Anonymous RAM

Michael Backes[1,2], Amir Herzberg[3], Aniket Kate[4], and Ivan Pryvalov[1]

[1] CISPA, Saarland University, Germany
[2] MPI-SWS, Germany
[3] Bar-Ilan University, Israel
[4] Purdue University, USA

Abstract. We define the concept of and present provably secure constructions for *Anonymous RAM (AnonRAM)*, a novel multi-user storage primitive that offers strong privacy and integrity guarantees. AnonRAM combines privacy features of anonymous communication and oblivious RAM (ORAM) schemes, allowing it to protect, simultaneously, the *privacy of content, access patterns* and *user's identity*, from curious servers and from other (even adversarial) users. AnonRAM further protects *integrity*, i.e., it prevents malicious users from corrupting data of other users. We present two secure AnonRAM schemes, differing in design and time-complexity. The first scheme has simpler design; like efficient ORAM schemes, its time-complexity is poly-logarithmic in the number of cells (per user), however, it is *linear* in the number of users. The second AnonRAM scheme reduces the overall complexity to poly-logarithmic in the total number of cells (of all users), at the cost of requiring two (non-colluding) servers.

Keywords: Anonymity, Access Privacy, Oblivious RAM, Out-sourced Data, (Universal) Re-randomizable Encryption, Oblivious PRF

1 Introduction

The advent of cloud-based outsourcing services has been accompanied by a growing interest in *security and privacy*, striving to prevent exposure and abuse of sensitive information by adversarial cloud service providers and users. This in particular includes the tasks of *data privacy*, i.e., hiding users' data from overly curious entities such as the provider, as well as *access privacy*, i.e., hiding information about data-access patterns such as *which* data element is being accessed and *how* (read/write?). The underlying rationale is that exposure of data access patterns may often lead to a deep exposure of *what* the user intends to do. An extensive line of research has produced impressive results and tools for achieving both data and access privacy. In particular oblivious RAM (ORAM) schemes, first introduced by Goldreich and Ostrovsky [22], have been extensively investigated in the last few years, yielding a multitude of elegant and increasingly efficient results [6, 15, 19, 20, 24, 26, 27, 29].

Another important privacy goal is to hide *who* is accessing the data, i.e., conceal the *identity* of the user, to ensure anonymity. This area spawned exten-

sive research, and multiple protocols and systems for anonymous communication [4, 5, 7, 8]. The Tor network [28] currently constitutes the most widely used representative of these works.

We focus on the combination of these two goals: hiding content and access patterns as offered by ORAM schemes, but also concealing the user identities as offered by anonymous communication protocols. Experts in the relevant areas may not be completely surprised to find that designing this primitive is quite challenging. In particular, the privacy guarantees cannot be constructed by solely combining both approaches: The naïve idea to achieve these privacy properties simultaneously is to maintain separate ORAM data structures for each user, and having users access the system using the anonymous communication protocol. However, this construction does not hide the access patterns, since the server can determine if the same data structure is accessed twice, and thereby trivially link two accesses made by the same anonymous user. Instead of multiple ORAMs, one could try to use a single ORAM as black-box with data of all users contained in it. However, this does not work either, as inherently the users have to share the same key, and the privacy properties immediately fail in presence of curious adversaries. (See Section 3 for more details.) Supporting multiple, potentially malicious (or even 'just curious') users, is significantly harder than supporting multiple cooperating clients (e.g., devices of same user), as in [10, 16, 21, 30].

Furthermore, when considering adversarial environment, and in particular, malicious users, *integrity*, i.e., preventing one user from corrupting data of other users, is also critical. Notice the (popular) 'honest-but-curious' model is easier to justify for servers (e.g., running ORAM) than for clients; handling (also) malicious client is very important. Note also that ensuring integrity is fairly straightforward, when users can be identified securely; however, this conflicts with the goals of anonymity, and even more, with the desire for oblivious access, i.e., hiding even the pattern of access to data. As often happens in security, the mechanisms for the different goals do not seem to nicely combine, resulting in a rather challenging problem, to which we offer the first - but definitely not final - pair of solutions, albeit with significant limitations and room for improvement.

Our Contributions. We define *Anonymous RAM* (AnonRAM) schemes, and present two constructions that are provably secure in the random oracle model. AnonRAM schemes support multiple users, each user owning multiple memory cells. AnonRAM schemes simultaneously hide data content, access patterns and the users' identities, against honest-but-curious servers and against malicious users of the same service, while ensuring that data can only be modified by the legitimate owner.

The first scheme, called $\mathsf{AnonRAM}_{\mathsf{lin}}$, realizes a conceptually simple transformation that turns any secure single-user ORAM scheme into a secure AnonRAM scheme (that supports multiple users). The key idea here is to convert every single-user ORAM cell to a multi-cell having a cell for each user, and to employ re-randomizable encryption such that a user can hide her identity by re-randomizing all other cells in a multi-cell while updating her own cell. The drawback of $\mathsf{AnonRAM}_{\mathsf{lin}}$, however, is that its complexity is linear in the number

of users (although poly-logarithmic in the number of cells per users). This linear complexity stems from the requirement that a user has to touch one cell of each user when accessing her own cell.

The second scheme, called $\mathsf{AnonRAM}_{\mathsf{polylog}}$, reduces the overall complexity to poly-logarithmic in the number of users. This comes at the cost of requiring two non-colluding servers S and T. Server S maintains all user data in encrypted form using a *universal* re-encryption scheme, thereby disallowing S and other users to establish a mapping between a user and her data blocks. Essentially, $\mathsf{AnonRAM}_{\mathsf{polylog}}$ constitutes an extension of hierarchical ORAM designs, e.g., by Goldreich-Ostrovski [12], where the reshuffle operation and mapping to 'dummy' blocks are performed by the dedicated server T. This prevents user deanonymization by the server S or by other users. Furthermore, mappings to specific buckets are achieved by means of a specific Oblivious PRF.

For the sake of exposition, we first describe simplified variants of both schemes in the presence of honest-but-curious users. We subsequently show how to extend both constructions to handle malicious users as well. The extension mainly involves adding an integrity element to the employed (universal) re-encryption, such that any user can only re-encrypt data of other users, but not corrupt it.

Finally, we consider it an important contribution that we present a rigorous model and definition for this challenging problem of AnonRAM, and show their suitability by providing provable security protocol instantiations.

Related Work. Several multi-*client* ORAM solutions [10, 16, 21, 30] have been proposed in the literature; however, these works do not protect privacy of a client from malicious or 'curious' clients. Franz et al. [10] introduces the concept of delegatable ORAM, where a database owner can delegate access rights to other users and periodically perform reshuffling to protect the privacy of their accesses. [16, 21] allow multiple clients to share a server-side ORAM structure, but assume that all clients share the same symmetric key which none of them is going to provide to the server. [30] deals with concurrent accesses of multiple clients devices of the same user.

AnonRAM schemes avoid this strong non-collusion assumption between the users and the storage server. In other words, we consider the problem of anonymously accessing the server by multiple users, where the server (cooperating with some users) should not be able to learn which honest user accessed which cell over the server. Notably, we achieve our stronger privacy guarantees against a stronger adversary without requiring any communication among the users.

The only other multi-*user* ORAM scheme has been proposed by Zhang et al. [18]. Their scheme uses a set of intermediate nodes to convert a user's query to an ORAM query to the server. Privacy of the scheme is, however, analyzed only for individual non-anonymous user accesses and not for multi-user anonymous access patterns. Furthermore, their scheme does not provide integrity protection against malicious users. Moreover, their work lacks both definitions and proofs; as the reader will see from our work, the definitions and proofs we found necessary to claim security of our schemes are non-trivial.

Outline. In Section 2, we introduce and define AnonRAM schemes. In Section 3 and Section 4, we present our two AnonRAM schemes. Section 5 summarizes our findings.

2 AnonRAM Definitions

We consider a set of N users $\mathcal{U} = \{\mathsf{U}_1, \ldots, \mathsf{U}_N\}$, a set of η servers $\mathcal{S} = \{\mathsf{S}_1, \ldots, \mathsf{S}_\eta\}$, a set Σ of messages, and we let M denote the number of data cells available to each user. All protocols are parametrized by the security parameter λ. Before we define the class of AnonRAM schemes, we provide the definitions of access requests and access patterns.

Definition 1 (Access Requests). *An* access request *AR is a tuple $(j, \alpha, m) \in [1, M] \times \{\mathsf{Read}, \mathsf{Write}\} \times \Sigma$. Here j is called the* (cell) index *of AR, α the* access type*, and m the* input message.

