PProx: efficient privacy for recommendation-as-a-service

Guillaume Rosinosky, Simon da Silva, Sonia Ben Mokhtar, Daniel Négru, Laurent Réveillère, Etienne Rivière

PProx: Efficient Privacy for Recommendation-as-a-Service

Guillaume Rosinosky
ICTEAM, UCLouvain, Belgium

Simon Da Silva
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Sonia Ben Mokhtar
INSA Lyon, LIRIS, CNRS, France

Daniel Négru
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Laurent Réveillère
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Etienne Rivière
ICTEAM, UCLouvain, Belgium

Abstract

We present PProx, a system preventing recommendation-as-a-service (RaaS) providers from accessing sensitive data about the users of applications leveraging their services. PProx does not impact recommendations accuracy, is compatible with arbitrary recommendation algorithms, and has minimal deployment requirements. Its design combines two proxying layers directly running inside SGX enclaves at the RaaS provider side. These layers transparently pseudonymize users and items and hide links between the two, and PProx privacy guarantees are robust even to the corruption of one of these enclaves. We integrated PProx with the Harness recommendation engine and evaluated it on a 27-node cluster. Our results indicate its ability to withstand a high number of requests with low end-to-end latency, horizontally scaling up to match increasing workloads of recommendations.

CCS Concepts • **Security and privacy**;

Keywords privacy, security, recommendation, TEE

ACM Reference Format:
Guillaume Rosinosky, Simon Da Silva, Sonia Ben Mokhtar, Daniel Négru, Laurent Réveillère, and Etienne Rivière. 2021. PProx: Efficient Privacy for Recommendation-as-a-Service. In *Middleware '21: Middleware '21: 22nd International Middleware Conference, December 6–10, 2021, Québec City, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/TBD

1 Introduction

Recommender systems [16] complement traditional navigation on websites and applications, improving user experience [41], enabling personalized services [40], and eventually increasing service providers' revenue [36]. Recommendations can be new directions, items, or media. These are computed based on a user's past interactions and feedback (*e.g.*, item likes, navigation clicks) combined with the interactions of other users with similar interests. Recommendations are used by major Web services providers but can also benefit smaller players (*e.g.*, discussion forums or online stores).

As configuring and operating an efficient and scalable recommendation service is far from trivial, several companies offer *Recommendation-as-a-Service* (RaaS). Examples include Darwin & Goliath [14], Mediego [1], Plista [2], or Recombee [3]. In the RaaS service model, illustrated in Figure 1, application providers delegate the collection

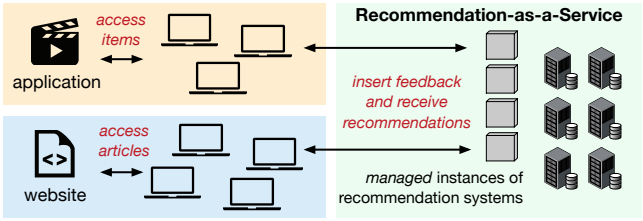

Figure 1. Principle of Recommendation-as-a-Service.

of interaction information (*feedback*) from their users, the construction of models based on this feedback, and the generation of personalized item recommendations from their catalog.

The major downside of using recommendation systems is their impact on users' privacy. Computing recommendation requires, indeed, the collection of massive amounts of sensitive data, which raises legitimate concerns amongst users [74]. Access histories and feedbacks may reveal personal traits or interests, *e.g.*, based on access to different topics in an online forum or specific movies in a review platform. An adversary observing in the clear interactions with the recommender system or its database may infer private information about users such as their faith, sexual preferences, or health condition [21], or simply use profiling information for non-legitimate purposes such as unsolicited advertising.

Privacy risks of recommender systems can be, unfortunately, amplified by the use of RaaS. Users of applications that outsource the generation of recommendations to a RaaS provider now have to trust this provider, an additional third-party, for receiving and storing sensitive data. It is desirable, therefore, that RaaS providers be able to provide privacy-by-design guarantees to the users of their client applications.

The research community proposed various solutions to deal with the privacy concerns of recommender systems. These solutions can be classified in three categories: (i) those based on cryptography [46, 77] where computations are performed over fully-encrypted data; (ii) differentially private solutions [39, 56, 70, 71] that add noise for disallowing the re-identification of a specific user and their data and (iii) decentralized/federated solutions [17, 22, 26, 30, 33, 61] where users keep their preferences locally and compute similarity with other users in a decentralized manner. These solutions present drawbacks such as performance issues for cryptography-based solutions or accuracy issues for differentially private solutions (due to the addition of noise) and peer-to-peer solutions (due to the partial knowledge users have on the overall system). Perhaps more importantly, none is well adapted to the RaaS service model, and that for two key reasons:

- They all target a *specific* type of recommendation algorithm (*e.g.*, using matrix factorization [33, 71] or collaborative filtering [22,

72]). This goes against the need for RaaS providers to support a variety of such algorithms [16, 20];
- They require to install complex code and to maintain specific or even sensitive information at the user-side, at odds with the "turn-key" service model of RaaS. Solutions based on encrypted processing [46, 77] require to provision secret keys to the user side with associated risks of leakage and additional complexity of large-scale private key management. Differentially private solutions implemented at the client side [56, 70] require to provide clients with models of the data domain to enable adaptive noise addition.

Contributions. We present PProx, an efficient and easily-deployable solution for privacy-by-design in RaaS.

PProx introduces a *privacy-preserving proxy service*, standing between users and an unmodified legacy recommendation system (LRS). This proxy service intercepts feedback insertions and requests for recommendations. It pseudonymizes on the fly users' and items' identifiers and hides links between the two. This guarantees *unlinkability* between clients and both the items they access or receive as recommendations. The deployment of PProx does not require to provision private keys or models to the user side. PProx does not modify in any way the results returned by the LRS (*e.g.*, by adding noise) and its use is totally transparent for the users.

PProx leverages the support in modern cloud infrastructure for trusted execution environments (TEE), allowing to run secure enclaves on untrusted hardware. TEEs include ARM's TrustZone, AMD's MET, and Intel SGX [29] that we use in this paper. Applications client of RaaS provision these enclaves with secrets allowing the pseudonymization of their users and of items of their catalog, and these secrets are not accessible to the RaaS provider. In contrast with earlier work using SGX to protect users' privacy from online service providers [50, 51, 60] the design of PProx acknowledges the possible vulnerability of SGX enclaves to side-channel attacks [18, 24, 43, 58, 68, 75, 78]. These attacks are, however, costly to implement, with completion times in the tens of minutes and significant degradation of the attacked enclave performance [63]. PProx makes, therefore, the assumption that while the adversary may break *one* such enclave, it may not break *multiple* enclaves synchronously before a breach detection mechanism triggers appropriate countermeasures [25, 45, 64].

PProx prevents a corrupted RaaS provider from breaking unlinkability properties between users and items, by using a data partitioning principle. Information necessary to link a user to a specific item or recommendation is *split* between two layers running in *different* SGX enclaves. In addition, PProx provides protection against traffic correlation attacks that would be performed by a malicious provider observing the sequence of network exchanges between users, enclaves, and the recommendation engine. Protection against such timing attacks is achieved through request and response *shuffling*, hiding the correlation between flows while respecting tight bounds on additional service latency.

PProx is integrated with the *Universal Recommender* [7] module of Harness [6], an open-source machine learning platform. Harness is representative of an LRS used by a RaaS provider: It supports high-throughput and low-delay operations and scales horizontally to serve growing user bases. PProx is also able to similarly scale horizontally to handle varying load while minimizing the performance impact of privacy preservation.

Our evaluation over a 27-node/54-core Kubernetes cluster of Intel SGX-capable NUC servers, and using a real-world workload, shows that PProx is able to efficiently protect privacy while respecting strict end-to-end latency objectives, and scale up to handle increasing workloads in unison with the scaling of the LRS. A single instance of PProx can handle 250 requests per second using 4 cores, and scale up to 1.000 requests per second using 8 proxy instances, matching the capacity of a 32-core deployment of Harness.

Outline. We detail our system and adversary model in Section 2, and give a high-level overview of PProx in Section 3. We present the construction of the proxy service in details in Section 4, and its implementation in Section 5. We discuss the security of PProx in Section 6, and overview its integration with Harness in Section 7. We evaluate the resulting system in Section 8, present related work in Section 9, and conclude in Section 10.

2 System model and objectives

We start by defining our system model, our assumptions, our security objectives, and the power of the adversary.

2.1 System model

Figure 2 illustrates the constituents of the system. Users interact with a website or application offering access to items, *e.g.*, books, news articles, or movies (①). This service outsources the management of a recommendation feature embedded in its front-end to a Recommender-as-a-Service (RaaS) solution running in the cloud (②).

The RaaS runs a legacy recommendation system (LRS) for the application (③), accessed via a simple REST API. A $\texttt{post}(u, i[, p])$ request allows user u to send feedback to the recommendation engine about access to item i with an optional payload p, if required by the recommendation algorithm. For instance, a movies recommender may leverage *ratings* by the user, while a recommender for items in an online store may only require identifiers. A $\texttt{get}(u)$ request returns a collection of n items $(i_1, \ldots, i_n)$ recommended to user u.

PProx introduces an additional component, the privacy-preserving proxy service (④), lying between the clients and the LRS. It runs as part of the RaaS in the same cloud as the LRS to avoid indirections through multiple data centers and the resulting impact on latency.

A thin user-side library is easily embeddable in the application or web front-end as static web code, and offers the exact same REST API as the LRS. This library intercepts, encrypts and forwards clients' API calls to the proxy service. In the case of get calls, it returns the list of recommendations (⑤).

2.2 Trust and operational assumptions

The user-side library is considered trusted for the processing and handling of personal data, *i.e.*, its code is public, in an interpreted language, and it can be audited by external parties. We also trust the user: protecting against a compromised browser or application is orthogonal to this work.

