrigan-Gibbs, Zaharia, and Boneh [19] modifies this test by replacing 2PC with a 2-party SNIP. Yet another (less efficient) sanity check for square-root–sized DPFs was proposed by Corrigan-Gibbs, Boneh, and Mazières [14, §5.1]. Our sanity check uses similar probabilistic batch-testing ideas, but it provides vastly superior performance by taking advantage of the additional non-colluding computation parties at our disposal. Indeed, whereas the Boyle et al.'s linear sketch uses about $2r$ modular multiplications and r modular subtractions, the cost of our protocol is dominated by an expected $r/2$ exclusive-ORs on similar-sized elements. (Note that every "multiplication" computed as part of the above inner products uses a coefficient of either 0 or 1.)

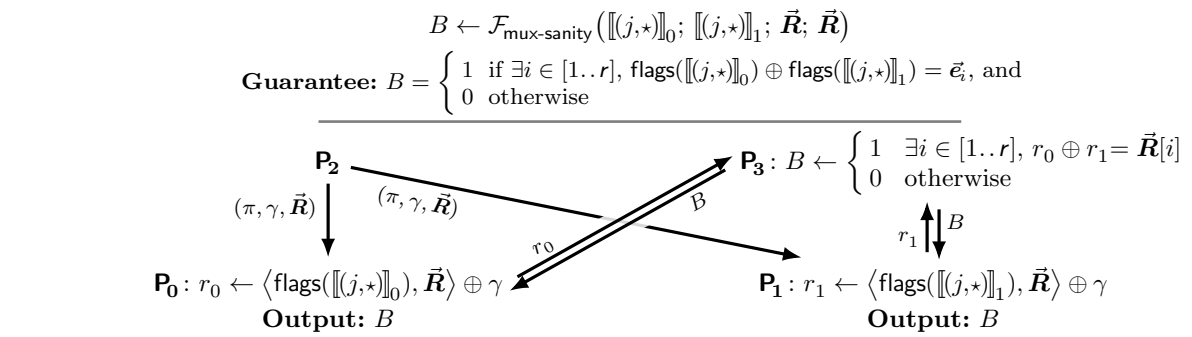


Fig. 5. The first component of the 4-party DPF sanity check protocol, as described in Appendix B.1.

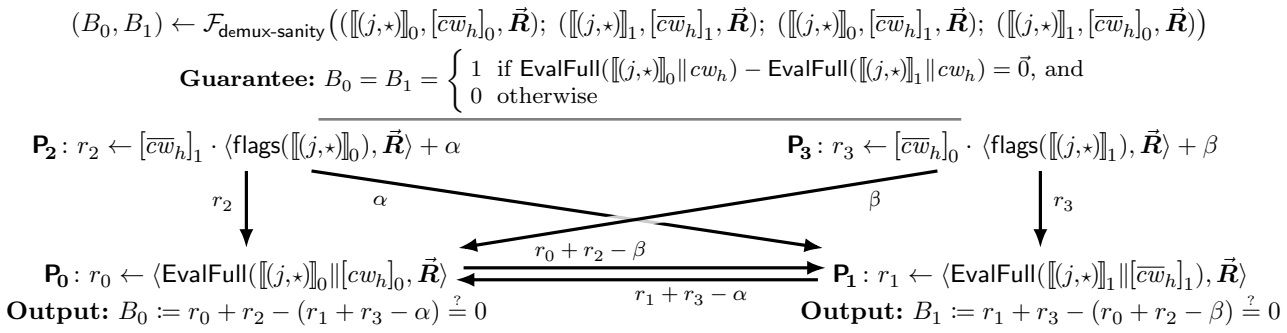


Fig. 6. Second component of 4-party DPF sanity check, as described in Appendix B.2.