Intuitively, an access request (j, α, m) will denote that m should be written into cell j (if $\alpha = \mathsf{Write}$), or that the content of cell j should be read (if $\alpha = \mathsf{Read}$; in this case m is ignored, and we often just write $(j, \alpha, *)$).

Definition 2 (Access Patterns). *An* access pattern *is a series of tuples (i, AR_i) where $i \in [1, N]$ is a user identifier and AR_i is an access request.*

For notational simplicity, we will write (i, j, α, m) instead of $(i, (j, \alpha, m))$ for the individual elements of access patterns.

We next define AnonRAM schemes. In this work, we consider sequential schemes where one participant is active at any point in time.

Definition 3 (AnonRAM Schemes). *An* AnonRAM *scheme is a tuple* $(\mathsf{Setup}, \mathsf{User}, \mathsf{Server}_1, \ldots, \mathsf{Server}_\eta)$ *of $\eta + 2$ PPT algorithms, where:*

- *The* initialization algorithm Setup *maps a security parameter λ and an identifier id, to an initial state, where $id \in \{0, 1, \ldots, \eta\}$ identifies one of the servers (for $id > 0$), or the user (for $id = 0$).*
- *The* user algorithm User *processes two kinds of inputs: (a) access requests (from the user) and (b) pairs (l, m) where $l \in [1, \eta]$ denotes a server and m a message from server S_l.* User *maps the current state and input to a new state and to either a response provided to the user or a pair (l, m) with $l \in [1, \eta]$ denoting a server and m being a message for S_l.*
- *The* server algorithm Server_l *for server S_l maps the current server state and input (message from user or from another server), to a new server state, and a message either to the user or to another server.*

Adversarial Models and Protocol Execution. We consider two different adversarial models: (i) *honest-but-curious* (HbC) adversaries that learn the state of one server S^* and of a subset $\mathcal{U}^*$ of users, and (ii) *malicious users* ($\mathsf{Mal\_Users}$) adversaries that learn the state of one server (as before), and that additionally control a subset $\mathcal{U}^*$ of users. In both models, the adversary can additionally

eavesdrop on all messages sent on the network, i.e., between users and servers, and between two servers.

We now define the sequential execution $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP, \zeta)$ of an AnonRAM scheme $\mathcal{AR}$ in the presence of an adversary Adv and a given access pattern AP, assuming an adversarial model $\zeta \in \{\mathsf{HbC}, \mathsf{Mal\_Users}\}$.

Definition 4 (Execution). *Let $\mathcal{AR}$ be an AnonRAM scheme* $(\mathsf{Setup}, \mathsf{User}, \mathsf{Server}_1, \ldots, \mathsf{Server}_\eta)$*,* Adv *be a PPT algorithm, $\zeta \in \{\mathsf{HbC}, \mathsf{Mal\_Users}\}$ and AP be an access pattern. The* execution $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP, \zeta)$ *is the following randomized process:*

1. All parties are initialized using Setup*, resulting in initial states σ_{U_i} for each user U_i, and σ_{S_l} for each server S_l.*

2. Adv *selects a server S^* and a strict subset $\mathcal{U}^* \subset \mathcal{U}$.*

3. Let $(i, j, \alpha, m_{i,j})$ be the first element of AP; if AP is empty, terminate.

4. If $\mathsf{U}_i \in \mathcal{U}^*$ and $\zeta = \mathsf{Mal\_Users}$, then let (l, m) be the output of Adv *on input $(i, j, \alpha, m_{i,j})$. Otherwise, let (l, m) be the output of* User *on input $(j, \alpha, m_{i,j})$, with state σ_{U_i}, and update σ_{U_i} accordingly.*

5. Invoke S_l with (input) message m. The server S_l may call other servers (possibly recursively) and finally produces an (output) message m'.

6. If $\mathsf{U}_i \in \mathcal{U}^*$ and $\zeta = \mathsf{Mal\_Users}$, provide the message m' to Adv*. Otherwise, provide m' to user U_i. U_i (*Adv *if $\mathsf{U}_i \in \mathcal{U}^*$ and $\zeta = \mathsf{Mal\_Users}$) may repeat sending messages to any servers. Eventually, U_i (*Adv*) terminates.*

7. Repeat the loop (from step 3) with the next element of AP (until empty).

Throughout the execution, the adversary learns the internal states of S^* and of all users in $\mathcal{U}^*$, as well as all messages sent on the network.

A trace *is the random variable defined by an execution, using uniformly-random coin-tosses for all parties. The trace includes the sequence of messages in the execution corresponding to access requests, and the final state of the adversary. Let $\Theta(x)$ denote the trace of execution x.*

Privacy and Integrity of AnonRAM schemes. To define privacy for AnonRAM schemes, we consider an additional PPT adversary $\mathcal{D}$ called the distinguisher. $\mathcal{D}$ outputs two arbitrary access patterns of the same finite length, which differ only in inputs to unobserved users. We then randomly select and execute one of these two patterns. The distinguisher's goal is to identify which pattern was used. Since these two accesses may differ in user, cell, operation or value, this definition encompasses all relevant privacy properties in this setting, including anonymity (identity privacy), confidentiality (value privacy), and obliviousness (cell and operation privacy). We call an adversary Adv *compliant* with a pair of access patterns (AP_0, AP_1) if Adv only outputs sets $\mathcal{U}^*$ of users in Step 2) of $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_0, \zeta)$ and $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_1, \zeta)$ such that AP_0 and AP_1 are identical when restricted to users in $\mathcal{U}^*$.

Definition 5 (Privacy of AnonRAM). *An AnonRAM scheme $\mathcal{AR}$ preserves* privacy *in adversarial model $\zeta \in \{\mathsf{HbC}, \mathsf{Mal\_Users}\}$, if for every pair of (same*

finite length) access patterns (AP_0, AP_1) *and for every pair of PPT algorithms* $(\mathsf{Adv}, \mathcal{D})$ *s.t.* Adv *is compliant with* (AP_0, AP_1)*, we have that*

$$\left| \Pr\left[b^* = b \colon b^* \leftarrow \mathcal{D}\left(\Theta\left(\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_b, \zeta)\right)\right)\right] - \frac{1}{2} \right|$$

is negligible in λ*, where the probability is taken over uniform coin tosses by all parties, and* $b \leftarrow_R \{0,1\}$*.*

Note that when all-but-one (i.e., $N-1$) users are observed and $\zeta = \mathsf{HbC}$, our privacy property corresponds to the standard ORAM access privacy definition [12]. ORAM is hence a special case of AnonRAM with a single user ($N = 1$).

AnonRAM should ensure *integrity* to prevent invalid executions caused by parties deviating from the protocol. Informally, a trace is invalid if a value read from a cell does not correspond to the most recently written value to the cell.

Definition 6 (Integrity of AnonRAM). *Let* ϑ *be a trace of execution with access pattern* AP*, and let* $AR = (j, \mathsf{Read}, *)$ *with* $(i, AR_i) \in AP$ *be a read request for cell* j *of user* U_i*, returning a value* x*. Let* $AR' = (j, \mathsf{Write}, x')$ *be the most recent previous write request to cell* j *of user* U_i *in* AP*, or* $\perp$ *if there was no such previous write request. If* $x \neq x'$*, we say that this read request is* invalid*. If any read request in the trace is invalid, then the trace is invalid.*

An AnonRAM scheme $\mathcal{AR}$ *preserves* integrity *if there is negligible (in* λ*) probability of invalid traces when the traces are constrained to the view of the honest users (all* $\mathsf{U}_i \in \mathcal{U}$ *in the* HbC *model, and all users* $\mathsf{U}_i \in \mathcal{U}/\mathcal{U}^*$ *in the* $\mathsf{Mal\_Users}$ *model), for any PPT adversary and any access pattern* AP*.*

3 Linear-complexity AnonRAM

In this section, we present our first AnonRAM constructions and prove them secure in the underlying model. For the sake of exposition, we start with a few seemingly natural, but flawed approaches to construct AnonRAM schemes.

3.1 Seemingly Natural but Flawed Approaches

A natural first idea to design an AnonRAM scheme is to maintain all the $M \cdot N$ cells in encrypted form on the server and to only access them via an anonymous channel such as Tor [28]. However, this approach fails to achieve AnonRAM privacy, since the adversary can simply observe all memory accesses on the server and thereby determine how often the same cell j of a user is accessed. One may try to overcome this problem using a shared $(M \cdot N)$-cell *stateless* ORAM [16] containing M cells for each of N users, and assuming that every user executes her ORAM requests via an anonymous channel. In this case, all users will have to use the *same* private key in the symmetric encryption scheme employed in the ORAM protocol to hide their cells from the server. However, this allows Eve, an HbC user, to break privacy of honest users, by observing the values in cells

(allocated to honest users) which she downloaded and decrypted as a part of her legitimate ORAM requests.