The RaaS provider runs its service in the cloud, and its infrastructure may be subject to attacks performed by external or internal malicious operators. The RaaS providers wishes, therefore, to offer *privacy-by-design* guarantees to its clients, and be robust to the occurrence of such attacks and resulting data leaks. We assume the availability in the cloud used by the RaaS provider of trusted

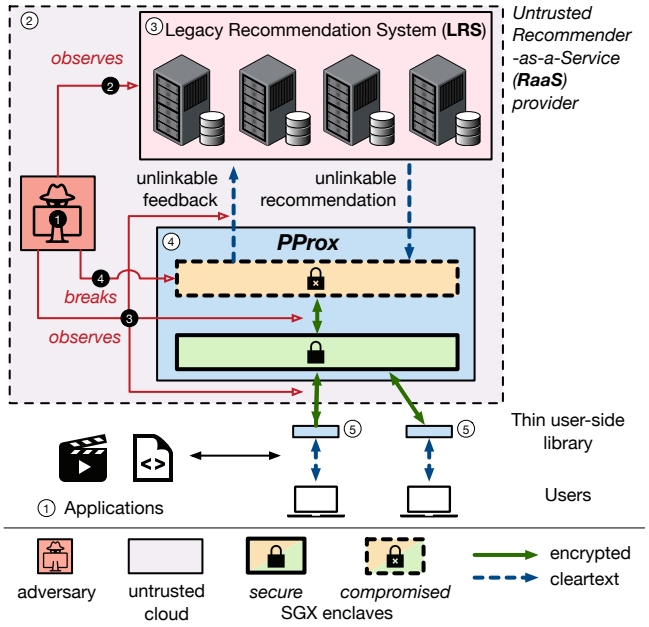

Figure 2. PProx system constituents (①-⑤ in §2.1) and adversary model (❶-❹ in §2.3).

execution environments (TEE). Our implementation uses specifically Intel SGX [29]. We trust Intel for the certification of genuine SGX-enabled CPUs, and we assume that the code running inside enclaves is properly attested before being provided with secrets.

An LRS is typically built over data processing frameworks, *e.g.*, Apache Spark, and databases, *e.g.*, MongoDB, preventing from using source-based application partitioning techniques such as Glamdring [54]. As the data that the LRS uses is almost entirely of sensitive nature, the trusted computing base is potentially very large, preventing the use of full-application containment, *e.g.*, using SCONE [9]. It is not desirable, under these conditions, to run the LRS itself inside SGX enclaves and we reserve their use for the proxy service, which we design to take into account TEEs' constraints.

2.3 Privacy objectives and adversary model

The goal of PProx is to preserve *User-Interest unlinkability*. It should be impossible for an adversary to relate a specific user (as identified by their identifier or any unique characteristic, *e.g.*, their IP address or geographical location) to an access to an item or of their possible interests as reflected by received recommendations. More formally, it should be impossible for an adversary to (1) learn that a user u called $\texttt{post}(u, i[, p])$ for item i and (2) that a user u received a recommendation for an item i following a $\texttt{get}(u)$ call.

We consider a powerful adversary (Figure 2, ❶). It wishes to break the unlinkability property by observing all components of the RaaS backend. It does not, however, interfere with the functionality of the system: It does not attempt to manipulate or bias the recommendations returned by the LRS, and it does not block or delay the access to the service for specific users or specific applications. More specifically, the adversary interest is in stealing information about users for its own profit, but not in triggering alerts for failed service-level agreements, that could reveal its operations.

The adversary may be an insider to the RaaS organization (e.g., a corrupted operator) or an outsider exploiting a fault in the RaaS software stack. It can bypass traditional security measures such as system-level access control or the use of secure connections (TLS/SSL) to and from the users [79]. This adversary can, as a result, see all API calls to the LRS *in the clear*, and can access any data manipulated by the LRS when computing recommendations (❷). In addition, it may monitor network flows between the nodes forming this infrastructure, both with the outside world and internally (❸), and correlate in time its observations.

Our design takes into account the possibility, highlighted by recent work, of time-based or cache-based side-channel attacks on SGX [18, 24, 43, 58, 68, 75, 78], allowing an adversary to access the secrets that were provisioned to an enclave. This contrasts with previous designs that consider enclaves as inviolable [9, 50, 51, 60]. Implementing such attacks is, however, costly and time consuming. Reported attacks indicate completion times in tens of minutes while making enclave performance drop significantly [63]. Delete: ►resulting in effective attack times when performance should not drop below a detection threshold significantly longer in practice.◄ Mechanisms such as Cloak [45], Déjà Vu [25] or Varys [64] allow, furthermore, to detect the occurrence of such attacks and to respond appropriately[1]. If the adversary wishes to perform an attack without significantly impairing performance (thereby avoiding detection) the effective time can be, in practice, much longer. Delete: ►(*e.g.*, by shutting down the system and restart it after a security audit and using new secrets).◄ Our model includes, therefore, the possibility for the adversary to compromise and break into *a single enclave* at a time, on any server. For instance, we illustrate in Figure 2 that an enclave of the proxy service has been compromised and its secrets leaked to the adversary (❹).

3 PProx in a nutshell

In addition to security features, PProx targets two key systems objectives: performance and ease of deployment.

PProx must sustain the request load that is supported by the LRS, with a potentially high throughput of user requests for both insertions of feedback and collections of recommendations. The interaction of users with the RaaS solution must, in addition, happen with small delays (typically, at most a few hundred milliseconds as required for the user to consider the system interactive [8]). It is not desirable, therefore, to rely on anonymity services such as AnonyFlow [57] or Tor [32], for their lack of reliability and important impact on latency, and PProx security enforcement must not impose latencies that would violate the RaaS service-level objectives.

Ease of deployment requires that the integration with the website or application using the RaaS service only relies on static code and globally known information. It must not require any intervention of users and should not suppose that client-specific or session-specific state is kept by the clients, other than their identifier with the website or application.

A two-layer privacy-preserving proxy service. At the core of PProx is a proxy service that guarantees that (1) the LRS only sees pseudonymous information, for both user identifiers and item

[1] The appropriate response must take into account the fact that secrets provisioned to the corrupted enclave are now in the hands of the adversary. Available options include dropping the database content and re-starting the system with new secrets, downloading the LRS state for local re-encryption before re-uploading it and provisioning fresh enclaves and the user-side library with new secrets, or employing an LRS-specific proxy re-encryption technique using (or not) an enclave [28, 65].

identifiers[2] and (2) that it is impossible to relate a call from some user to a call sent to the LRS (and similarly for responses from the LRS to the user).

We start by observing that mapping a user identifier to a pseudonym in a *single* SGX enclave acting as a proxy and forwarding the pseudonymized request to the LRS is not sufficient under our adversary model. The adversary may, indeed, compromise this single enclave and learn the direct associations between user identifiers and item identifiers. PProx uses instead a two-layer proxy service, with the two layers running in distinct SGX enclaves on different servers. The foundational principle of this design is that no enclave is provisioned with *all* the secrets necessary to an adversary to break unlinkability:

- The first layer, the **User Anonymizer** (UA) is responsible for hiding the identity of the user by replacing it with a pseudonymous identity. It is able to see the IP address and the identifier of the user but it is not able to see the identifiers of the items sent by or returned to this user.
- The second layer, the **Item Anonymizer** (IA) is the one that directly interacts with the LRS. It is the only layer able to access items identifiers in the clear, but it is not able to access user identifiers or IP addresses. It can map actual item identifiers as used in the application's catalog to pseudonymous identifiers used by the LRS, and reversely.

Protection from network observation attacks. Attacking an enclave is not the only way the adversary may attempt to break unlinkability. As we consider it may observe communications with the LRS in the clear, the adversary could monitor the series of interactions that occur between the user and the UA layer, between the UA and the IA layers, and finally between the IA layer and the LRS. It could, eventually, link a specific IP address and both the pseudonymous user identifier and items used for the actual request.

PProx protects against such attacks by *shuffling* communication for anonymizing requests of multiple users between the UA and IA layers. We make the assumption that the system is under a flow of requests of sufficiently high volume (*e.g.*, 50 per second in our evaluation). Redirections only happen after a configurable number of requests have been buffered, and these requests are sent in a randomized order. The adversary cannot, as a result, determine precisely which final request sent to the LRS in the clear correspond to a specific incoming request to the UA. The same applies to responses sent back from the LRS to the user side. The use of buffering introduces a queuing delay to every request but this delay does not prevent from achieving overall latencies of at most a few hundred milliseconds, as per our objective.

4 PProx protocol design

We detail in this section the PProx protocol, from the interception of requests at the application side to their handling by the proxy service, and their final processing by the LRS. We use the notations listed in Table 1. We focus on the protocol in this section and discuss its implementation and scaling in Section 5. We analyze its security in Section 6.

[2]We note that for companies operating in the EU market, the storage of pseudonymous information for user identifiers can help comply with the requirements of the EU's General Data Protection Regulation [48].

u	User identifier
$i_1, ..., i_n$	Item identifiers
UA	User Anonymizer, 1st layer of the proxy service
IA	Item Anonymizer, 2nd layer of the proxy service
$\mathsf{pk}_{\mathrm{UA}}$	Public key of User Anonymizer layer
$\mathsf{sk}_{\mathrm{UA}}$	Private key of User Anonymizer layer
$\mathsf{pk}_{\mathrm{IA}}$	Public key of Item Anonymizer layer
$\mathsf{sk}_{\mathrm{IA}}$	Private key of Item Anonymizer layer
k_{UA}	Permanent symmetric key of User Anonymizer layer
k_{IA}	Permanent symmetric key of Item Anonymizer layer
k_u	Temporary symmetric key generated by user u
$\mathbf{enc}(x, \{\mathsf{p}\|\mathsf{s}\}\mathsf{k})$	asymmetric encryption of x using public/private key k
$\mathbf{det\_enc}(x, \mathsf{k})$	deterministic symmetric encryption of x using key k
S	Size of the shuffling buffer

Table 1. Notations.