B.2 Testing the leaf-layer CW shares

The second component of the sanity check employs the same "mix-and-match" pattern as the 4-party MUX (§6.2.2) to test if the leaf-layer CW shares are well formed. Suppose P_0 and P_1 respectively hold $(\llbracket(j,\star)\rrbracket_0, [\overline{cw}_h]_0)$ and $(\llbracket(j,\star)\rrbracket_1, [\overline{cw}_h]_1)$, while P_2 and P_3 respectively hold $(\llbracket(j,\star)\rrbracket_0, [\overline{cw}_h]_1)$ and $(\llbracket(j,\star)\rrbracket_1, [\overline{cw}_h]_0)$; the goal is to verify that indeed

$$\vec{0} \stackrel{?}{=} \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_0 \| \overline{cw}_h) - \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_1 \| \overline{cw}_h)\,. \quad (4)$$

By construction [9, Fig. 1], we have that

$$\begin{aligned} \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_0 \| [\overline{cw}_h]_0) &= \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_0 \| \vec{0}) \\ &\quad + [\overline{cw}_h]_0\, \mathsf{flags}(\llbracket(j,\star)\rrbracket_0) \\ &= \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_0 \| [\overline{cw}_h]_1) \\ &\quad + ([\overline{cw}_h]_0 - [\overline{cw}_h]_1)\, \mathsf{flags}(\llbracket(j,\star)\rrbracket_0)\,, \end{aligned}$$

and, symmetrically, that

$$\begin{aligned} \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_1 \| [\overline{cw}_h]_1) &= \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_1 \| [\overline{cw}_h]_1 - [\overline{cw}_h]_0) \\ &\quad + [\overline{cw}_h]_0\, \mathsf{flags}(\llbracket(j,\star)\rrbracket_1)\,; \end{aligned}$$

hence, Equation (4) holds if and only if

$$\begin{aligned} \vec{0} \stackrel{?}{=}\ & \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_0 \| [\overline{cw}_h]_0) + [\overline{cw}_h]_1\, \mathsf{flags}(\mathsf{seed}_0) \\ & - \mathsf{EvalFull}(\llbracket(j,\star)\rrbracket_1 \| [\overline{cw}_h]_1) - [\overline{cw}_h]_0\, \mathsf{flags}(\mathsf{seed}_1)\,. \end{aligned} \quad (5)$$

Disclosing the four vectors comprising Equation (5) to a single party would leak both $\overline{cw}_h$ and the value on the selection wire—to say nothing of the prohibitively high communication overhead. Instead, each of the four parties computes the inner product of their respective vectors with a random vector $\vec{R}$ and then they use a 2-move protocol to check that the resulting inner products sum to 0. Note that the same random vector $\vec{R}$ can be (i) shared among the two components of the test and (ii) reused to test an arbitrary number of DPFs. Figure 6 illustrates this component of the sanity check.

C Simulators for subprotocols

In this section we provide the security proofs for MPC subprotocols used in this paper. These proofs rely on the simulatability of DPFs (see [9] for the proofs that DPFs are secure).

C.1 2-Party MUX

In the 2-Party MUX there is no communication between P_0 and P_1. Therefore, the security of the 2-Party MUX,

directly follows from the security of Distributed Point Functions.

C.2 4-Party MUX

Lemma 1. *There exists a simulator which simulates the views of* P_0, P_1, P_2 *and* P_3 *in* $\mathcal{F}_{\mathsf{mux}}$ *(4-Party Multiplexer).*

Proof. The simulator works as follows. Parties P_2 and P_3 receive no messages in the protocol.

1. P_0 receives a blinded profile from P_2, and a blind of profile size from P_3.
2. Similarly, P_1 receives a blinded profile from P_3, and a blind of profile size from P_2.

Therefore, the simulation for P_b is pair of uniformly generated random profiles. □

C.3 MUX-sanity check

Lemma 2. *There exists a simulator which simulates the views of* P_0, P_1, P_2, *and* P_3, *in* $\mathcal{F}_{\mathsf{mux\text{-}sanity}}$ *(the MUX-Sanity Check).*

Proof. The first component of the DPF sanity check verifies that $\mathsf{flags}(\mathsf{seed}_0)$ and $\mathsf{flags}(\mathsf{seed}_1)$ are indeed the shares of a standard basis vector. P_2 does not have any output. Parties $\mathsf{P}_0, \mathsf{P}_1, \mathsf{P}_3$ have an output $B \in \{0,1\}$. $B = 1$ indicates that the $\mathsf{flags}(\mathsf{seed}_0)$ and $\mathsf{flags}(\mathsf{seed}_1)$ are shares of a standard basis vector and $B = 0$ indicates otherwise. P_b (for $b \in \{0,1\}$)) receives a random permutation, a blinding factor and random vector from P_2.