Another natural design would be to use a separate ORAM for the M cells of each user, and rely on anonymous access to hide user identities. This use would hide the users' individual access patterns, but the server can identify all accesses by the same user and thereby violate the AnonRAM privacy requirement.

The AnonRAM schemes presented in this paper overcome such problems, by having users re-randomize cells belonging to other users as well whenever own cells are being accessed, in addition to encrypting the user's own cells.

3.2 $\mathsf{AnonRAM}_{\mathsf{lin}}$ and its Security Against HbC Adversaries

We now present the $\mathsf{AnonRAM}_{\mathsf{lin}}$ construction and prove it secure in the HbC adversarial model. $\mathsf{AnonRAM}_{\mathsf{lin}}$ uses an anonymous communication channel [28] and the (single-user, single-server) Path ORAM [27], or other ORAM scheme satisfying a property identified below.

In Path ORAM, the user's cells are stored on the server RAM as a set of encrypted data *blocks* such that each block consists of a single ciphertext, and all blocks are encrypted with the same key known to the user's ORAM client. A block encrypts either a user's cell, or auxiliary information used by the User algorithm. To access a cell, the ORAM client *reads* (and decrypts) a fixed number of blocks from the server, and *writes* encrypted values (cells or some special messages) in a fixed number of blocks. The server's duty is to execute these user's read and write requests.

$\mathsf{AnonRAM}_{\mathsf{lin}}$ employs N instances (one per user) of Path ORAM for M cells each, while requiring a single server.[5] To encrypt data as required in the ORAM scheme, $\mathsf{AnonRAM}_{\mathsf{lin}}$ uses a semantically secure re-randomizable encryption (RE) scheme $(\mathsf{E}, \mathsf{R}, \mathsf{D})$ (e.g., ElGamal encryption), where E, R, and D are respectively encryption, re-randomization, and decryption operations. The $\mathsf{AnonRAM}_{\mathsf{lin}}$ client of user U_i, has access to her private key sk_i and to the public keys $(pk_1, \ldots, pk_N)$ of all users. In $\mathsf{AnonRAM}_{\mathsf{lin}}$, the ORAM scheme uses this RE scheme $(\mathsf{E}, \mathsf{R}, \mathsf{D})$, instead of the (symmetric) encryption scheme used in 'regular' Path ORAM.

Intuitively, an $\mathsf{AnonRAM}_{\mathsf{lin}}$ client internally runs an ORAM client and mediates its communication with the server. Whenever the ORAM client reads or writes a specific block, the $\mathsf{AnonRAM}_{\mathsf{lin}}$ client performs corresponding read or write operations *for all users*, without divulging the user identity to the server at the network level as follows: Reading a block of another user can be trivially achieved, since the block is encrypted for the owner's key, but the contents are not used (our goal is only to create indistinguishable accesses for all users). Writing a block belonging to other user's ORAM must not corrupt the data inside and is hence achieved by re-randomizing the blocks of other users.

The Setup and Server algorithms of $\mathsf{AnonRAM}_{\mathsf{lin}}$, are simply N instances of the corresponding algorithm of the underlying ORAM scheme (e.g., Path ORAM).

[5] $\mathsf{AnonRAM}_{\mathsf{lin}}$ can also use an ORAM scheme that uses multiple servers. In this case, $\mathsf{AnonRAM}_{\mathsf{lin}}$ will use the same number of servers.

upon access request (j, α, m) from user U_i :
  Invoke the ORAM client $\mathcal{O}_c$ with access request (j, α, m).

upon read request from $\mathcal{O}_c$ for block j' :
  for $k \in [1, N]$ **do** Let $B[j', k] \leftarrow$ Read block j' of user U_k kept by the server.
  Return $B[j', i]$ to $\mathcal{O}_c$.

upon write request from $\mathcal{O}_c$, for value (ciphertext) B in block j' :
  $B[j', i] \leftarrow B$
  for $k \in [1, N] | k \neq i$ **do** $B[j', k] \leftarrow \mathsf{R}_k(B[j', k])$
  for $k \in [1, N]$ **do**
    Write block $B[j', k]$ to position j' of user U_k and release it from memory.

upon Receiving a result res from $\mathcal{O}_c$:
  Return res to U_i.

Fig. 1. User algorithm of $\mathsf{AnonRAM}_{\mathsf{lin}}$ with access request (j, α, m) for user U_i.

Namely, the Setup initializes state for N copies of the ORAM (one per user), and the Server receives a 'user identifier' i together with each request, and runs the ORAM's Server algorithm using the i^{th} state over the request. The Server algorithm for the $\mathsf{AnonRAM}_{\mathsf{lin}}$ scheme simply processes Read/Write requests sent by the users as in the ORAM scheme, e.g. the server returns the content of the requested block for Read requests, or overrides the content of the requested block with the new value for Write requests.

We finally describe the User algorithm of $\mathsf{AnonRAM}_{\mathsf{lin}}$ using pseudocode in Fig. 1 to increase readability. It relies on an oracle $\mathcal{O}_c$ for the ORAM client, and an RE scheme $(\mathsf{E}, \mathsf{R}, \mathsf{D})$. We write $(\mathsf{E}_i, \mathsf{R}_i, \mathsf{D}_i)$ for the corresponding encryption, re-encryption and decryption operations, using the corresponding keys for user U_i. The pseudocode depicts which operations are performed for an individual access request (j, α, m) of user U_i. Its execution starts with invoking user U_i's local ORAM client $\mathcal{O}_c$ with the access request (j, α, m), and ends with a **Return** message to U_i. The process involves multiple instances of Read and Write requests from $\mathcal{O}_c$, for specified blocks kept by the server. These requests to Read and Write blocks kept by the server, should not be confused with access requests (j, α, m), where $\alpha \in \{\mathsf{Read}, \mathsf{Write}\}$, for ORAM cells.

We this far selected Path ORAM as a specific ORAM instantiation. However, any other ORAM scheme is equally applicable, provided that it exhibits an additional property: individual accesses have to be indistinguishable, i.e., the adversary observing just one access request from an access pattern should not be able to tell apart how many accesses the honest user performed so far. We call this property *indistinguishability of individual accesses*, and it is trivially satisfied by Path ORAM. Hierarchical ORAMs (e.g., [12, 16, 19, 20]), however, do not achieve indistinguishability of individual accesses, as the runtime of individual accesses depends on the number of accesses performed so far; in particular the client has to reshuffle periodically a variable amount of data.

Theorem 1. $\mathsf{AnonRAM_{lin}}$ *preserves access privacy in the adversarial model* HbC*, when using a secure ORAM scheme* $\mathcal{O}$ *that satisfies indistinguishability of individual accesses, and a semantically secure re-randomizable encryption scheme* $(\mathsf{E}, \mathsf{R}, \mathsf{D})$.

Proof. Assume to the contrary that some PPT HbC adversary $\mathcal{D}$ can efficiently distinguish, with a non-negligible advantage, between a pair of access-patterns $AP = \{(i_u, j_u, \alpha_u, m_u)\}, AP' = \{(i'_u, j'_u, \alpha'_u, m'_u)\}_{u\in[1,len]}$ of length len.

Let $AP_v = \{(i^*_u, j^*_u, \alpha^*_u, m^*_u)\}_{u\in[1,len]}$ be a 'hybrid' access pattern, where $(i^*_u, j^*_u, \alpha^*_u, m^*_u) = (i_u, j_u, \alpha_u, m_u)$ for $u \leq v$, and $(i^*_u, j^*_u, \alpha^*_u, m^*_u) = (i'_u, j'_u, \alpha'_u, m'_u)$ for $u > v$. In fact, let v be the smallest such value, where some adversary (say $\mathcal{D}$) can distinguish between AP_{v-1} and AP_v, and such $v > 0$ exists by the standard 'hybrid argument' as AP and AP' differ at least in one access.

If $i_v = i'_v$ (i.e., for the same user), the executions only differ in the ORAM client $\mathcal{O}_c$ Read/Write blocks for U_{i_v}; however, this immediately contradicts the privacy of the underlying ORAM scheme. Notice that a user does not decrypt or modify the other users' data during her accesses.