4.1 Provision and use of cryptographic material

The UA layer is provisioned with private key $\mathsf{sk}_{\mathrm{UA}}$ and a permanent symmetric key k_{UA}. The IA layer is similarly provisioned with $\mathsf{sk}_{\mathrm{IA}}$ and k_{IA}. The enclaves implementing the two layers are attested upon their bootstrap before being provisioned with these keys by the RaaS client application. The two types of keys serve complementary purposes:

- Public/private key pairs enable the user-side library to encrypt information for exclusive visibility by one of the two layers. For instance, the user identifier should only be visible in the clear by the UA layer. The user-side library intercepts the cleartext request and transforms u into $\mathbf{enc}(u, \mathsf{pk}_{\mathrm{UA}})$ so that only the UA layer may recover u from the ciphertext using $\mathsf{sk}_{\mathrm{UA}}$. However, this same ciphertext cannot be used as the pseudonym of u with the LRS, as it is the result of randomized encryption: Two encryptions of the same u yield two different ciphertexts and do not allow linking to a single pseudonymous user profile.
- The permanent symmetric keys k_{UA} and k_{IA} are used for *deterministic* encryption of the users' and items' identifiers, enabling their *pseudonymization.* The UA layer, accessing some user identifier u in the clear, can encrypt it such that the resulting ciphertext is the same as with another encryption of the same input. The same applies to the IA layer, which must be able to deterministically encrypt an item identifier i_x it sees in the clear. While deterministic encryption has lower security (*e.g.*, less resilience against know-plaintext attacks than probabilistic encryption), it is necessary to allow the LRS to recognize two encrypted user or item identifiers as being the same entity. We enable deterministic symmetric encryption by using the AES 256 CTR block cipher with a constant initialization vector.

In addition to these permanent keys provisioned to the two layers, PProx uses a distinct *temporary* symmetric key for each `get` request generated by the user-side library, and allowing to protect the returned collection of recommendations. A temporary symmetric key for a user u's `get` request is denoted as k_u. We note that, unlike for using k_{UA} and k_{IA} in symmetric encryption for pseudonymization, the encryption of returned results uses regular randomized encryption, *i.e.*, AES with a random initialization vector.

4.2 Transparent REST calls redirection

The LRS offers a REST API and the user-side library intercepts unmodified calls to this API. The user-side library and the two proxy service layers modify the headers, to implement redirections, and payloads, to enable encryption. Each layer maintains a table

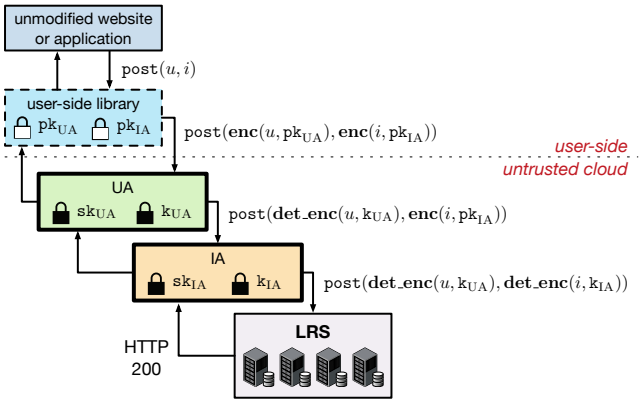

Figure 3. Lifecycle of a post request (insert feedback).

T storing the association between an inbound socket I (from the user-side library or from another proxy) and an outbound socket O (to another proxy or to the LRS). Responses from the LRS are forwarded backward using the same path as for the incoming request. The response is finally provided to the application by the user-side library as if it was returned by the LRS itself. We discuss the implementation and performance of redirections and the maintenance of T in Section 5 and focus in the following on the end-to-end lifecycle of post and get operations.

4.2.1 Insertion of feedback (post requests)

A post request inserts feedback about the access to an item i by a user u. There is no specific return value for this call, other than the HTTP header's success or error code from the REST API. The end-to-end lifecycle of a post call is illustrated in Figure 3 and detailed below.

The user-side library first transforms the call $\texttt{post}(u, i)$ by encrypting the two arguments, yielding a new call

$$\texttt{post}(\textbf{enc}(u, \mathsf{pk}_{\mathrm{UA}}), \textbf{enc}(i, \mathsf{pk}_{\mathrm{IA}}))$$

that is sent to the UA layer. This layer decrypts u using private key $\mathsf{pk}_{\mathrm{UA}}$. It pseudonymizes plaintext u by deterministically encrypting it using k_{UA}. The resulting call

$$\texttt{post}(\textbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}}), \textbf{enc}(i, \mathsf{pk}_{\mathrm{IA}}))$$

is forwarded to the IA layer. This layer can decrypt i using private key $\mathsf{sk}_{\mathrm{IA}}$, and similarly pseudonymize the plaintext item identifier using key k_{IA}. The call containing the unlinkable information is finally forwarded to the LRS as

$$\texttt{post}(\textbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}}), \textbf{det\_enc}(i, \mathsf{k}_{\mathrm{IA}}))$$

and the response traverses back the two layers.

4.2.2 Collection of recommendations (get requests)

A get request returns a set of recommended items $(i_1, \ldots, i_n)$ tailored for a specific user u. The LRS maintains information about previous feedbacks in its database using pseudonymous item identifiers, which must be decrypted by the IA layer. This list must not be visible by the UA layer that can access the user identifier in the clear. The lifecycle of a get request is illustrated in Figure 4.

When intercepting a get request, the user-side library generates a temporary key k_u and encrypts it using $\mathsf{pk}_{\mathrm{IA}}$. This key k_u will be used by the IA layer to encrypt the list of recommendations and hide it from the UA layer, and is therefore encrypted with the IA

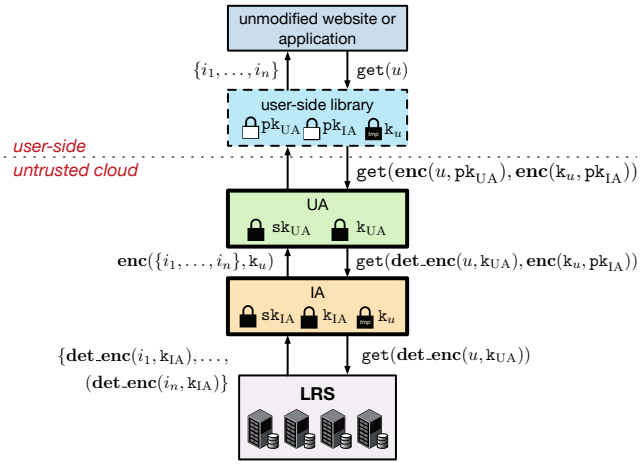

Figure 4. Lifecycle of a get req. (collect recommendations).

layer's public key $\mathsf{pk}_{\mathrm{IA}}$. The user identifier is encrypted, as for a post request, using the UA layer's public key, yielding the call

$$\texttt{get}(\textbf{enc}(u, \mathsf{pk}_{\mathrm{UA}}), \textbf{enc}(\mathsf{k}_u, \mathsf{pk}_{\mathrm{IA}})).$$

The UA layer when receiving this call pseudonymizes the user identifier as for a post request and sends the call

$$\texttt{get}(\textbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}}), \textbf{enc}(\mathsf{k}_u, \mathsf{pk}_{\mathrm{IA}}))$$

to the IA layer. The IA layer then sends $\texttt{get}(\textbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}}))$ to the LRS. The returned list

$$\{\textbf{det\_enc}(i_1, \mathsf{k}_{\mathrm{IA}}), \ldots, (\textbf{det\_enc}(i_n, \mathsf{k}_{\mathrm{IA}})\}$$

contains pseudonymized item identifiers. These identifiers are decrypted to plaintext item identifiers used by the application using k_{IA}. The recommendations list is then re-encrypted to hide it from the UA layer using the user key k_u, yielding $\textbf{enc}(\{i_1, \ldots, i_n\}, \mathsf{k}_u)$. The call traverses back the layers until the user-side library, which decrypts the list of recommended item identifiers using k_u and returns it in the clear, and transparently, to the application.

4.3 Requests and response shuffling

The pseudonymization of user and item identifiers is necessary, but not sufficient, to enable the property of User-Interest unlinkability. The adversary can observe, indeed, all network communications between the nodes hosting the system components: between the user and the UA layer, between the UA and IA layers, and between the IA layer and the LRS. By correlating these observations in time, it can relate an input request (from the user to the UA layer) to a pseudonymized request from the IA layer to the LRS. If, in addition, the adversary was able to compromise the IA layer, it could learn the association between this user IP address and the item identifiers in the clear.

We first ensure that the adversary cannot distinguish between encrypted messages exchanged between the user-side library and the UA layer, and between the UA and IA layers. The size of all encrypted messages is constant, by using fixed-size user and item identifiers, and padding when necessary. The list of items returned by the LRS has a maximal size (20 in our implementation) and we use padding to fill in missing entries. The pseudo-items used for padding are automatically discarded by the user-side library.

