PProx: efficient privacy for recommendation-as-a-service
Guillaume Rosinosky, Simon da Silva, Sonia Ben Mokhtar, Daniel Négru, Laurent Réveillère, Etienne Rivière

To cite this version:

HAL Id: hal-03752290
https://hal.science/hal-03752290
Submitted on 16 Aug 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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:

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 Reombee [3]. In the RaaS service model, illustrated in Figure 1, application providers delegate the collection 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,
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, 73, 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 ( EventArgs). 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 ( EventArgs).

The RaaS runs a legacy recommendation system (LRS) for the application ( EventArgs), accessed via a simple REST API. A post (u, i, p, l) 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 get (u) request returns a collection of n items (i1, . . . , in) recommended to user u.

PProx introduces an additional component, the privacy-preserving proxy service ( EventArgs), 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 ( EventArgs).

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
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 (➊). In addition, it may monitor network flows between the nodes forming this infrastructure, both with the outside world and internally (➋), 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].

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
identifiers and 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 corresponds 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.

| $u$ | 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 |
| $pk_{UA}$ | Public key of User Anonymizer layer |
| $sk_{UA}$ | Private key of User Anonymizer layer |
| $pk_{IA}$ | Public key of Item Anonymizer layer |
| $sk_{IA}$ | 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 $u$ |
| $\text{enc}(x, \{p|s\}k)$ | asymmetric encryption of $x$ using public/private key $k$ |
| $\text{det}_{\text{enc}}(x, k)$ | 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 $sk_{UA}$ and a permanent symmetric key $k_{UA}$. The IA layer is similarly provisioned with $sk_{IA}$ 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 it into $\text{enc}(u, pk_{UA})$ so that only the UA layer may recover $u$ from the ciphertext using $sk_{UA}$. 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 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 $i$, 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

---

*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].*
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 `post(u, i)` by encrypting the two arguments, yielding a new call

\[
\text{post(\text{enc}(u, \text{pk}_{UA}), \text{enc}(i, \text{pk}_{IA}))}
\]

that is sent to the UA layer. This layer decrypts u using private key \(\text{pk}_{UA}\). It pseudonymizes plaintext u by deterministically encrypting it using \(\text{k}_{UA}\). The resulting call

\[
\text{post(\text{det_enc}(u, \text{k}_{UA}), \text{enc}(i, \text{pk}_{IA}))}
\]

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

\[
\text{post(\text{det_enc}(u, \text{k}_{UA}), \text{det_enc}(i, \text{k}_{IA}))}
\]

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 \(\text{k}_u\) and encrypts it using \(\text{pk}_{UA}\). This key \(\text{k}_u\) 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 layer’s public key \(\text{pk}_{IA}\). The user identifier is encrypted, as for a post request, using the UA layer’s public key, yielding the call

\[
\text{get(\text{enc}(u, \text{pk}_{UA}), \text{enc}(\text{k}_u, \text{pk}_{IA}))}
\]

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

\[
\text{get(\text{det_enc}(u, \text{k}_{UA}), \text{enc}(\text{k}_u, \text{pk}_{IA}))}
\]

to the IA layer. The IA layer then sends \(\text{get(\text{det_enc}(u, \text{k}_{UA}), \text{enc}(\text{k}_u, \text{pk}_{IA}))}\) to the LRS. The returned list

\[
\{\text{det_enc}(i_1, \text{k}_{IA}), \ldots, \text{det_enc}(i_n, \text{k}_{IA})\}
\]

contains pseudonymized item identifiers. These identifiers are decrypted to plaintext item identifiers used by the application using \(\text{k}_{IA}\). The recommendations list is then re-encrypted to hide it from the UA layer using the user key \(\text{k}_u\), yielding \(\text{enc}(\{i_1, \ldots, i_n\}, \text{k}_u)\).

The call traverses back the layers until the user-side library, which decrypts the list of recommended item identifiers using \(\text{k}_u\) 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
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.

**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 output 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

---

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.

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.
We present in this section the security analysis of PProx.

To break the unlinkability between a user adversary must either (1) leak information from the LRS database or (2) get access to items recommended by the LRS in response to a 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.