Therefore, assume $i_v \neq i'_v$. Since we expect our ORAM client $\mathcal{O}_c$ to satisfy indistinguishability of individual accesses, the difference between these two patterns are only between the encryption of the blocks output by $\mathcal{O}_c$ and the re-encryption of the blocks received anonymously by the ORAM server. However, ability to distinguish between these, contradicts the indistinguishability property of the semantically secure re-randomizable encryption scheme $(\mathsf{E}, \mathsf{R}, \mathsf{D})$. □

Let c_S and c_B denote the amortized costs of client-side storage and communication complexity of the underlying ORAM protocol. Then, the respective amortized costs of $\mathsf{AnonRAM_{lin}}$ are $N \cdot c_S$ and $N \cdot c_B$. For example, using Path ORAM, the client-side storage and communication complexity costs of $\mathsf{AnonRAM_{lin}}$ become $O(N \log M)$ and $O(N \log^2 M)$.

3.3 $\mathsf{AnonRAM^{M}_{lin}}$ and its Security Against Malicious Users

When some users are malicious, we need to ensure that only a user knowing the private key associated with a block can update the value inside the block, while other users should only be able to re-randomize it. Leveraging the security of $\mathsf{AnonRAM_{lin}}$to the adversarial model of malicious users, we require a semantically secure encryption primitive such that a ciphertext C' can replace a ciphertext C if C' is a re-randomization of C or if the encryptor knows the encryption key for C. Whenever a block is written, the user attaches a zero-knowledge proof showing *either* that the ciphertext is re-encryption of the previous ciphertext, *or* that the user has the (secret) encryption key. The server verifies the proof before updating the block in its RAM memory. This ensures indistinguishability of re-encryption from new encryptions, while ensuring that one user cannot corrupt or modify value of another user. We denote the resulting scheme as $\mathsf{AnonRAM^{M}_{lin}}$.

The required ZK proofs are standard. For the re-randomizable CPA-secure ElGamal encryption scheme, this will involve a disjunction of ZK proof of knowledge of discrete logarithm and ZK proof of equality of the discrete logarithm of

two pairs of group elements. Following the formal notation from [17] and extending it for disjunction, the required proof is

$$PoK\{x_i | pk_i = g^{x_i}\} \ \vee \ P\{\exists r \text{ s.t. } (C_1', C_2') = (C_1^r, C_2^r)\},$$

where PoK stands for proof of knowledge, g is a generator of a group of prime order q, $(pk_i, C = (C_1, C_2), C' = (C_1', C_2'))$ are group elements, and (x_i, r) are elements in $\mathbb{Z}_q$.

Theorem 2. $\mathsf{AnonRAM}_{\mathsf{lin}}^{\mathbf{M}}$ *based on a secure ORAM scheme* $\mathcal{O}$ *that satisfies indistinguishability of individual accesses, CPA-secure public-key encryption scheme (e.g., ElGamal), and a disjunction ZK proof defined above, preserves integrity and privacy in the adversarial model* $\mathsf{Mal\_Users}$.

Proof Sketch. The integrity argument is simple: use of the disjunction ZK proof ensures that the adversarial users cannot modify the cell of other honest users. The adversarial users also cannot change the order of cells in a sequence of cells as the server verifies correctness of one cell at a time. They can only re-randomize the cells of the honest users.

The privacy properties are also preserved similarly to $\mathsf{AnonRAM}_{\mathsf{lin}}$ as the disjunctive nature of the included ZK proof does not allow the server to determine which of N cells is modified by an honest user. While privacy of the accessed cell-index as well as the access type is maintained by the employed ORAM scheme. □

4 Polylogarithmic-complexity AnonRAM

The $\mathsf{AnonRAM}_{\mathsf{lin}}$ scheme exhibits acceptable performance for small number of users, but linear overhead renders it prohibitively expensive as the number of users increases. In this section, we present $\mathsf{AnonRAM}_{\mathsf{polylog}}$, an AnonRAM scheme whose overhead is poly-logarithmic in the number of users.

$\mathsf{AnonRAM}_{\mathsf{polylog}}$ is conceptually based on the hierarchical Goldreich-Ostrovsky ORAM (GO-ORAM) construction [12], where a user periodically reshuffles her cells maintained over a *storage server* S. To reshuffle together cells belonging to multiple users, we introduce in $\mathsf{AnonRAM}_{\mathsf{polylog}}$ an additional server, the so-called *tag server* T. The tag server reshuffles data on the users' behalf, without knowing the data elements, and thereby maintains a user privacy from the storage server S as well as from the other users. The tag server only requires constant-size storage to perform this reshuffling, and we show that, similarly to the storage server, it cannot violate (on its own or with colluding users) the privacy requirements of AnonRAM schemes.[6]

In what follows, we first describe the employed cryptographic tools, and then present the $\mathsf{AnonRAM}_{\mathsf{polylog}}$ construction and its complexity and security analysis, first for the honest-but-curious case, and after that its extension, $\mathsf{AnonRAM}_{\mathsf{polylog}}^{\mathbf{M}}$, to cope with malicious users.

[6] Adhering to our adversarial model from Section 2, we only consider the corruption of a single server, and hence assume non-colluding servers S and T.

4.1 Cryptographic Building Blocks

Universally Re-randomizable Encryption. A universally re-randomizable encryption (UREnc) scheme [13, 25] allows to re-randomize given ciphertexts without requiring access to the encryption key. We use the construction of Golle et al. [13]: For a generator g of a multiplicative group G_q of prime order q and a private/public key pair (x_i, g^{x_i}) for party i with $x_i \in \mathbb{Z}_q^*$, the encryption $C = \mathsf{E}_i^*(m)$ of a message m is computed as an El-Gamal encryption of m together with an El-Gamal encryption of the identity $1 \in G_q$; i.e., $C = (g^a, g^{ax_i} \cdot m, g^b, g^{bx_i} \cdot 1)$ for $a, b \in \mathbb{Z}_q^*$. The ciphertext C can be re-randomized, denoted $\mathsf{R}^*(C)$ by selecting $a', b' \leftarrow_R \mathbb{Z}_q^*$ and outputting $(g^a \cdot (g^b)^{a'}, g^{ax_i} \cdot m \cdot (g^{bx_i})^{a'}, (g^b)^{b'}, (g^{bx_i})^{b'})$ as the new ciphertext. Note that this scheme is also multiplicatively homomorphic.

We employ a distributed version of the UREnc scheme, where the private key is shared between two servers such that both have to be involved in decryption.

(Partially Key-Homomorphic) Oblivious PRF. An oblivious pseudo-random function (OPRF) [11, 17] enables a party holding an input tag μ to obtain an output $f_s(\mu)$ of a PRF $f_s(\cdot)$ from another party holding the key s without the latter party learning any information about the input tag μ.

We use the Jarecki-Liu OPRF construction [17] as our starting point. Here, the underlying PRF $f_s(\cdot)$ is a variant of the Dodis-Yampolskiy PRF construction [9] such that $f_s(\mu) := \mathsf{g}^{1/(s+\mu)}$ is defined over a composite-order group of order $n = p_1 p_2$ for safe primes p_1 and p_2. This function constitutes a PRF if factoring safe RSA moduli is hard and the Decisional q-Diffie-Hellman Inversion assumption holds on a suitable group family G_n [17].

To securely realize pre-tag randomization in our $\mathsf{Reshuffle}$ algorithm (explained later), we propose a modification of the Jarecki-Liu OPRF where a second key $\hat{s}$ is used to define a new PRF $f_{s,\hat{s}}(\mu) := \mathsf{g}^{\hat{s}/(s+\mu)}$. We call such a PRF *partially key-homomorphic* as $(f_{s,\hat{s}}(\mu))^{\delta} = f_{s,(\hat{s}\cdot\delta)}(\mu)$ holds for it. For unlinkability of PRF values of the same input μ with updated δ, we expect the Composite DDH assumption [7] [3] to hold in G_n. We denote our OPRF construction as $\mathsf{OPRF}_{s,\hat{s}}^{\mathcal{A},\mathcal{B}}(\mu)$, where $\mathcal{A}$ denotes a party with input μ, and $\mathcal{B}$ denotes a server possessing the keys s and $\hat{s}$. Our OPRF protocol makes only minor changes to the Jarecki-Liu OPRF, and we postpone its full description and security analysis to Appendix A.