We implement request *shuffling* to protect from network inference attacks, as illustrated in Figure 5. Shuffling hides the direct

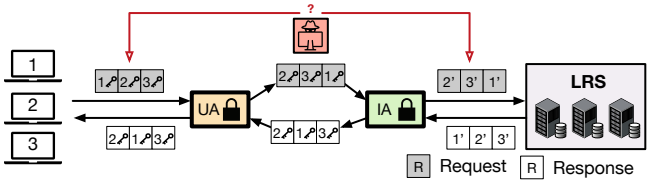

Figure 5. Shuffling disallows the adversary from determining which of S (here $S = 3$) incoming requests to the UA layer corresponds to a specific request sent to the LRS. The same strategy is applied to responses from the LRS.

mapping between an input request from the user to the UA layer and the redirection of this request to the IA layer. Similarly, it hides the correspondence between a response from the LRS to the IA layer, and the corresponding redirection to the UA layer holding the user's connection. Both types of mappings are, in fact, made indistinguishable from $S-1$ other requests, S being the size of a *shuffling buffer* used by the corresponding proxy layer (UA for requests, IA for responses). Incoming requests are buffered until S requests are received, or until a timer expires, and then sent in random order to the next stage. The size of the buffer S is a compromise between the additional latency imposed on requests and responses and the power of the attacker. This bears similarities with the principle of k-anonymity in privacy-preserving databases [52, 73][3].

5 Implementation

We focus in this section on the implementation of the privacy-preserving proxy service running in SGX enclaves. The implementation of the user-side library in Javascript and its integration into a webpage, is straightforward; we do not detail it due to space restrictions.

The proxy service must be able to support a large number of concurrent requests. This is particularly challenging as (1) part of the proxy logics resides in SGX enclaves and (2) this logic must perform CPU-intensive cryptographic computations. In addition to a high level of concurrency, the proxy design must target *fairness* in the processing of requests, in order to control service time tail latency. This requires ensuring that no request gets delayed arbitrarily more than the delay that shuffling already introduces. Scheduling the processing of requests should not introduce, on the other hand, significant synchronization overheads.

The proxy service is implemented in C++ using the Intel SGX SDK [4]. Cryptographic operations use Intel's OpenSSL SGX port [5], using RSA for asymmetric encryption and AES-CTR mode for symmetric encryption. We use a constant initialization vector (IV) for deterministic encryption (user and item pseudonymization). For regular encryption of data to and from the client, we use a randomly-generated IV that is stored temporarily in the enclave memory. Data from/to the client and from/to the LRS is structured in JSON, and the encrypted content is handled and stored in the `base64` format. The implementation is split in two parts, server and data processing, which we detail below.

[3]We note that, in contrast with some past work using SGX enclaves to protect the privacy of online services such as web search [60], `PProx` does not attempt to hide legitimate user requests amongst artificial (fake) traffic. The principal reason is that, unlike for read-only web search queries, insertion of feedback (`put` queries) in the LRS mutate its state, and would bias future recommendations. While fake `get` queries could leverage the replay of previous user requests, they would be difficult to make undistinguishable by the attacker without using costly re-encryption in the UA layer.

Server The *server* runs outside of SGX enclaves and is identical for the UA and IA layers. It (i) handles connection requests and schedules their processing, implementing shuffling, and (ii) is in charge of receiving and sending packets. The server is the only component that performs system calls with the local OS: data processing enclaves only process data in memory that has been prepared by the server.

We adopt an event-driven approach to the scheduling and handling of incoming requests. The server runs as a single thread listening to incoming connection requests notification using the `epoll()` data structure and associated system calls of the Linux kernel. Incoming connections' file descriptors are pushed into a queue, to be consumed *in order*[4] by the pool of data processing threads. We use a lock-free, scalable concurrent queue implementation by Desrochers [31].

The server thread maintains table T, the routing table for pending requests, as a map from outbound file descriptors to inbound file descriptors (sockets). When the `epoll()` call raises an event for a file descriptor f, the server thread can lookup T to establish the corresponding return path.

Table T is also used for implementing request shuffling. When the number of elements in T reaches S or when the timer expires, the server enqueues all pending requests in a randomized order into the shared concurrent queue. Note that the size of T should be larger than S in order to avoid dropping incoming requests between the reaching of the threshold and the processing of the requests. We stress that the server only processes encrypted content without the possibility of accessing it in the clear: clients' identities, keys, IVs, and data are stored inside the enclave memory.

Data processing The *data processing* part of the proxy is supported by a pool of threads running in the SGX enclave[5]. Each data processing thread dequeues work from the tail of the shared concurrent queue. For each processed packet, a thread (i) parses it (HTTP headers and JSON payloads); (ii) performs cryptographic operations as detailed in Section 4 and (iii) forges a new packet to forward to the other proxy layer, to the LRS, or back to the client. We implemented a lightweight JSON parser inside the enclave, able to retrieve and/or update JSON fields in place and with minimal copy overhead. An in-memory key-value store in the EPC (Enclave Page Cache) holds the information necessary for handling requests responses on their way back from the LRS.

Horizontal scaling `PProx` is horizontally scalable similarly to the underlying LRS, *i.e.*, it can support increased requests loads by using more enclave instances in each proxy layer. All enclaves from the same layer are provisioned with the same secrets, but they do not need to share a common mutable state. New enclaves are attested upon their bootstrap before being provisioned with the corresponding keys.

Incoming requests from the clients are balanced to any of the enclaves in the UA layer. The following request from the UA to

[4]The order of notifications across several `epoll_wait()` system calls follows the real-time order of requests reception, except for requests received between calls that may be ordered arbitrarily. The number of such requests when the system has not reached saturation is limited and the processing of requests is, in practice, very close to the order of their reception.

[5]We use a thread pool size equal to the number of cores (2) in our evaluation, but deployments on a large multicore CPU could use one less thread in the pool than the number of cores and pin the server thread to the remaining core to reduce scheduling overheads.

the IA layer is also balanced to any of the enclaves of the latter. Responses follow the reverse path to the client.

Shuffling relies on the assumption that a sufficiently high amount of traffic is available for *each* enclave, in order to fill in the shuffling buffer before the timer expires. The two proxy layers need, therefore, to elastically scale up and down based on observed request load, dynamically implementing a compromise between throughput and latency.

Using multiple enclaves for each proxy layer does not lower security: an adversary able to break into one enclave will compromise one, but never both, of the layers, as was the case with a single enclave per layer.

6 Security analysis

We present in this section the security analysis of PProx. We first present an informal proof of the User-Interest unlinkability property (§6.1), then analyze the impact of shuffling (§6.2), and finally discuss limitations (§6.3).

6.1 User-Interest Unlinkability

To break the unlinkability between a user u and an item i, the adversary must either (1) leak information from the $\texttt{post}(u, i[, p])$ message sent by u; (2) get access to items recommended by the LRS in response to a $\texttt{get}(u)$ message, in which the item i appears or (3) de-anonymize the database of the LRS. We consider the adversary defined in Section 2. This adversary can observe network flows, read all data stored by the LRS and break into *one* of the enclaves (*i.e.*, obtain secrets from *either* a UA or an IA enclave). We proceed in steps and consider the two layers of PProx separately.

Case 1: the adversary breaks a UA enclave. The adversary gains access to the following secrets: the private key $\mathsf{sk}_{\mathrm{UA}}$ used to decrypt the user identifier u contained in a transformed $\texttt{post}(u, i[, p])$ message; and the permanent key k_{UA} used to encrypt the same user identifier u toward its storage by the LRS in pseudonymous form $\mathbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}})$. We consider in the following these three (not mutually exclusive) cases: (a) the adversary intercepts the transformed $\texttt{post}(u, i[, p])$ message at a UA enclave; (b) the adversary intercepts the response to the $\texttt{get}(u)$ message containing i as a recommended item; (c) the adversary gets access to the content of the LRS database.

Case 1.(a): *the adversary intercepts a* post *request at a broken UA enclave.* ▹ Call $\texttt{post}(u, i[, p])$ has been transformed at the user side to $\texttt{post}(\mathbf{enc}(u, \mathsf{pk}_{\mathrm{UA}}), \mathbf{enc}(i, \mathsf{pk}_{\mathrm{IA}}))$. The adversary intercepts this message and knows the origin of the request. It can link the IP address to u by decrypting $\mathbf{enc}(u, \mathsf{pk}_{\mathrm{UA}})$ using the stolen secret $\mathsf{sk}_{\mathrm{UA}}$. By accessing the LRS database, it may link u with $\mathbf{det\_enc}(i, \mathsf{k}_{\mathrm{IA}})$ as it knows k_{UA} and can thus decrypt $\mathbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}})$. However, as long as it does not steal IA layer's secrets, the adversary cannot decrypt $\mathbf{det\_enc}(i, \mathsf{k}_{\mathrm{IA}})$ and cannot, therefore, link u and i.

Case 1.(b): *the adversary intercepts the response to a* get *request at a broken UA enclave.* ▹ The adversary accesses a list of encrypted item identifiers containing $\mathbf{enc}(\{i\}, \mathsf{k}_u)$. It also knows the final destination, *i.e.*, the IP address of user u. However, it is not able to decrypt item identifiers as it does not have access to k_u, only available at the client and to the IA layer. Linking u and i would require, again, to get secrets from IA enclaves at the same time as from UA enclaves.

Case 1.(c): *the adversary breaks a UA enclave and also gets access to the content of the LRS database.* ▹ In this case, the adversary can de-pseudonymize user identifiers using k_{UA}, but it is not able to de-pseudonymize items as obtaining k_{IA} would require breaking into a second enclave, in the IA layer.

Case 2: the adversary breaks an IA enclave. Breaking an IA enclave allows the adversary to gain access to the following secrets: the private key $\mathsf{sk}_{\mathrm{IA}}$ used to decrypt an item identifier i contained in a transformed $\texttt{post}(u, i[, p])$ message; and the permanent key k_{IA} used to pseudonymize this item identifier i as $\mathbf{det\_enc}(i, \mathsf{k}_{\mathrm{IA}})$ for use by the LRS. As for Case 1, we consider the following three (not mutually exclusive) cases: (a) the adversary intercepts the transformed $\texttt{post}(u, i[, p])$ message at an IA enclave; (b) it intercepts the response to a get request, containing i as a recommended item; (c) it gets access to the LRS database.