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 post(u, i, [p]) message sent by u; (2) get access to items recommended by the LRS in response to a 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 skUA used to decrypt the user identifier u contained in a transformed post(u, i, [p]) message; and the permanent key kUA used to encrypt the same user identifier u toward its storage by the LRS in pseudonymous form det_enc(u, kUA). We consider in the following these three (not mutually exclusive) cases: (a) the adversary intercepts the transformed post(u, i, [p]) message at a UA enclave; (b) the adversary intercepts the response to the 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 post(u, i, [p]) has been transformed at the user side to post(enc(u, pkUA), enc(i, pkIA)). The adversary intercepts this message and knows the origin of the request. It can link the IP address to decrypt enc(u, pkUA) using the secret skUA. By accessing the LRS database, it may link u with det_enc(i, kIA) as it knows kUA and can thus decrypt det_enc(u, kUA). However, as long as it does not steal IA layer’s secrets, the adversary cannot decrypt det_enc(i, kIA) 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 intercepts a list of encrypted item identifiers containing enc(i, kIA). 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 kUA 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 kUA, but it is not able to de-pseudonymize items as obtaining kIA 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 skIA used to decrypt an item identifier i contained in a transformed post(u, i, [p]) message; and the permanent key kIA used to pseudonymize this item identifier i as det_enc(i, kIA) for use by the LRS. As for Case 1, we consider the following three (not mutually exclusive) cases: (a) the adversary intercepts the transformed 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., post(det_enc(u, kUA), enc(i, pkIA)). The adversary can decrypt enc(i, pkIA) using the leaked secret skIA 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 det_enc(u, kIA), having access to permanent key kIA. However, as long as it does not simultaneously break one of the UA enclaves, the adversary cannot decrypt det_enc(u, kUA) 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 det_enc(i, kIA). It can, therefore, decrypt i using the leaked secret kIA. 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 kIA, 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 outUA, and the set of messages output by the IA layer as outIA. 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 \( \frac{1}{\text{outUA}} = \frac{1}{S} \). 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 $\frac{1}{|\text{outbound}|} = \frac{1}{IU}$. 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 $\frac{1}{IU}$ 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 $S \sim 1$ 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 $\text{det}_{\text{enc}}(i,k_{\text{IA}})$ 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 Kubespark 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.

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

---

6One 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.

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]. 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 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.

**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.
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. 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.

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

### Table 3. Macro-benchmark experimental configurations.

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Enc.</th>
<th>SGX</th>
<th>S</th>
<th>UA</th>
<th>IA</th>
<th>LRS</th>
<th>RPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>✓</td>
<td>✗</td>
<td>9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>250</td>
</tr>
<tr>
<td>b2</td>
<td>✓</td>
<td>✗</td>
<td>9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>500</td>
</tr>
<tr>
<td>b3</td>
<td>✓</td>
<td>✗</td>
<td>9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>750</td>
</tr>
<tr>
<td>b4</td>
<td>✓</td>
<td>✗</td>
<td>9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1000</td>
</tr>
<tr>
<td>f1</td>
<td>✓</td>
<td>✗</td>
<td>10</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>250</td>
</tr>
<tr>
<td>f2</td>
<td>✓</td>
<td>✗</td>
<td>10</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>500</td>
</tr>
<tr>
<td>f3</td>
<td>✓</td>
<td>✗</td>
<td>10</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>750</td>
</tr>
<tr>
<td>f4</td>
<td>✓</td>
<td>✗</td>
<td>10</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>1000</td>
</tr>
</tbody>
</table>

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-grade CPUs with support for SGX.
and 9. With 50 RPS the impact of request shuffling is important, in particular for configurations 7 to 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 trade-off 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.

We can contrast these latency results with the experimental evaluation of privacy-preserving recommendation algorithms based on the decentralized computation of recommendation models 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 at-
tacker 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 pri-
vacy 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 re-
viewers for their comments. This work was partially funded by
the French-German ANR-DFG project PRIMA'TE (ANR-17-CE25-0017),
and by the Belgian FNRS project DAPOCA (33694951).

References
tion experiences. In Sixth IEEE International Conference on Cloud Computing.
tion experiences. In Sixth IEEE International Conference on Cloud Computing.
[18] Ferdinand Brasser, Urs Müller, Alexandra Dimitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache at-
tacks are practical. In 26th USENIX Workshop on Offensive Technologies (WOOT).

ulative Execution. In European Symposium on Security and Privacy (EuroSP). IEEE.
[26] Richard Cissey and Sahin Albayrak. 2007. An agent-based approach for privacy-preserving recommender systems. In 6th international joint conference on Autonomou-
ous agents and multiagent systems (AAMAS). ACM.
[48] Seongmin Kim, Juhhyung Han, Jaeheyeong Ha, Taeroo Kim, and Dongyu Han. 2018. SGX-Tor: A Secure and Practical Tor Anonymity Network With SGX Enclaves. ACM/IEEE Transactions on Dependable and Secure Networks (TDSN).
[49] Taeoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. shieldstore: Shielded In-memory Key-value Storage with SGX. In 14th