Additively Homomorphic Encryption. For appropriately computing on our OPRF outputs, we need a suitable additively homomorphic encryption scheme whose decryption is shared between our two servers. Similarly to our OPRF construction, we employ a semantically-secure variant [17] of the Camenisch-Shoup encryption (CSEnc) [2] scheme. With its roots in Paillier encryption [23], the CSEnc scheme is secure in the common reference-string model under the composite decisional residuosity assumption [23]. The scheme is defined in the safe RSA moduli ($n = p_1 p_2$) setting with plaintext messages belonging to $\mathbb{Z}_n$. Here, an encryption $\mathsf{E}_{\mathsf{pk}}^+(m)$ denotes a message $m \in \mathbb{Z}_n$ encrypted under a public

[7] Composite Decisional Diffie-Hellman assumption [3] is a variant of the standard DDH assumption [1], but defined over a composite order group.

key $\mathsf{pk} = \mathsf{g}^x$, where g is a generator of group G_n of size n, and the private key x belongs to $\mathbb{Z}^*_{n/4}$. CSEnc is an additively homomorphic encryption scheme such that $\mathsf{E}^+_{\mathsf{pk}}(m)\cdot\mathsf{E}^+_{\mathsf{pk}}(m') = \mathsf{E}^+_{\mathsf{pk}}(m+m')$ for any $m, m' \in \mathbb{Z}_n$, and $\mathsf{E}^+_{\mathsf{pk}}(m)^\delta = \mathsf{E}^+_{\mathsf{pk}}(m\cdot\delta)$ for any $\delta \in \mathbb{Z}^*_n$. This scheme moreover allows shared decryption; i.e., given public/private keys pairs $(\mathsf{pk}_\mathcal{A}, \mathsf{sk}_\mathcal{A})$ and $(\mathsf{pk}_\mathcal{B}, \mathsf{sk}_\mathcal{B})$ of parties $\mathcal{A}$ and $\mathcal{B}$ and the joint public key $\mathsf{pk} = \mathsf{pk}_\mathcal{A} \cdot \mathsf{pk}_\mathcal{B}$, parties $\mathcal{A}$ and $\mathcal{B}$ can jointly decrypt a ciphertext $\mathsf{E}^+_{\mathsf{pk}}(m)$ for a receiver using their private keys $\mathsf{sk}_\mathcal{A}$ and $\mathsf{sk}_\mathcal{B}$. In our construction, given a ciphertext encrypted under the joint public key of servers S and T, they jointly decrypt the ciphertext such that the plaintext message is available to T.

Oblivious Sort. In Oblivious Sort (OSort), one party (in our case, S) holds an encrypted data array, and the other party (T) operates on the data array, such that the data array becomes sorted according to some comparison criteria, and S learns nothing about the array (therefore, the name "oblivious" sort). OSort can be instantiated by the recently introduced *randomized ShellSort* algorithm [14], which runs in $O(z\log(z))$ for z elements.

4.2 $\mathsf{AnonRAM}_{\mathsf{polylog}}$ Data Structure

$\mathsf{AnonRAM}_{\mathsf{polylog}}$ caters N independent users $(\mathsf{U}_1, \ldots, \mathsf{U}_N)$ with their $M \cdot N$ cells (i.e., M cells per user) using a storage server S and a tag server T. Similarly to other hierarchical schemes [12, 19, 20, 24], blocks are organized in $L = \lceil\log(M \cdot N)\rceil + 1$ levels, where each level $\ell \in [1, L]$ contents 2^ℓ buckets. Each bucket contains $\beta := \lceil c_\beta \log(M \cdot N)\rceil$ blocks, where c_β is a (small) constant.

Similarly to GO-ORAM, during each access, the user reads a pseudo-randomly chosen (entire) bucket from each level such that server S cannot learn anything by observing the bucket access patterns. $\mathsf{AnonRAM}_{\mathsf{polylog}}$ adopts a recent improvement to GO-ORAM proposed in [19, 20, 24] to avoid duplicate user blocks in the server-side (RAM) storage at any point in time. To achieve this, on every access, the user has to write a 'dummy' block into the location where it finds the data such that S cannot distinguish between the added 'dummy' block and the 'real' data block. These user-added dummy blocks are periodically removed to avoid RAM memory expansion, and the rest of the blocks are periodically reshuffled to allow users to access the same cell multiple times.

In existing single-user single-server GO-ORAM designs [12,19,24], this reshuffling is performed by the user. In $\mathsf{AnonRAM}_{\mathsf{polylog}}$, reshuffling operations involve blocks of different users, and it cannot be performed by one or more users without interacting with all other users. As we want to avoid interaction among the users, reshuffling in $\mathsf{AnonRAM}_{\mathsf{polylog}}$ is jointly performed by two *non-colluding* servers (the storage-server S and the tag-server T) without exposing a user's data or access pattern to either server.

Blocks Types. Each block in $\mathsf{AnonRAM}_{\mathsf{polylog}}$ consists of two parts: a CSEnc-encrypted OPRF output called *pre-tag* part and a UREnc-encrypted *value* part. We consider three types of blocks: *real*, *empty*, and *dummy* blocks.

A *real* block is of the form $\langle \mathsf{E}^+_{\mathsf{TS}}(\theta_i), \mathsf{E}^*_{\mathsf{U}_i}(j, m_{i,j})\rangle$. Here, the value part contains the j^{th} cell of user U_i with value $m_{i,j}$ encrypted with UREnc for U_i, while

the pre-tag part contains a pre-tag θ_i computed using OPRF for some secret input of U_i and encrypted using CSEnc for a joint public key of T and S. The pre-tag θ_i is computed by U_i with help from the storage server S using OPRF, and it is used to map the block to a particular bucket in the given level. Given a pre-tag θ, for a level $\ell \in [1, L]$, the bucket index (or tag) is computed by applying a random oracle hash function, $h_\ell : \{0,1\}^* \to \mathbb{Z}_{2^\ell}$. The mapping changes after 2^ℓ accesses, which we refer to as an *epoch*.

Empty blocks are padding blocks that are used to form buckets of the required size β on the storage server S. An empty block is of the form $\langle \mathsf{E}^+_{\mathsf{TS}}(0), \mathsf{E}^*_{\mathsf{TS}}(\text{“}empty\text{”})\rangle$, where “*empty*” is a constant in the UREnc message space. An empty block will be encrypted similarly to other types of blocks to ensure the privacy against the storage server S, and the server should not be able to determine whether a user fetched an empty block or a real block. The first part of empty block is an encryption of unity 0 as it allows the tag server T to determine if a block is empty during the reshuffle.

Finally, similarly to most ORAM algorithms, we use *dummy* blocks to hide locations of the real blocks. Once a real block with a specific index is found at some level, it is moved to a new bucket at the first level and is replaced with a dummy block in its old location. A dummy block is of the form $\langle \mathsf{E}^+_{\mathsf{TS}}(\theta_{\mathcal{D}}), \mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”})\rangle$, where the pre-tag $\theta_{\mathcal{D}}$ is computed using OPRF on the number (t) of accesses made by the users so far and a secret input $\mu_{\mathcal{D}}$ known only to server T, and “*dummy*” is a constant in the UREnc message space.

Note that different blocks are completely indistinguishable to non-colluding servers S and T individually. Nevertheless, during the reshuffle operations, when necessary, with help from server S, server T can determine type of a block.

4.3 $\mathsf{AnonRAM}_{\mathsf{polylog}}$ Protocol Overview

Initialization. We need to initialize UREnc, CSEnc, and OPRF. For the security parameter λ, we choose a multiplicative group G_q of an appropriate prime order q for UREnc, and a multiplicative group G_n of order equal to an appropriate safe RSA modulus n for CSEnc and OPRF. Let g and g be generators of groups G_q and G_n respectively.

Given this setup, every user generates her UREnc key from $\mathbb{Z}^*_q$. The two servers select their individual shared private keys for both UREnc and CSEnc, and publish the corresponding combined public key for CSEnc; we do not need UREnc public key for two servers. We represent these encryptions as follows: $\mathsf{E}^*_{\mathsf{U}_i}(\cdot)$ represents a UREnc encryption for user U_i; $\mathsf{E}^*_{\mathsf{TS}}(\cdot)$ and $\mathsf{E}^+_{\mathsf{TS}}(\cdot)$ respectively represent shared UREnc and CSEnc encryptions for the servers S and T. The servers make an encrypted empty block $\mathsf{E}^*_{\mathsf{TS}}(\text{“}empty\text{”})$ and an encrypted dummy block $\mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”})$ public to all users.