Case 2.(a): *the adversary intercepts a* post *message at the broken IA enclave.* ▹ The message available to the IA layer is the result of transformations by the user-side library and by the UA layer, *i.e.*, $\texttt{post}(\mathbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}}), \mathbf{enc}(i, \mathsf{pk}_{\mathrm{IA}}))$. The adversary can decrypt $\mathbf{enc}(i, \mathsf{pk}_{\mathrm{IA}})$ using the leaked secret $\mathsf{sk}_{\mathrm{IA}}$ to obtain i. However, it cannot know the origin of the request thanks to the shuffling of messages performed by the UA layer. By observing the LRS, the adversary can further link i with $\mathbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}})$, having access to permanent key k_{IA}. However, as long as it does not simultaneously break one of the UA enclaves, the adversary cannot decrypt $\mathbf{det\_enc}(u, \mathsf{k}_{\mathrm{UA}})$ and cannot, therefore, link u and i.

Case 2.(b): *the adversary intercepts the response to a* get *request at the broken IA enclave.* ▹ The adversary accesses a list of encrypted item identifiers containing $\mathbf{det\_enc}(i, \mathsf{k}_{\mathrm{IA}})$. It can, therefore, decrypt i using the leaked secret k_{IA}. However, thanks to message shuffling, the adversary is not able to learn for which user (IP address) the response is, making it unable to link u and i.

Case 2.(c): *the adversary breaks an IA enclave and accesses the content of the LRS database.* ▹ The adversary does not have access to the permanent key k_{UA}, held by UA enclaves. It cannot decrypt pseudonymous user identifiers in the LRS database, preserving unlinkability between u and i.

In summary, even if it breaks one of the UA enclaves or one of the IA enclaves, an adversary cannot break user-interest unlinkability despite actively observing network activity and accessing the content of the LRS database.

6.2 Impact of Shuffling

We analyze the impact of shuffling as described in §4.3. The UA layer sends requests to the IA layer in randomized batches of S requests, on the way from the user to the LRS, and the IA layer does the same on the way back to the user when forwarding the response to the UA layer.

We first consider a single proxy instance per layer, and the user-to-LRS path. For a given time window, let us denote the set of messages output by the UA layer as out_{UA}, and the set of messages output by the IA layer as out_{IA}. Let us further assume that the adversary is interested in linking an incoming client request R to the related message R' reaching the LRS. Packets are encrypted and of the same size and, therefore, all outbound packets from the UA layer to the IA layer are equally likely to correspond to R. The odd for the attacker to correctly "guess" the correct outbound packet given an inbound packet from the client is $\frac{1}{|out_{\mathrm{UA}}|} = \frac{1}{S}$. Note that the same applies for responses from the LRS going back towards users.

We now factor in horizontal scaling, *i.e.*, a varying number of proxy instances in each layer. On the way from the user to the LRS, the number of instances in the UA layer does not impact unlinkability, as the adversary can observe the origin (IP address) of requests to any of the instances. We denote as I the number of instances in the IA layer. The horizontal scaling of I improves unlinkability: the probability to select the correct outbound message R' for an inbound message R becomes $\frac{1}{|out_{\text{UA}}|\times I} = \frac{1}{S\times I}$. From the LRS to the user, the number of IA layer instances has no impact, and the probability for the attacker to rightly guess that a response from the LRS is for a specific IP is $\frac{1}{S\times U}$ where U is the number of UA layer instances.

6.3 Limitations

Assumption on traffic. The effectiveness of shuffling depends on our assumption that there is sufficient traffic. In certain cases, *e.g.*, for unpopular websites or for some given periods of times (*e.g.*, at night time), this assumption may not hold for a given application. In this case, an adversary could break the unlinkability between a user and an item if, and only if, it successfully steals secrets from the IA layer in addition to timing network requests. Such an attack is difficult to orchestrate and may be of little interest for low-traffic applications. This situation may also arise for new websites, receiving low traffic in their bootstrap period. Possible mitigation would be for the RaaS provider to leverage multi-tenancy, *i.e.*, use the same proxy layer for multiple applications, thereby increasing the minimum traffic. This comes, however, with increased risks in case an enclave is broken, as secrets for multiple applications could be stolen at once.

History-based attacks. Shuffling makes a *specific* input query undistinguishable from $S-1$ others. An adversary targeting a specific IP address could collect over time a series of associated sets of S queries to the LRS. If the corresponding user repeatedly receives the same recommendations, or inserts feedback for the same items, the adversary could identify recurrent pseudonymized items identifiers and associate them with that IP address, and learn the associated pseudonymized user identifier. If such attacks are a concern, a solution is to trade off latency for privacy, using an HTTP redirection from the service using RaaS rather than issuing queries directly from clients, thereby hiding their IP addresses.

Disabling item pseudonymization. In PProx, we send pseudonymous item identifiers to the LRS by default. For a large fraction of recommendation algorithms, and in particular those based on collaborative filtering, the use of pseudonymous items has no impact and is recommended for increased privacy. For algorithms that would need item identifiers in the clear, *e.g.*, for recommendations based on the semantics of the items [55], it is easy to disable the pseudonymization of items, by using i directly instead of $\textbf{det\_enc}(i, \text{k}_{\text{IA}})$ for calls to the LRS. This would have, however, an impact on our provided security properties. Accepting that items be sent in clear requires, indeed, to lower down our assumed adversary to still preserve unlinkability between users and their interests. This is an example of the privacy-utility tradeoff: Disabling item pseudonymization means unlinkability is preserved if and only if UA enclaves are not broken.

7 Integration and Reproducibility

We integrate PProx with a representative LRS, the *Universal Recommender* [7] (UR), initially developed for Apache Mahout and the prediction.io frameworks and integrated with Harness [6], an open-source machine learning platform. UR implements collaborative filtering based on the Correlated Cross-Occurrence (CCO) algorithm. CCO aggregates indicators (in our setup, feedback on the access to items) and builds profiles allowing to predict users' interests based on the history of other profiles with high similarity.

Harness uses several modules to support the UR model construction and the generation of predictions. A MongoDB database persists engine-related data and inputs pending processing (*i.e.*, feedback received via post requests). UR uses an elasticsearch instance to persist the recommendation mode, and periodic runs of Apache Spark for rebuilding this model including new inputs fetched from MongoDB. Harness frontend modules provide a REST API allowing to query the model and return JSON-encoded recommendations. These frontend modules handle the most significant part of the load. All modules can scale horizontally by adding new instances. We further detail the performance and scaling of Harness supporting UR in our evaluation (§8.2).

7.1 Workload injection and stub LRS

We built an HTTP load injector based on the high-performance loadtest library [35] for node.js. The injector issues REST API calls and times their execution. When testing PProx in isolation from Harness, we use a stub service with the nginx high-performance HTTP server to serve a static payload of the same size as Harness recommendations lists.

7.2 Experimental reproducibility

All of our code and experimental material are available open-source (link at the end of the paper). We target the experimental reproducibility of our results through the use of an "everything-as-code" approach. All components (PProx, Harness, our workload injector, and nginx) are deployed as Docker containers in a cluster managed with MaaS [23] and running Kubernetes [19] v1.15, deployed using Kubespray v2.12.3. Since support for Intel SGX is yet to be integrated into the main version of Kubernetes we used the Kubernetes Device Plugin for Intel SGX developed by Vaucher et al. [76]. The deployment and configuration of all containers composing the system rely on charts for the Helm [27] package manager. We implement horizontal scaling of PProx proxy layers and of all Harness modules using Kubernetes integrated load balancing mechanisms (kube-proxy module). We collect logs in a systematic fashion using fluentd [37] and store them in a MongoDB instance separate from the one used by Harness. Experiments are described by Jupyter [67] notebooks in order to systematize deployment, orchestration and analysis of experimental results, and allow other researchers to reproduce them.

8 Evaluation

We evaluate PProx using the reproducible experimental setup presented in the previous section. All our deployments are performed on a cluster of 27 nodes. Each node is an Intel Next Unit of Computing (NUC) Kit with a 2-core 3.50 GHz Intel i7 processor and 32 GB of RAM[6].

Our evaluation aims at answering the following research questions: (1) What is the impact of each of the privacy-enabling features of PProx (encryption, use of SGX, and request shuffling) on service latency? (2) How does the performance of Harness equipped with

[6] One of the models recommended by Intel for experimenting with SGX.

PProx compare to an unprotected deployment? (3) Is PProx able to scale to handle larger Harness deployments, and what are the comparative costs of the two sub-systems?

We answer the question (1) through a series of micro-benchmarks with PProx connected to a stub server. We answer questions (2) and (3) through macro-benchmarks of PProx connected to Harness, with increasingly large deployments.

Metrics and workload. Our primary evaluation metric is the *distribution* of round-trip service latencies, as measured by workload injector instance(s). When measuring the performance of a given configuration with an increasing number of requests per second (RPS), we present results up to the last value measured before reaching saturation (*i.e.*, where latencies increase drastically due to congestion). This allows measuring the supported workload under acceptable conditions rather than the peak throughput, which comes at the price of very high latencies and is, therefore, of little interest in our context. We run each experiment (*i.e.*, for each configuration and RPS pair) 6 times and report the aggregated distribution of round-trip service latencies[7].