The simulation for P_0 and P_1 are (i) a random permutation vector, (ii) a uniformly random vector, and (iii) a uniformly random blinding factor.

The simulation for P_3 is conditioned on the value of B. (i) If $B = 0$, then the simulation for P_3 is pair of random numbers, and (ii) If $B = 1$, then the simulation is r_0 generated uniformly at random and $r_1 \leftarrow r_0 \oplus \vec{\boldsymbol{R}}[i]$ for some i. □

C.4 DeMUX sanity check

Lemma 3. *There exists a simulator which simulates the views of* P_0, P_1, P_2, *and* P_3 *in* $\mathcal{F}_{\mathsf{demux\text{-}sanity}}$ *(the DeMUX Sanity Check).*

Proof. P_2 and P_3 do not receive any messages in the protocol.

The simulation for P_0 works as follows. The secret input for P_0 is its seed for Evalfull and $\vec{\boldsymbol{R}}$.

1. P_0 receives something completely random from P_2 and something completely random from P_3. Call them r_2 and r_3 respectively.
2. P_0 computes the Evallfull on its seed and takes the dot-product with $\vec{\boldsymbol{R}}$. Call it r.
3. P_0 receives r_0 from P_1 with the property that, $(r_0 - (r + r_2)) = 0$.

Therefore, the simulation for P_0 is (i) a random r_2 (simulating the message from P_2, (ii) a uniformly random β (simulating the message from P_3, and (iii) $r_0 = -(r_2 + \mathsf{EvalFull}(\llbracket\langle j,\star\rangle\rrbracket_0 \| (cw_h - \delta)), \vec{\boldsymbol{R}}\rangle)$. The simulation for P_1 is symmetrical to that of P_0. □

C.5 3PC vector normalization

Lemma 4. *There exists a simulator which simulates the views of* P_0, P_1, *and* P_2 *in* $\mathcal{F}_{\mathsf{isqrt}}$ *(the 3-party inverse square root).*

Proof. P_2 does not receive any messages in the protocol. P_0 and P_1 receive a DPF key, a blinding factor and share of a sign bit.

The simulation for P_0 and P_1 is (i) A random DPF key., (ii) A uniformly random number., and (iii) A uniformly random bit.

□

C.6 3PC integer comparison

Lemma 5. *There exists a simulator which simulates the views of* P_0, P_1, *and* P_2 *in* $\mathcal{F}_{\mathsf{geq}}$ *(the 3-party Comparison).*

Proof. P_2 does not receive any messages in the protocol. P_0 and P_1 receive a DPF key and blinding factor.

The simulation for P_0 and P_1 is (i) A random DPF key, and (ii) A uniformly random number. □

D Security analysis

This appendix sketches a proof of security for PIRSONA in the ideal-world/real-world simulation paradigm.

In the ideal world, all PIRSONA users hand their requests directly to some benevolent trusted party, $\mathcal{T}$, who provides honest answers and correctly updates user and item profiles in response to incoming requests. We consider an attacker $\mathcal{A}$ who corrupts an arbitrary subset of PIRSONA users, adaptively choosing corrupted users' requests to $\mathcal{T}$ and examining $\mathcal{T}$'s responses in a bid to draw inferences about the behaviour of one or more non-corrupted users.

In the real world, we replace $\mathcal{T}$ with the multi-server PIRSONA protocols described in the main text. We also permit the attacker to corrupt one of the PIRSONA servers (in addition to an arbitrary subset of users). As in the ideal world, $\mathcal{A}$ seeks to draw inferences about non-corrupted users—only now $\mathcal{A}$ can leverage any information learned through its privileged position as a PIRSONA server to help in drawing these inferences. We stress that $\mathcal{A}$ is assumed to be PPT and may only employ semi-honest strategies; that is, it logs, analyzes, and exploits any information it encounters during the PIRSONA execution, but it otherwise follows all MPC sub-protocols precisely as specified.