Similarly to all existing hierarchical ORAM constructions, all levels in the $\mathsf{AnonRAM}_{\mathsf{polylog}}$ data structure on S are initially empty. In particular, the complete first level is filled up with empty blocks, while the rest of the levels are not yet allocated. The users write $M \cdot N$ cells initialized to some default value,

one by one, at the first level such that at the end of the initialization procedure, $M \cdot N$ users' cells will be stored at level L and the remaining levels will be empty (w.l.o.g. we assume that $M \cdot N$ is power of 2). Let t denote the access counter, which is made available publicly by the servers. Each level ℓ has an epoch counter $\xi(t, \ell)$ that increments after every $2^{\ell-1}$ accesses. In other words, for level ℓ and t accesses, the epoch counter is $\xi(t, \ell) = \lfloor t/2^{\ell-1} \rfloor$.

Recall that our OPRF employs two keys. S generates the first (and fixed) OPRF key $s \leftarrow_R \mathbb{Z}_n^*$, and then a series of second OPRF keys $\hat{s}[\ell, \xi(t, \ell)] \leftarrow_R \mathbb{Z}_n^*$, for each level $\ell \in [1, L]$ and the current access t. A user U_i generates independently a secret PRF input $\mu_i \in \mathbb{Z}_n$, and computes a pre-tag θ for her block j using μ_i by performing OPRF with S. Similarly, the tag server T generates a secret input $\mu_{\mathcal{D}}$ for dummy blocks. To tag blocks, the construction uses a hash function family $\{h_\ell\}$ domain $[0, 2^\ell - 1]$, for each level $\ell \in [1, L]$. In particular, a tag (or bucket index) for a pre-tag θ is computed as $h_\ell(\xi(t, \ell)||\theta)$, where $||$ represents string concatenation.

Protocol Flow. Similarly to our constructions in Section 3, users have to communicate with the servers via anonymous channels. To access a cell j during the t^{th} access, user U_i first computes the associated pre-tags for all levels θ_i using OPRF with S on her secret inputs μ_i and j. She also obtains $\theta_{\mathcal{D}}$ pre-tags from server T for all levels for the current value of access counter t. Here, T computes pre-tags for dummy blocks by interacting with S and sends those to the users as the users cannot locally compute them. These pseudorandom pre-tag values depend on the level and the current epoch through the PRF keys used by S. Due to the oblivious nature of OPRF and secret inputs μ_i for U_i and $\mu_{\mathcal{D}}$ for T, server S does not learn the pre-tag values.

Once pre-tags are computed, the user maps each of those to a bucket index (or *tag*) in their level ℓ using h_ℓ. Now, she starts searching for her cell j from level 1 using tags computed using a pre-tag θ_i. Similarly to other hierarchical schemes, after obtaining her cell, she searches for the rest of the levels with tags computed using $\theta_{\mathcal{D}}$ values. The updated cell j is added back to the level 1. During this process, a pre-tag θ associated with the user's cell changes to another value θ' indistinguishable from random. Fig. 2 shows the main sub-flow of User algorithm executed by U_i in cooperation with servers S and T. In User flow, this sub-flow is repeated once for each level. Finally, at the end of User, the user computes a new pre-tag for possibly updated cell j, and computes and stores a block with them at the first level.

Although dummy pre-tags and tags are computed by and known to T, it cannot learn a tag employed by a user while requesting blocks from S, as communication between the user and server S is encrypted. T cannot learn this information based on the content of blocks of specific tags retrieved by observed users, since S *re-randomizes* blocks before sending them to users.

The main task of T is to reshuffle the blocks *without* involving the users. In the $\mathsf{Reshuffle}$ protocol, while reshuffling levels 1 to ℓ into level $\ell + 1$, server T copies, re-randomizes or changes blocks from levels 1 to ℓ, and then sorts them using oblivious-sorting (OSort) such that the users can obtain their required cells

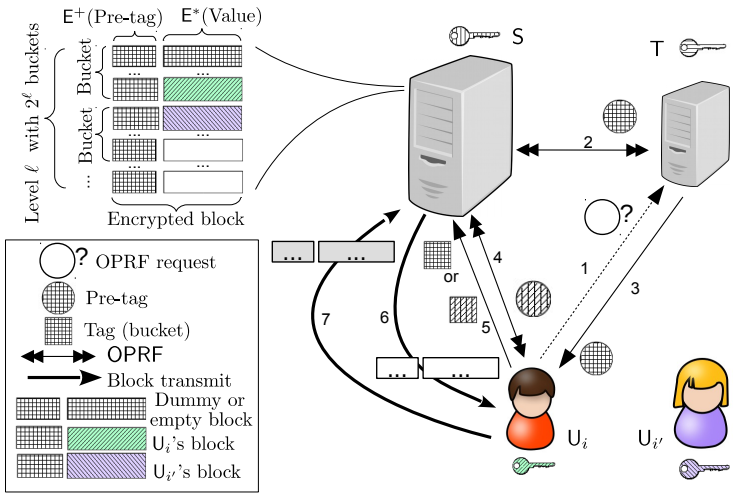

Fig. 2. Flow of User algorithm in $\mathsf{AnonRAM}_{\mathsf{polylog}}$ for user U_i, cell j, and level ℓ: 1) U_i asks the tag server T for a dummy pre-tag. 2) T runs an OPRF protocol with the storage server S, such that T learns the dummy pre-tag, and S learns nothing. 3) T sends the dummy pre-tag to U_i. 4) U_i runs OPRF with S to learn a pre-tag for her cell j obliviously. 5) Depending on whether cell j is found in the previous levels or not, U_i selects one of the two pre-tags to compute a tag and sends the tag to S. 6) S re-randomizes and sends the block(s) associated with the user's tag. 7) U_i re-randomizes or updates the block(s), and possibly learns the value of cell j. If $\ell = 1$, steps 4) and 5) are skipped, and in step 6) S sends all blocks from that level.

over level $\ell + 1$ by procuring the appropriate pre-tag values from server S. This step requires server S helping server T to decrypt randomized version of pre-tags in blocks. Here, for every second access, T performs reshuffle of level 1 into level 2 on S, to empty level 1. For every fourth access, all the real blocks at levels 1 and 2 will be moved to level 3, and so on.

The crucial property is that, while reshuffling, server T should not learn any information about user's data from pre-tags. To prevent T from identifying users' cells by pre-tags, S proactively shuffles all blocks that T will access during Reshuffle and updates the pre-tags associated with the blocks. Here, S utilizes homomorphic properties of OPRF: in particular, for some pre-tag $\theta = f_{s,\hat{s}}(\mu)$ for server S's OPRF keys $s, \hat{s}$, the server computes $\theta^{\delta} = f_{s,(\hat{s}\cdot\delta)}(\mu)$ for some random δ. Although pre-tags in the blocks are stored in the encrypted form and cannot be decrypted by S alone, the homomorphic properties of CSEnc allow S to apply the aforementioned trick to ciphertexts *without* knowing pre-tags in plain. Finally, S partially decrypts the pre-tags of the blocks that have to be reshuffled by T and moves these blocks to a temporary array.

After the pre-processing by server S, server T decrypts pre-tags of the blocks and reshuffles non-empty blocks to arrange them into buckets based on pre-tags. This process is essentially the same as the Oblivious-Hash step in GO-ORAM [12] except for de-duplication of blocks [24]. Specifically, while reshuffling blocks from levels 1 to ℓ into level $\ell + 1$, T first adds 2^{ℓ} *forward* dummy blocks that can potentially be accessed by a user in subsequent accesses. It then assigns tags

to non-empty blocks using hash function $h_{\ell+1}$ and ensures that no tag gets assigned to more than β blocks. Finally, T pads the temporary array with the tagged empty blocks such that exactly β blocks have the same tag, replaces forward dummy blocks with empty ones, and moves all these blocks to level $\ell+1$ on server S. Here, T cannot link the pre-tags seen in a current Reshuffle execution to those observed during previous reshuffles as the value δ chosen by S is unknown to T.

4.4 User Algorithm

In the User algorithm, a user searches in all levels for a block containing her cell j. Once the block is found, it is moved to a new location at the first level after a possible update (in case of Write operation), and a dummy block is instead added to the old location.

User algorithm for user U_i, on input (j, α, m) consists of the following steps:

1. Allocate local space to hold a single encrypted block value res and initialize it to $\mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”})$. Set boolean variable **found** to $false$.