The target Service-Level Objective (SLO) for round-trip service latency depends on the nature of the application or website using RaaS services. As a rule of thumb, we consider in this evaluation that a median latency below 300 ms (not accounting the latency to and from the data center hosting the RaaS services) and never exceeding twice that value should comply with typical SLOs for online services [69][8]. We use the MovieLens dataset `ml-20m` [44, 47] as our experimental workload. This dataset is classically used for the evaluation of recommender systems. It contains feedbacks (ratings and free-text reviews) from users for movies on the collaborative MovieLens website. We use the years 2014 and 2015 as a source of feedback, corresponding to 562,888 ratings for 17,141 different movies made by 7,288 different users. In all of our experiments, we proceed in two phases: We inject feedback for one minute and trigger the training phase of UR (using Apache Spark) in a first phase, and collect recommendations for a duration of 5 minutes in a second phase. Note that: (1) We do not report on the *quality* of recommendations. This is an orthogonal concern for PProx that depends on the LRS. Recommendations are strictly the same as when using UR in Harness directly. (2) We focus on reporting the performance of `get` requests, as these are the costlier in terms of encryption and payload.[9] (3) We trim the first and last 15 seconds of each measurement period to avoid perturbations linked with the warm-up and slow-down of injection.

8.1 Micro-benchmarks

Our micro-benchmarks connect the PProx proxy service to the `nginx` stub returning static recommendations. We consider the configurations listed in Table 2, with various configurations of PProx allowing to analyze the contribution of each security-enabling feature (use of encryption, use of SGX enclaves, use of requests shuffling) in configurations `m1-m6` and the scalability of the proxy service in configurations `m6-m9`. We use one (for `m1-7`) or two (for `m8-9`) injector nodes and increments of 50 RPS (for `m1-6`, using a single instance in each proxy layer) or 250 RPS (for `m6-m9`, when analyzing scalability). The single `nginx` server is not a bottleneck: Direct requests from the injector(s) to the stub have a median latency of 1 to 2 ms and scale well over 1,000 RPS.

	§	Fig.	Enc.	SGX	S	UA	IA	RPS
m1	8.1.1	6	✗	✗	✗	1	1	250
m2	8.1.1	6	✓	✗	✗	1	1	250
m3	8.1.1	6, 7	✓	✓	✗	1	1	250
m4	8.1.1	6	★	✓	✗	1	1	250
m5	8.1.1	7	✓	✓	5	1	1	250
m6	8.1.1, 8.1.2	7, 8	✓	✓	10	1	1	250
m7	8.1.2	8	✓	✓	10	2	2	500
m8	8.1.2	8	✓	✓	10	3	3	750
m9	8.1.2	8	✓	✓	10	4	4	1000

Table 2. Micro-benchmark configurations.
"§" and "Fig." resp. denote the section(s) and figure(s) in which the configuration is used. "Enc." stands for the use of encryption, with ★ denoting that item pseudonymization is disabled. "S" is the shuffling parameter, "UA" and "IA" the number of nodes in each proxy service layer, and "RPS" the maximal amount of Requests Per Second supported by this configuration.

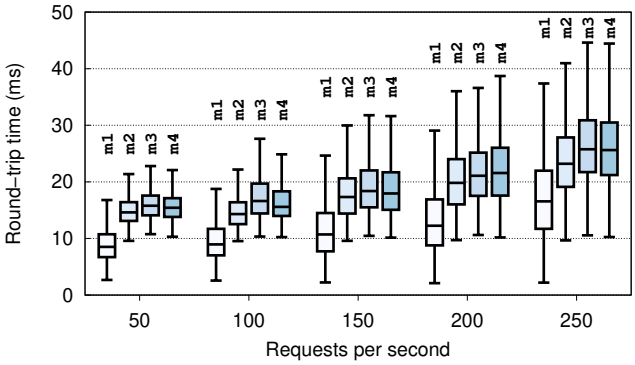

Figure 6. Performance of the proxy service with no security-enabling feature (`m1`), when adding encryption (`m2`), and when adding the use of SGX enclaves (`m3`); Impact of disabling item pseudonymization (`m4`).

8.1.1 Dissecting the impact of privacy features

Figure 6 presents the distribution of latencies when adding each of the security-enabling features of PProx one by one, except shuffling that we evaluate separately. We emphasize that reported values are the requests round-trip time, *i.e.*, requests traverse the UA and IA layer twice, once in each direction. We can observe that the added cost of encryption is slightly higher than the cost of using SGX enclaves. The use of SGX enclaves introduces 2 to 5 ms additional median or maximal latency, about half as much as adding encryption. We also disable in configuration `m4` the use of pseudonymization for item identifiers, as discussed in §6.3. The impact is negligible, confirming that using pseudonymous item identifiers can remain the default unless explicitly required by the recommendation algorithm.

[7] Each such distribution is represented as a candlestick chart: the box boundaries represent the 25th and 75th percentiles of the distribution: the difference between these two values is the interquartile range (IQR). The middle line in each box represent the median. The whiskers extend from the end of the box to the most distant point whose value lie within 1.5 times the IQR starting from the box boundary.
[8] For instance, Google representatives reported back in 2006 that search results displayed in more than 500 ms resulted in drops of 20% in traffic [42].
[9] We evaluated the costs of `post` requests and these systematically follow the same trends as for `get` requests, with only marginally lower latencies.

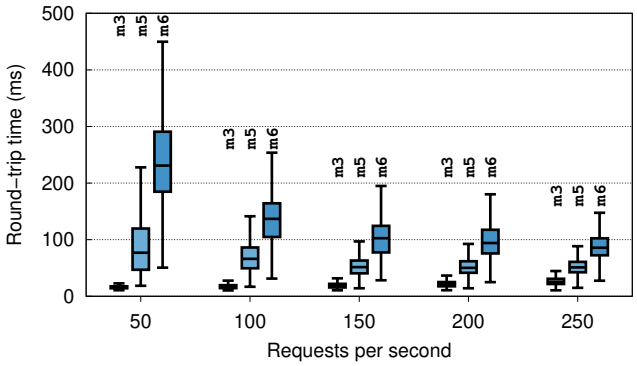

Figure 7. Impact of shuffling: reference configuration with no shuffling (m3), and with $S = 5$ (m5) and $S = 10$ (m6).

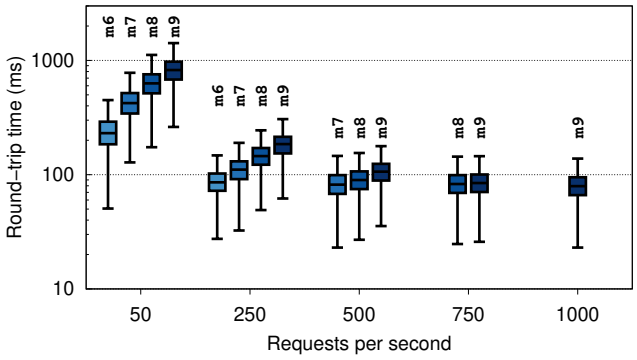

Figure 8. Scalability of PProx using 1 (m6) to 4 (m9) instances in each proxy layer (2 to 8 nodes), using all privacy-enabling features and $S = 10$.

Figure 7 compares the performance of a configuration with no shuffling (m3, same as in Figure 6) with configurations using shuffling. The impact of shuffling depends, unsurprisingly, on the number of requests received per second, impacting the time required to fill the buffer and send requests in a random order to the next stage (in both directions, from the UA to the IA, and from the IA to the UA). With a value of $S = 5$ and low throughput of 50 RPS, latency remains within usable boundaries for building an interactive service (at most a few hundred milliseconds) but can be too high for most SLOs when $S = 10$. With a larger number of requests per second, median round-trip service latency remains well below 200 ms in both cases.

8.1.2 PProx proxy service scaling

We finally evaluate the ability of the PProx proxy to scale and handle higher throughputs, starting from the complete configuration m6 with all features and $S = 10$ from our previous experiment and using only one instance per proxy service layer. We report the results in Figure 8. Note that starting from this figure and for the rest of this section we switch the ordinates to a logarithmic scale for readability.

Using more proxy instances in each layer allows supporting increasing amounts of requests, *i.e.*, each additional pair of UA and IA proxy instances enables an additional 250 RPS without reaching saturation. With 4 instances of each proxy, PProx can offer round-trip latencies that are consistently under 200 ms for 1.000 RPS[10]. We also confirm the observation made in Section 5: When using an over-provisioned system (*e.g.*, m7-9 with 50 RPS or m9 with 250 RPS) latencies due to request shuffling may become too high to comply with the recommendation service SLO and, therefore, the number of proxy instances should ideally be elastically scaled down.

	Fig.	Enc.	SGX	S	UA	IA	LRS	RPS
–baseline configurations: only LRS–								
b1	9	✗	✗	✗	✗	✗	7: 3+4	250
b2	9	✗	✗	✗	✗	✗	10: 6+4	500
b3	9	✗	✗	✗	✗	✗	13: 9+4	750
b4	9	✗	✗	✗	✗	✗	16: 12+4	1000
–full configurations: proxy service and LRS–								
f1	10	✓	✓	10	1	1	7: 3+4	250
f2	10	✓	✓	10	2	2	10: 6+4	500
f3	10	✓	✓	10	3	3	13: 9+4	750
f4	10	✓	✓	10	4	4	16: 12+4	1000

Table 3. Macro-benchmark experimental configurations.
"Fig." denotes the figure using the configuration. "Enc." stands for the use of encryption, S is the shuffling parameter, UA, IA and LRS are the number of nodes allocated to the proxy service layers and the LRS (front-end + support nodes). "RPS" is the maximal throughput achievable with this configuration.

8.2 Macro-benchmarks: PProx with the Harness LRS

We deploy PProx and Harness using the configurations listed in Table 3. Configurations b1-4 are for Harness deployed alone. They serve as a baseline. We vary the number of Harness front-end services from 3 to 12, and use 4 nodes for support services (three for Elasticsearch, one for MongoDB and Apache Spark). The front-end service is the main source of load for serving requests and these 4 support nodes are necessary and sufficient in all configurations. This translates to Harness configurations of 7 to 16 nodes.