Informally, we wish to show that operating in the real world provides $\mathcal{A}$ with no tangible advantage over operating in the ideal world, where all inferences about non-corrupted users necessarily result from model leakage inherent to the underlying collaborative filtering recommendations—not from the MPC components PIRSONA uses to realize those recommendations. We demonstrate this by constructing an efficient *simulator* that uses $\mathcal{A}$'s ideal-world views to sample "simulated" real-world views for $\mathcal{A}$. The existence of such a simulator guarantees that any inferences $\mathcal{A}$ can draw in the real-world, it can just as easily draw in the ideal world.

PIRSONA in the ideal world

Each user i interacts with the ideal-world trusted party $\mathcal{T}$ through a confidential, point-to-point channel (so that $\mathcal{T}$ always knows with which user it is interacting) via two public interfaces: $\mathcal{T}.\mathsf{Query}_i(\cdot)$ and $\mathcal{T}.\mathsf{Recommend}_i(\cdot)$. In addition to servicing requests via these interfaces, $\mathcal{T}$ curates (i) the database $\boldsymbol{D} \coloneqq \left[\vec{\boldsymbol{D}}_1; \cdots; \vec{\boldsymbol{D}}_r\right]$, (ii) the item- and user-profile matrices $\boldsymbol{V} \coloneqq \left[\vec{\boldsymbol{u}}_1; \cdots; \vec{\boldsymbol{u}}_m\right]$ and $\boldsymbol{U} \coloneqq \left[\vec{\mathbf{v}}_1; \cdots; \vec{\mathbf{v}}_r\right]$, (iii) a publicly viewable list $\mathcal{L}$ summarizing its interactions with PIRSONA users[10], (iv) a queue $\mathcal{Q}$ of queries for which $\mathcal{T}$ has yet to update the collaborative filtering user and item profiles, and (v) a queue $\mathcal{Q}'$ of *all* queries previously submitted to $\mathcal{T}$. We stress that $\mathcal{T}$ represents and performs arithmetic operations on the components of $\boldsymbol{V}$ and $\boldsymbol{U}$ via the same fixed-point representation used by PIRSONA in the real world.

The following specifications dictate how $\mathcal{T}$ responds to requests from user i on each public interface:

1. $\mathcal{T}.\mathsf{Query}_i(j)$: If $j \notin [1..r]$, then $\mathcal{T}$ appends $(\mathsf{Query}; i, \mathtt{invalid})$ to $\mathcal{L}$ and returns $\perp$ to user i; otherwise, it appends $(\mathsf{Query}; i, \mathtt{valid})$ to $\mathcal{L}$ and (i, j) both to $\mathcal{Q}$ and to $\mathcal{Q}'$, and then it returns the requested item $\vec{\boldsymbol{D}}_j$ to user i.
2. $\mathcal{T}.\mathsf{Recommend}_i(u)$: $\mathcal{T}$ appends $(\mathsf{Recommend}; i, u)$ to $\mathcal{L}$ and then it emulates the steps of the protocol of Section 7 to render a recommendation; that is, it computes

$$cdf_i \coloneqq \left(P_1, P_2, \ldots, P_r\right),$$

using $P_k \coloneqq \sum_{j=1}^{k} p_{i,j}$ for each $k = 1, \ldots, r$, where

$$p_{i,j} \coloneqq \begin{cases} 0 & (i,j) \in \mathcal{Q}' \text{ and} \\ \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle & \text{otherwise,} \end{cases}$$

and then it sends $\min\{k \mid P_k \leq u \cdot P_r\}$ to user i.

In addition to the two public interfaces, $\mathcal{T}$ also implements the following profile-updation interface that is invoked periodically (pursuant to some policy specific to the given PIRSONA deployment):

3. $\mathcal{T}.\mathsf{Update}()$: $\mathcal{T}$ appends (Update) to $\mathcal{L}$ and then, while $\mathcal{Q}$ is nonempty, it pops (i, j) from the front of $\mathcal{Q}$ and verifies that $j \in [1..r]$; if so, it updates

$$\vec{\boldsymbol{u}}_i \leftarrow \vec{\boldsymbol{u}}_i + \vec{\mathbf{v}}_j\left(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle\right)$$

and

$$\vec{\mathbf{v}}_j \leftarrow \vec{\mathbf{v}}_j + \vec{\boldsymbol{u}}_i\left(1 - \langle \vec{\boldsymbol{u}}_i, \vec{\mathbf{v}}_j \rangle\right).$$

Once $\mathcal{Q}$ is empty, $\mathcal{T}$ uses the piecewise-linear approximation specified by coeffs (see Section 6.4) to normalize (i) every item profile $\vec{\mathbf{v}}_j$ and (ii) each user profile $\vec{\boldsymbol{u}}_i$ that was updated in the preceding while loop.