2. Receive from T the pre-tag $\theta_{\mathcal{D}} := \mathsf{OPRF}^{\mathsf{T},\mathsf{S}}_{\hat{s}[1,\xi(t,1)]}(\mu_{\mathcal{D}} + t)$ computed by T by performing OPRF with S. Read all blocks at level 1. Let B denote a current block at level 1, *re-randomized* and sent by S to U_i. Parse block B into its two components (B_1, B_2), where the first part is a CSEnc ciphertext and the second part is a UREnc ciphertext. User U_i deciphers B_2 using her UREnc private key. If block with cell index j is found, then the user sets **found** to *true*, copies B_2 to res, and replaces B with a dummy block $\langle \mathsf{E}^+_{\mathsf{TS}}(\theta_{\mathcal{D}}), \mathsf{R}^*(\mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”}))\rangle$. Otherwise, U_i replaces the block B with its re-randomized version.

3. For each level ℓ from 2 to L:

(a) Compute a pre-tag $\theta_i \leftarrow \mathsf{OPRF}^{\mathsf{U}_i,\mathsf{S}}_{\hat{s}[\ell,\xi(t,\ell)]}(\mu_i + j)$ by interacting with S, and receive a pre-tag $\theta_{\mathcal{D}} \leftarrow \mathsf{OPRF}^{\mathsf{T},\mathsf{S}}_{\hat{s}[\ell,\xi(t,\ell)]}(\mu_{\mathcal{D}}+t)$ from T. If **found**, then compute tag $\tau := h_\ell(\xi(t,\ell)||\theta_i)$, else $\tau := h_\ell(\xi(t,\ell)||\theta_{\mathcal{D}})$.

(b) Read all blocks of bucket τ at level ℓ. Let B denote a current block of bucket τ at level ℓ, *re-randomized* and sent by S to U_i. If B is U_i's real block with index j, the user sets **found** to *true*, copies the value of B to res, and replaces B with a dummy block $\langle \mathsf{E}^+_{\mathsf{TS}}(\theta_{\mathcal{D}}), \mathsf{R}^*(\mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”}))\rangle$. Otherwise, i.e. if B is not U_i's real block with index j, then U_i replaces block B with a re-randomized version of B.

4. If $\alpha = \mathsf{Write}$, update res to the new value $\mathsf{E}^*_{\mathsf{U}_i}(m)$.

5. Re-randomize and send $\langle \mathsf{E}^+_{\mathsf{TS}}(\theta_i), res\rangle$ to S, where $\theta_i := \mathsf{OPRF}^{\mathsf{U}_i,\mathsf{S}}_{\hat{s}[1,\xi(t,\ell)]}(\mu_i + j)$. Server S writes the block to the first available empty slot at level 1.

6. If $\alpha = \mathsf{Read}$, return res.

4.5 Reshuffle

Reshuffle of every level ℓ into a higher level, is performed every 2^ℓ accesses, when the number of non-empty blocks (real or dummy) at level ℓ reaches 2^ℓ. We reshuffle *all* blocks, from levels 1 to ℓ into level $\ell+1$, and there are no duplicates.

Recall that after User, the number of non-empty blocks at the first level increases by 1. After two accesses, all blocks from the first level will be reshuffled into the second level. After two more accesses, all blocks from first level should be reshuffled into the second level. This event triggers the reshuffle of all blocks from the first two levels into third level. After this point, there will be four non-empty blocks at level 3, and levels 1 and 2 will be empty.

For the current value of t, let $\ell_m := \max\{\ell > 0 \text{ s.t. } 2^\ell | t\}$. Then, *before* the reshuffle is performed, level 1 has two non-empty blocks, and each level $\ell \in [2, \ell_m]$ has $2^{\ell-1}$ non-empty blocks. As we show in Lemma 2 later, level $\ell_m + 1$ is empty. Hence, the $\mathsf{Reshuffle}$ procedure takes all elements from levels 1 up to ℓ_m and moves them into level $\ell_m + 1$. The total number of non-empty blocks in levels 1 to ℓ_m is $2 + \sum_{\ell=2}^{\ell_m} 2^{\ell-1} = 2^{\ell_m}$, so 2^{ℓ_m} real or dummy blocks will be added into level $\ell_m + 1$. The array is sparse; there are $(2^{\ell_m+1} - 2)\beta$ blocks at levels 1 to ℓ_m including empty ones, and among them only 2^{ℓ_m} dummy or real ones. A $\mathsf{Reshuffle}$ protocol between S and T requires two operations performed by T on data stored at S: Scan and OSort.

A generic $\mathsf{Reshuffle}$ algorithm for levels 1 up to ℓ_m into level $\ell_m + 1$ is given below; steps 1-3 are performed by S, and remaining steps 4-13 by T:

1. S allocates space for a temporary array A to hold $2^{\ell_m+2} \cdot \beta$ blocks.
2. For each level ℓ from 1 to ℓ_m:

(a) S moves all $2^\ell \cdot \beta$ blocks from level ℓ into a temporary array A', and fills the level ℓ with empty blocks. Each new empty block is just a re-randomization of the public empty block $\mathsf{E}^*_{\mathsf{TS}}(\text{“}empty\text{”})$.

(b) Let C denote the encrypted pre-tag of some block in A', and $C = \mathsf{E}^+_{\mathsf{TS}}\big(\theta_{\hat{s}[\ell,\xi(t,\ell)-1]}\big)$, where $\theta_{\hat{s}}$ denotes the value of PRF $f_{s,\hat{s}}(\mu)$ for some input $\mu \in \mathbb{Z}_n$. For each block in A', S replaces C with $C' \leftarrow C^{\hat{s}[\ell+1,\xi(t,\ell+1)]/\hat{s}[\ell,\xi(t,\ell)-1]}$. Thanks to the properties f and homomorphic properties of CSEnc, $C' = \mathsf{E}^+_{\mathsf{TS}}\big(\theta_{\hat{s}[\ell,\xi(t,\ell+1)]}\big)$.

(c) S moves all blocks from array A' to array A.

3. S pads A with empty blocks and partially decrypts pre-tags of all blocks. Finally, S permutes A.

4. *Decrypting pre-tags*: Decrypt pre-tags using partial decryption of S and attach pre-tags to the blocks using (fast) local encryption scheme.

5. *Adding forward dummies:* Add 2^{ℓ_m} dummy blocks $\langle \mathsf{E}^+_{\mathsf{TS}}(\theta_0), \mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”})\rangle$, $\ldots$, $\langle \mathsf{E}^+_{\mathsf{TS}}(\theta_{2^{\ell_m}-1}), \mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”})\rangle$, where T computes pre-tags in interaction with S as $\theta_k := \mathsf{OPRF}^{\mathsf{T},\mathsf{S}}_{\hat{s}[\ell_m,\xi(t+k,\ell_m+1)]}(\ \mu_{\mathcal{D}} + (t+k))$, and for $k \in [0, 2^{\ell_m} - 1]$. Array A now contains 2^{ℓ_m+1} non-empty blocks. Scan A and add temporary encryption for flag to the blocks: value 1 to the forward dummy blocks, otherwise value 0.

6. *Mapping to buckets*: Scan A and use hash function h_{ℓ_m+1} to assign tags to non-empty blocks. Since there are no duplicates, each non-empty block has (with overwhelming probability) a unique input to the hash function. Specifically, T attaches to a block the tag $h_{\ell+1}(\xi(t, \ell_m + 1)||\theta)$, where θ denotes the pre-tag of that block.

7. Obliviously-sort A, using the OSort protocol, according to the following criteria: (a) non-empty blocks before empty ones, (b) lower tags first.

8. *Checking if there is no bucket overflow*: Scan A and check that no single tag was given to more than β blocks. If there is such a tag, this is considered as overflow. An overflow can happen only with low probability; in this case, choose another hash-family $h_{\ell+1}$, and go to step 6.

9. Scan A and assign tags to $2^{\ell_m+1}\beta$ *untagged* empty blocks, one tag per β blocks. This steps ensures that each tag $0, \ldots, 2^{\ell_m+1} - 1$ is represented by *at least* β blocks.

10. OSort A according to the following criteria: (a) tagged blocks before untagged, (b) lower tags first, (c) non-empty blocks before empty ones (among blocks with the same tag).

11. Scan A and make sure that exactly β blocks have the same tag, erasing excessive blocks. Note that all excessive blocks are empty.

12. *Prepare buckets for level* $(\ell_m + 1)$: OSort A according to the following criteria: (a) tagged blocks before untagged, (b) lower tags first.

13. Scan A to replace the dummy blocks introduced in Step 5 with empty ones; these blocks have the value of encrypted flag 1. Scan A to erase tags, temporary encryptions of pre-tags (attached to the blocks in Step 4), and flags for forward dummy blocks. Move the $2^{\ell_m+1}\beta$ prefix of A into level $\ell_m + 1$, one by one in the same order as the blocks appear in A.