Figure 9 presents Harness baseline performance. As previously, we present round-trip service latency for each configuration before reaching saturation. For instance, configuration b3 with 13 nodes can serve 750 RPS with sub-second latency but saturates with 1.000 RPS. The service time latencies of Harness are representative of the type of algorithm used, that require non-trivial reads to a shared database and complex (pre-built) user models to generate recommendations. These service times are below 100 ms in all configurations for low to moderate throughput (up to 500 RPS) and the spread of the distribution becomes wider for higher throughput, with peak service times around 300 ms for the largest configuration b4 under 1.000 RPS.

Figure 10 finally presents the performance of the complete integrated system, with PProx using 2 to 8 nodes and Harness using 7 to 16 nodes. These configurations f1-4 correspond to the combination of previously-detailed configurations m6-9 for PProx and b1-4 for Harness, supporting the same multiple of 250 RPS as maximal throughput prior to saturation. These configurations include all privacy-enabling features of PProx and use $S = 10$. The infrastructure cost of PProx ranges, therefore, from 30% (f1) to 50% (f4) additional nodes compared to privacy-unprotected Harness. Latencies are, as expected, the sum of latencies observed in Figures 8

[10]We emphasize that the NUCs used in our evaluation only feature two cores and mobile-grade CPUs; we expect the supported throughput to also scale *vertically* using server-grades CPUs with support for SGX.

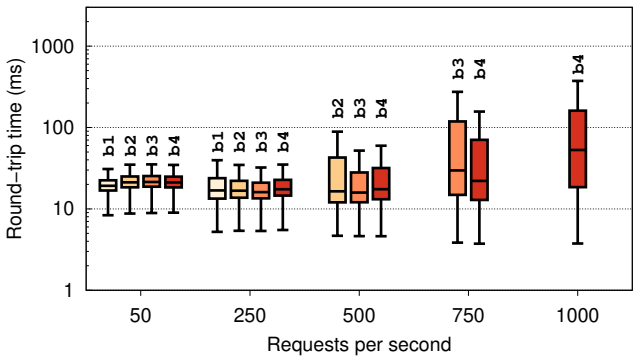

Figure 9. Baseline performance of the Harness LRS.

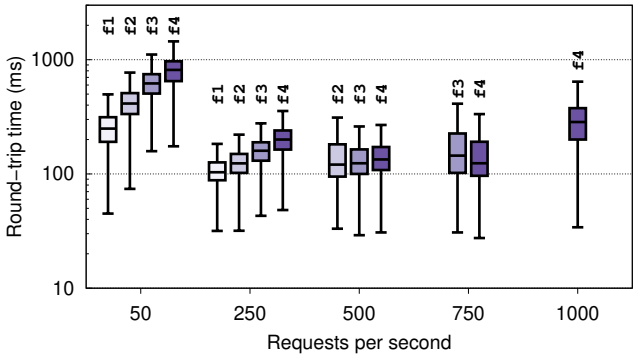

Figure 10. Performance of Harness when used in combination with PProx with increasingly large deployments.

and 9. With 50 RPS the impact of request shuffling is important, in particular for configurations f2-4. This is intrinsic to the need to prevent network observation attacks, as previously observed, and a lower value of S would shift the privacy-performance tradeoff towards the latter. For workloads of 250 to 750 RPS, however, overall latencies are systematically below 300 ms, with a median between 100 and 200 ms. With 1,000 RPS the max service time increases to 450 ms but median latency remains below 200 ms. Delete: ►We can contrast these latency results with the experimental evaluation of privacy-preserving recommendation algorithms based on encrypted processing [12, 13, 77] yielding latencies of *several* seconds.◄ Evaluations of privacy-preserving recommendation algorithms based on encrypted processing by other researchers often yield, in contrast, latencies for client requests that exceed several seconds and require non-trivial client-side computations. CryptoRec [77] reports, for instance, exchanges of multi-MB messages between clients and servers, and an order of magnitude of 10s of seconds of computation on both side to prepare and decrypt the results. Basu *et al.* [12, 13] analyzed the performance of an homomorphically-encrypted variant of the Slope One collaborative filtering algorithm [53] running on the Google App Engine and Amazon AES cloud platforms. They report base latencies for get queries in the order of several seconds, as well as a sharp increase in these latencies as the number of items in the dataset increases.

9 Related Work

Privacy violations in recommendation systems received considerable attention [16, 38]. Representative risks are the inference of individual users' profiles from temporal changes in the public outputs of a recommender system [21], or statistical deanonymization attacks [62]. Surveyed users generally consider that recommendation systems violate their privacy [59] and would prefer not to be profiled [10].

Privacy preservation can involve cryptographic schemes such as homomorphic encryption, to compute recommendations over encrypted user preferences, *e.g.*, using X-Rec [46] or CryptoRec [77]. These solutions have a high computational overhead, leading to high latencies in collecting recommendations. Slope One predictors [11] evaluations using support for homomorphic computations of the Paillier cryptosystem [66] report, indeed, latencies in the order of several seconds in public clouds [12, 13], similarly as for CryptoRec [77]. PProx only imposes a limited latency on top of the base performance of an unmodified LRS.

Differential privacy limits the disclosure of private information of records in the result of aggregate queries in a statistical database [34]. In the context of recommender systems, differential privacy can be used to add noise and obfuscate user preferences in the LRS storage and replies [39, 70, 71]. Such approaches come with a difficult-to-set tradeoff on the quality of recommendations. Under our fault model, the noise should further be added *before* sending put requests to the cloud, requiring the provision of user-side code with specific models. In contrast, PProx does not degrade the quality of recommendations and enables easy deployment.

A final approach to privacy preservation is to *distribute* the computation. Two recent approaches have been proposed in this context: the decentralized computation of recommendation models (e.g., Matrix Factorization) as in PDMFRec [33] or the use of Federated Learning principles for training recommendation models as in FedFast [61] and Fleet [30]. Other approaches include the pre-aggregation of several users' profiles and the use aggregated profiles in the cloud [72], or peer-to-peer approaches computing an overlay of nodes based on similar interests [15]. Data decentralization reduces risks of leaks in the cloud but increases such risks during direct exchanges between users. These solutions have to rely on additional noise to protect individual profiles, impacting the quality of recommendations, and their deployment is far from trivial (*e.g.*, considering NATs, firewalls, or the possibility of malware).

X-Search [60] implements web search proxies in SGX to protect the link between users and their search queries. While this presents similarities with user-interest unlinkability, X-Search employs fake queries to obfuscate this information. Such an approach would not apply to a recommender system as it would degrade the quality of recommendations. SGX-Tor [49] leverages SGX to strengthen the security of Tor. Shuffling in PProx presents similarities with onion routing in Tor, in that it helps prevent an adversary observing network exchanges from determining communication endpoints. Unlike PProx, and similarly to other work employing SGX [50, 51], X-Search or SGX-Tor do not consider the possibility for an adversary to steal secrets from an enclave.

10 Conclusion

We presented PProx, a system for privacy preservation fitting the requirements of Recommendation-as-a-Service. PProx contributes a privacy-preserving proxy service, that prevents the disclosure of the link between individuals and their interests. The security

guarantees of PProx hold even in the presence of a powerful attacker able to use recently-documented side-channel attacks on SGX enclaves, and observing all network traffic in the cloud. In contrast with previous work, privacy-preservation with PProx is not specific to a recommendation algorithm and does not require complex deployment of code or state at the users' side. In our future work, we intend to explore the use of PProx foundations for privacy preservation in general services accessed through REST APIs, and for easing the automation of pseudonymization in systems handling sensitive user data in untrusted clouds.

Artifact availability: The code of PProx together with all material allowing the reproduction of our experiments is available at the companion repository:

https://github.com/CloudLargeScale-UCLouvain/PProx.

Acknowledgments

We thank our shepherd, Roy Campbell, and the anonymous reviewers for their comments. This work was partially funded by the French-German ANR-DFG project PRIMaTE (ANR-17-CE25-0017), and by the Belgian FNRS project DAPOCA (33694591).