10 The summary $\mathcal{L}$ captures any cleartext metadata that PIRSONA servers automatically learn in the real-world analogs of the two interfaces.

Security theorem

Theorem 1. *For every semi-honest PPT adversary $\mathcal{A}$ controlling at-most-1 server in a real-world $(s+1)$-server PIRSONA deployment, there exists an efficient simulator $\mathcal{S}$ that, by interacting with $\mathcal{T}$, samples simulated protocol views from a distribution computationally indistinguishable from that describing real-world views of $\mathcal{A}$.*

Proof (sketch). We assume without loss of generality that $\mathcal{A}$ controls P_k for some $k \in [1..3]$, and then we sketch the strategy $\mathcal{S}$ employs to simulate each of the three interfaces that $\mathcal{T}$ exposes. Note that this setup implies that $\mathcal{A}$ holds a copy of the database $\boldsymbol{D}$ along with additive shares of $\boldsymbol{V}$; if $k = 0$ or 1, then $\mathcal{A}$ further holds additive shares of $\boldsymbol{U}$.

Meanwhile, $\mathcal{S}$ holds a copy of $\boldsymbol{D}$ and all value sent to or received from $\mathcal{T}$ by corrupted users; it uses $\mathcal{T}$'s publicly viewable list $\mathcal{L}$ to decide which interface call to simulate next—and with which inputs—as it progresses through the simulation. Specifically, $\mathcal{S}$ performs an in-order traversal through the entries in $\mathcal{L}$, invoking one of the following three sub-routines depending on the entry it encounters at each step of the traversal. It maintains queues $\hat{\mathcal{Q}}$ and $\hat{\mathcal{Q}}'$ which serve purposes analogous to the queue's $\mathcal{Q}$ and $\mathcal{Q}'$ that $\mathcal{T}$ maintains.

If the next entry is $(\mathsf{Query}; i, x)$**:** There are two cases to consider, namely when user i is corrupted versus when it is non-corrupted.

Case 1 (i is corrupted): In this case, both $\mathcal{S}$ and $\mathcal{A}$ are privy to the query index j. In the real world, $\mathcal{A}$ is also privy to the query–response pair $(\mathsf{seed}_{k'}, \mathsf{response}_{k'})$ that user i sends to and receives from $\mathsf{P}_{k'}$ for each $k' = 0, \ldots, s$, plus the record $\vec{\boldsymbol{D}}_j$ that user i ultimately reconstructs.

To simulate this, $\mathcal{S}$ uses the (genuine) Hafiz–Henry algorithms for the simulation; that is, it invokes the Hafiz–Henry query algorithm with input j to sample $(\mathsf{seed}_0, \ldots, \mathsf{seed}_s)$, and then it produces the responses $(\mathsf{response}_0, \ldots, \mathsf{response}_s)$ by evaluating the Hafiz–Henry response algorithm against $\boldsymbol{D}$ using each resulting query vector, before finally running the Hafiz–Henry reconstruction algorithm to extract $\vec{\boldsymbol{D}}_j$ from the responses.

$\mathcal{S}$ adds the triple $(i, \mathsf{dpf}^{(0)}, x)$ to $\hat{\mathcal{Q}}$ and $\hat{\mathcal{Q}}'$, where $\mathsf{dpf}^{(0)}$ is the least-significant-bit DPF seed in the query vector seed_k.

Case 2 (i is non-corrupted): In this case, neither $\mathcal{S}$ nor $\mathcal{A}$ is privy to the query index j. In the real world, $\mathcal{A}$ learns only the query vector that user i sends to P_k and the response that P_k sends back to user i.