References

[1] 2019. Mediego. https://www.mediego.com/en/.
[2] 2019. Plista. https://www.plista.com.
[3] 2019. Recombee. https://www.recombee.com.
[4] 2020. Intel SGX SDK. https://software.intel.com/en-us/sgx/sdk.
[5] 2020. Intel Software Guard Extensions SSL. https://github.com/intel/intel-sgx-ssl.
[6] ActionML. [n. d.]. Harness: microservice based Machine Learning Server. https://actionml.com/harness
[7] ActionML. [n. d.]. The Universal Recommender. https://actionml.com/docs/h_ur
[8] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. 2014. Impact of response latency on user behavior in web search. In *37th international ACM SIGIR conference on Research & development in information retrieval*.
[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[10] Naveen Farag Awad and Mayuram S Krishnan. 2006. The personalization privacy paradox: an empirical evaluation of information transparency and the willingness to be profiled online for personalization. *MIS quarterly* (2006), 13–28.
[11] Anirban Basu, Jaideep Vaidya, Hiroaki Kikuchi, and Theo Dimitrakos. 2011. Privacy-preserving collaborative filtering for the cloud. In *23rd International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE.
[12] Anirban Basu, Jaideep Vaidya, Hiroaki Kikuchi, and Theo Dimitrakos. 2013. Privacy-preserving collaborative filtering on the cloud and practical implementation experiences. In *Sixth IEEE International Conference on Cloud Computing*.
[13] Anirban Basu, Jaideep Vaidya, Hiroaki Kikuchi, Theo Dimitrakos, and Srijith K Nair. 2012. Privacy preserving collaborative filtering for SaaS enabling PaaS clouds. *Journal of Cloud Computing: Advances, Systems and Applications* 1, 1 (2012), 8.
[14] Joeran Beel, Alan Griffin, and Conor O'Shea. 2019. Darwin & Goliath: A White-Label Recommender-System As-a-Service with Automated Algorithm-Selection. In *Demonstration at the 13th ACM Conference on Recommender Systems (RecSys)*.
[15] Yahya Benkaouz, Mohammed Erradi, and Anne-Marie Kermarrec. 2016. Nearest Neighbors Graph Construction: Peer Sampling to the Rescue. In *International Conference on Networked Systems (NETYS)*. Springer.
[16] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. 2013. Recommender systems survey. *Knowledge-based systems* 46 (2013).
[17] Antoine Boutet, Davide Frey, Rachid Guerraoui, Arnaud Jégou, and Anne-Marie Kermarrec. 2016. Privacy-preserving distributed collaborative filtering. *Computing* 98, 8 (2016), 827–846.
[18] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOTS)*.
[19] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Sixth ACM Symposium on Cloud Computing (SOCC)*.
[20] Robin Burke. 2002. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction* 12, 4 (2002).
[21] Joseph A Calandrino, Ann Kilzer, Arvind Narayanan, Edward W Felten, and Vitaly Shmatikov. 2011. "You might also like:" Privacy risks of collaborative filtering. In *IEEE Symposium on Security and Privacy (S&P)*.
[22] John Canny and John Canny. 2002. Collaborative filtering with privacy via factor analysis. In *25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM.
[23] Canonical. [n. d.]. MaaS: Very fast server provisioning for your data centre. https://maas.io
[24] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *European Symposium on Security and Privacy (EuroS&P)*. IEEE.
[25] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *ACM Asia Conference on Computer and Communications Security (AsiaCrypt)*.
[26] Richard Cissée and Sahin Albayrak. 2007. An agent-based approach for privacy-preserving recommender systems. In *6th international joint conference on Autonomous agents and multiagent systems (AAMAS)*. ACM.
[27] Cloud Native Computing Foundation. [n. d.]. Helm: The package manager for Kubernetes. https://helm.sh
[28] Stefan Contiu, Laurent Réveillère, and Etienne Rivière. 2020. Practical Active Revocation. In *21st International Middleware Conference*.
[29] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.
[30] Georgios Damaskinos, Rachid Guerraoui, Anne-Marie Kermarrec, Vlad Nitu, Rhicheek Patra, and Francois Taiani. 2020. FLeet: Online Federated Learning via Staleness Awareness and Performance Prediction. In *ACM Middleware*.
[31] Cameron Desrochers. 2020. Lock-free queue for C++11. https://github.com/cameron314/concurrentqueue.
[32] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. *Tor: The second-generation onion router*. Technical Report. Naval Research Lab Washington DC.
[33] Erika Duriakova, Elias Z Tragos, Barry Smyth, Neil Hurley, Francisco J Peña, Panagiotis Symeonidis, James Geraci, and Aonghus Lawlor. 2019. PDMFRec: a decentralised matrix factorisation with tunable user-centric privacy. In *13th ACM Conference on Recommender Systems (RecSys)*.
[34] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation (TAMC)*. Springer.
[35] Alex Fernández. 2019. alexfernandez/loadtest. https://github.com/alexfernandez/loadtest original-date: 2013-06-21T23:50:01Z.
[36] Daniel M Fleder and Kartik Hosanagar. 2007. Recommender systems and their impact on sales diversity. In *8th ACM conference on Electronic commerce*. ACM.
[37] Fluentd project. [n. d.]. Fluentd: an open source data collector for unified logging layer. https://www.fluentd.org
[38] Arik Friedman, Bart P Knijnenburg, Kris Vanhecke, Luc Martens, and Shlomo Berkovsky. 2015. Privacy aspects of recommender systems. In *Recommender Systems Handbook*. Springer.
[39] Chen Gao, Chao Huang, Dongsheng Lin, Depeng Jin, and Yong Li. 2020. DPLCF: Differentially Private Local Collaborative Filtering. In *43rd International Conference on Research and Development in Information Retrieval (ACM SIGIR)*.
[40] Florent Garcin, Boi Faltings, Olivier Donatsch, Ayar Alazzawi, Christophe Bruttin, and Amr Huber. 2014. Offline and online evaluation of news recommender systems at swissinfo.ch. In *8th ACM Conference on Recommender systems*.
[41] Mouzhi Ge, Carla Delgado-Battenfeld, and Dietmar Jannach. 2010. Beyond accuracy: evaluating recommender systems by coverage and serendipity. In *4th ACM conference on Recommender systems (RecSys)*.
[42] Google VP Marrisa Mayer. 2006. Presentation at Third Annual Web 2.0 Summit.
[43] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *10th European Workshop on Systems Security (EuroSec)*. ACM.
[44] GroupLens research at the University of Minnesota. [n. d.]. Description of the MovieLens ml-20m dataset. http://files.grouplens.org/datasets/movielens/ml-20m-README.html
[45] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium*.
[46] Rachid Guerraoui, Anne-Marie Kermarrec, Rhicheek Patra, Mahammad Valiyev, and Jingjing Wang. 2017. I know nothing about you but here is what you might like. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[47] F Maxwell Harper and Joseph A Konstan. 2015. The MovieLens datasets: History and context. *ACM transactions on interactive intelligent systems (TIIS)* 5, 4 (2015).
[48] Mike Hintze and Khaled El Emam. 2018. Comparing the benefits of pseudonymisation and anonymisation under the GDPR. *Journal of Data Protection & Privacy* 2, 2 (2018).
[49] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2018. SGX-Tor: A Secure and Practical Tor Anonymity Network With SGX Enclaves. *IEEE/ACM Transactions on Networking* 26, 5 (2018).
[50] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *14th*

ACM SIGOPS EuroSys Conference.
[51] Vaibhav Kulkarni, Bertil Chapuis, and Benoît Garbinato. 2017. Privacy-preserving location-based services by using Intel SGX. In *1st International Workshop on Human-centered Sensing, Networking, and Systems*. ACM.
[52] Kristen LeFevre, David J DeWitt, and Raghu Ramakrishnan. 2005. Incognito: Efficient full-domain k-anonymity. In *ACM SIGMOD international conference on Management of data*.
[53] Daniel Lemire and Anna Maclachlan. 2005. Slope one predictors for online rating-based collaborative filtering. In *International Conference on Data Mining*. SIAM.
[54] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference (ATC)*.
[55] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro. 2011. Content-based recommender systems: State of the art and trends. In *Recommender systems handbook*. Springer, 73–105.
[56] Frank McSherry and Ilya Mironov. 2009. Differentially private recommender systems: Building privacy into the Netflix prize contenders. In *15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*. ACM.
[57] Marc Mendonca, Srini Seetharaman, and Katia Obraczka. 2012. A flexible in-network IP anonymization service. In *International conference on communications (ICC)*. IEEE.
[58] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer.
[59] Itishree Mohallick, Katrien De Moor, Özlem Özgöbek, and Jon Atle Gulla. 2018. Towards New Privacy Regulations in Europe: Users' Privacy Perception in Recommender Systems. In *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage (SpaCCS)*. Springer.
[60] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. 2017. X-search: revisiting private web search using intel SGX. In *18th ACM/IFIP/USENIX Middleware Conference*.
[61] Khalil Muhammad, Qinqin Wang, Diarmuid O'Reilly-Morgan, Elias Tragos, Barry Smyth, Neil Hurley, James Geraci, and Aonghus Lawlor. 2020. FedFast: Going Beyond Average for Faster Training of Federated Recommender Systems. In *26th International Conference on Knowledge Discovery & Data Mining (ACM SIGKDD)*.
[62] Arvind Narayanan and Vitaly Shmatikov. 2008. Robust de-anonymization of large datasets (how to break anonymity of the Netflix prize dataset). In *IEEE Symposium on Security and Privacy*.
[63] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A Survey of Published Attacks on Intel SGX. arXiv:cs.CR/2006.13598
[64] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX Annual Technical Conference (ATC)*.
[65] Emanuel Onica, Pascal Felber, Hugues Mercier, and Etienne Rivière. 2015. Efficient key updates through subscription re-encryption for privacy-preserving publish/subscribe. In *16th Annual Middleware Conference*.
[66] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques (Eurocrypt)*. Springer.
[67] Project Jupyter. [n. d.]. Open-source software, open-standards, and services for interactive computing across dozens of programming languages. https://jupyter.org
[68] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726* (2019).
[69] Ron Sharp. 2012. Latency in cloud-based interactive streaming content. *Bell Labs Technical Journal* 17, 2 (2012).
[70] Yilin Shen and Hongxia Jin. 2014. Privacy-preserving personalized recommendation: An instance-based approach via differential privacy. In *International Conference on Data Mining (ICDE)*. IEEE.
[71] Hyejin Shin, Sungwook Kim, Junbum Shin, and Xiaokui Xiao. 2018. Privacy enhanced matrix factorization for recommendation with local differential privacy. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018).
[72] Reza Shokri, Pedram Pedarsani, George Theodorakopoulos, and Jean-Pierre Hubaux. 2009. Preserving privacy in collaborative filtering through distributed aggregation of offline profiles. In *3rd ACM conference on Recommender systems (RecSys)*. ACM.
[73] Latanya Sweeney. 2002. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 557–570.
[74] André Calero Valdez and Martina Ziefle. 2019. The users' perspective on the privacy-utility trade-offs in health recommender systems. *International Journal of Human-Computer Studies* 121 (2019).
[75] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*.
[76] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. 2018. SGX-aware container orchestration for heterogeneous clusters. In *38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.
[77] Jun Wang, Qiang Tang, Afonso Arriaga, and Peter YA Ryan. 2019. Novel Collaborative Filtering Recommender Friendly to Privacy Protection. In *International Joint Conference on Artificial Intelligence (IJCAI)*.
[78] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
[79] Zhenyu Wu, Zhang Xu, and Haining Wang. 2014. Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Transactions on Networking* 23, 2 (2014), 603–615.