To simulate this, $\mathcal{S}$ invokes the simulator for Boyle–Gilboa–Ishai $(2,2)$-DPF generation to sample an L-tuple seed_k of simulated $(2,2)$-DPF seeds,[11] and then it produces the response $\mathsf{response}_k$ by evaluating the Hafiz–Henry response algorithm against $\boldsymbol{D}$ using these simulated query vector.

$\mathcal{S}$ adds the triple $(i, \mathsf{dpf}^{(0)}, x)$ to $\hat{\mathcal{Q}}$ and $\hat{\mathcal{Q}}'$, where $\mathsf{dpf}^{(0)}$ is the least-significant-bit DPF seed in the query vector seed_k.

If the next entry is $(\mathsf{Recommend}; i, u)$**:** There are three main cases to consider, namely when $k = 3$, when $k = 2$, or when $k = 0$ or 1; each of these cases comprising two subcases, name when i is corrupted versus when it is non-corrupted. The "outer" cases capture differences in what $\mathcal{A}$ learns in its capacity as a party to the MPC, while the "inner" cases capture differences in what $\mathcal{A}$ learns (or does not learn) in its capacity as user. For brevity, we treat the MPC-specific part of the three "outer" cases in isolation, before separately addressing the two "inner" sub-cases.

Case 1 ($k = 3$): PIRSONA produces recommendations using a 3PC involving only P_0, P_1, and P_2; hence, there are no MPC-related steps to simulate.

Case 2 ($k = 2$): P_2 receives no input and produces no output in the MPC (i.e., it only sends Du-Atallah blinding factors and correction terms to P_0 and P_1); hence, the MPC-related parts of the simulation are trivial.

Case 3 ($k = 0$ or 1): $\mathcal{S}$ computes

$$[\vec{\mathsf{e}}]_k \leftarrow \bigoplus\nolimits_{(i,\mathsf{dpf}) \in \hat{\mathcal{Q}}'} \mathsf{EvalFull}(\mathsf{dpf}),$$

and then it passes $[\vec{\mathsf{e}}]_k$, $[\vec{\boldsymbol{u}}_i]_k$, and $[\boldsymbol{V}]_k$ to the simulator for generalized Du-Atallah multiplication-based computation of $[pmf_i]_k$ from which it computes its own simulated share of $[cdf_i]_k = \big([P_1]_k, \ldots, [P_r]_k\big)$. Finally, it repeatedly invokes the simulator for $\mathcal{F}_{\mathsf{geq}}$ to simulate comparing $u \cdot [P_r]_k$ with each entry of $[cdf_i]$, and then it outputs the inner product of the comparison results with $\langle 1, 2, \ldots, r \rangle$ to produce a recommendation share $[recomm]_k$.

For the sub-cases:

11 The pseudorandomness of leaf-layer CWs implies that leafless DPF seeds are indistinguishable from standard DPF seeds with the same domain and range; hence, no additional steps are required to simulate the least-significant-bit seed.

Sub-case 1 (i is corrupted): In this case, $\mathcal{A}$ produces no output and so there is nothing for $\mathcal{S}$ to output.

Sub-case 2 (i is non-corrupted): In this case, $\mathcal{A}$ learns both of the shares comprising $[recomm]_k$ (and, hence $recomm$ itself); thus, $\mathcal{S}$ computes $[recomm]_{1-k} = recomm - [recomm]_k$.

If the next entry is (Update)**:** While $\hat{\mathcal{Q}}$ is non-empty, $\mathcal{S}$ pops $(i, \mathsf{dpf}^{(0)}, x)$ from the front of $\hat{\mathcal{Q}}$. If $x = \texttt{invalid}$, then $\mathcal{S}$ simulates a rejecting run of the sanity check protocol and stops. Otherwise, if $x = \texttt{valid}$, then $\mathcal{S}$ simulates an accepting run of the sanity check protocol, invokes the simulator for the $\mathcal{F}_{\text{MUX}}$, invokes the simulator for computing the Du–Atallah-based gradient descent adaptation values, and then invokes the simulator for $\mathcal{F}_{\mathsf{demux}}$ to update item-profile shares. Once $\hat{\mathcal{Q}}'$ is empty, $\mathcal{S}$ also invokes the simulator for precision reduction and normalization, as mandated by the actual MPC protocols.

□