The result of Reshuffle is $2^{\ell_m+1}\beta$ blocks is stored at level $\ell_m + 1$. The first β blocks have tag 0, next β blocks tag 1, etc. Since the storage at server S organizes in buckets of size β, this layout corresponds to putting β blocks with same tag to one bucket.

Reshuffle algorithm ensures, using forward dummy blocks, that buckets do not overflow in the follow-up accesses made by users until the next reshuffle. Specifically, there are no buckets accessed more than β times at some level ℓ within one epoch. Note that if instead of accessing dummy blocks, a user chose random buckets, this could lead to a small, but *not* negligible probability of distinguishing specific access patterns.

Last Reshuffle. After reshuffle into level $L+1$, $M \cdot N$ real and $M \cdot N$ dummy blocks are located at that level. T and S eliminate dummy blocks by jointly decrypting the block value: if a block is dummy, the decryption succeeds. Finally, $M \cdot N$ real blocks from level $L+1$ are reshuffled into level L, thus achieving the state after the initial setup.

4.6 Complexity and Security Analysis

Computational and communication complexity of User is $O(\log^2(M \cdot N))$ since there are $L = \log(M \cdot N)$ levels, and for each level a user performs $\beta = O(\log(M \cdot N))$ encryptions, decryptions, and OPRF evaluations. Each of these operations requires $O(1)$ exponentiations.

Computational and communication complexity of Reshuffle depends on parameter t. Consider the state after Setup, and the state after $M \cdot N$ subsequent

accesses. They are identical, as all the real blocks are located at level L. Hence it suffices to analyze the aforementioned interval. Let $\rho(\ell)$ denote the complexity of $\mathsf{Reshuffle}(\ell)$ denoted the reshuffle from levels 1 to ℓ into level $\ell+1$. In $\mathsf{Reshuffle}(\ell)$, the number of blocks involved is $2^{\ell+1}\beta$, hence $\rho(\ell) = O(2^{\ell+1} \cdot \beta \cdot \log(2^{\ell+1} \cdot \beta))$ due to the cost of OSort. Then, within $M \cdot N$ accesses, there is one $\mathsf{Reshuffle}(L)$, none of $\mathsf{Reshuffle}(L-1)$ (since level L initially already contains $M \cdot N$ elements), two $\mathsf{Reshuffle}(L-2)$, four $\mathsf{Reshuffle}(L-3)$, etc. Thus, the total complexity of all reshuffles made within $M \cdot N$ accesses is $(\sum_{\ell=1}^{L-2} 2^{L-2-\ell} \cdot \rho(\ell)) + \rho(L) = (\sum_{\ell=1}^{L-2} 2^{\ell-1} \cdot O(2^{\ell+1} \cdot \beta \cdot \log(2^{\ell+1} \cdot \beta))) + O(2^{L+1} \cdot \beta \cdot \log(2^{L+1} \cdot \beta)) = O(\beta \cdot \sum_{\ell=1}^{L-2} 2^{2\ell} \cdot \log(2^{\ell+1}))$ $= O(\beta \cdot 2^L \cdot L \cdot \log L) = O(M \cdot N \cdot \log^2(M \cdot N) \cdot \log\log(M \cdot N))$. Hence, the amortized cost of $\mathsf{Reshuffle}$ is $\tilde{O}(\log^2(M \cdot N) \cdot \log\log(M \cdot N))$.

Theorem 3. $\mathsf{AnonRAM}_{\mathsf{polylog}}$ *preserves access privacy against HbC adversaries in the random oracle model, when instantiated with semantically secure universally re-randomizable encryption (UREnc) and additively homomorphic encryption schemes, and a secure (partially key-homomorphic) oblivious PRF scheme for appropriate compatible domains.*

The proof of Theorem 3 and subsequent proofs are postponed to the appendix.

4.7 $\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{polylog}}$ Secure Against Malicious Users

For $\mathsf{AnonRAM}_{\mathsf{polylog}}$, we need the users to satisfy some more conditions to avoid any tampering by the malicious users. Integrity and privacy of an honest user U_i can be achieved if a malicious user can neither change existing real entries of other users, nor introduce new real blocks encrypted using U_i's key. We present a modification of $\mathsf{AnonRAM}_{\mathsf{polylog}}$ ($\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{polylog}}$) secure against malicious users.

Without loss of generality and observing that blocks are written to the first level in a pre-defined manner, we may assume that User for tuple (i, j, α, m) parametrized with t does the following:

1. User U_i reads (t mod 2) blocks at level 1, and β blocks at each level $\ell \in [2, L]$.
2. Among all these blocks there is at least one block belonging to U_i, and exactly one block matches the target index j.
3. User U_i replaces one of the blocks that belongs to her, with the dummy block.
4. The replaced above real block is moved to a new pre-defined location at level 1 (replacing an empty block).

It is important to enforce both 3) and 4), since otherwise a malicious user could introduce a new real block of an unobserved user, thus violating integrity of honest users. We elaborate on appropriate zero-knowledge proofs for UREnc and CSEnc.

Recall that ZK proof system required in $\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{lin}}$ for a pair of old and new ciphertexts, ensures that the new ciphertext is either a re-randomization of the old ciphertext, or the user knows the associated with the old ciphertext

secret key. Re-randomization of ciphertext can be split into two parts: ciphertexts are encrypted under the same key, and encoded messages are the same. In $\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{polylog}}$, to achieve integrity we require also the former component, i.e. proof that two ciphertexts are encrypted under the same key.

Summarizing, we have the following types of proof systems w.r.t. a single ciphertext $C = (C_1, \ldots, C_4)$, or relations between two ciphertexts C and C':

- the user knows the decryption key of UREnc ciphertext C; it requires ZK proof of knowledge of discrete logarithm $PoK\{x_i | C_4 = C_3^{x_i}\}$.
- ciphertext C' is encrypted under the same key as ciphertext C; the required proof is ZK proof of equality of the discrete logarithm of two pairs of group elements $P\{\exists r \text{ s.t. } (C'_3, C'_4) = (C_3^r, C_4^r)\}$.
- ciphertext C' is a re-randomization of ciphertext C; the required proof is conjunction of two ZK proofs of equality of the discrete logarithm of two pairs of group elements if C is UREnc ciphertext, or just one such ZK proof if C is CSEnc ciphertext (in this case $C = (C_1, C_2)$).

We refer to Appendix C for the detailed description of necessary changes in $\mathsf{AnonRAM}_{\mathsf{polylog}}$ in order to obtain $\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{polylog}}$ secure against malicious users.

Theorem 4. $\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{polylog}}$ *in the random oracle model, when instantiated with semantically secure universally re-randomizable encryption (UREnc) and additively homomorphic encryption schemes, a secure (partially key-homomorphic) oblivious PRF scheme for appropriate compatible domains, and augmented with ZK proof system defined above, preserves integrity and privacy in the adversarial model* $\mathsf{Mal\_Users}$.

5 Conclusion

We have defined the concept of *anonymous RAM (AnonRAM)*, and presented provably secure constructions. AnonRAM simultaneously provides privacy of content, access patterns and the user identities, while additionally ensuring the integrity of the user's data. It hence constitutes a natural extension of the concept of oblivious RAM (ORAM) to a domain with multiple, mutually distrusting users. Our first construction exhibits an access complexity linear in the number of users, while the second one improves the complexity to an amortized access cost that is polylogarithmic in the total number of cells of all users, at the cost of requiring two non-colluding servers. Both constructions have a simpler version which assumes honest-but-curious users, but also a version secure against malicious users. Many open challenges remain, in particular, polylogarithmic access complexity using a single server.

References

1. Boneh, D.: The decision diffie-hellman problem. In: Algorithmic number theory, pp. 48–63. Springer (1998)

2. Camenisch, J., Shoup, V.: Practical verifiable encryption and decryption of discrete logarithms. In: Advances in Cryptology-CRYPTO, pp. 126–144. Springer (2003)
3. Catalano, D., Gennaro, R.: New efficient and secure protocols for verifiable signature sharing and other applications. In: Advances in Cryptology–CRYPTO. pp. 105–120 (1998)
4. Chaum, D.: Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM 4(2), 84–88 (1981)
5. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1(1), 65–75 (1988)
6. Chung, K., Liu, Z., Pass, R.: Statistically-secure ORAM with $\tilde{o}(\log^2 n)$ overhead. In: Advances in Cryptology–ASIACRYPT. pp. 62–81 (2014)
7. Danezis, G., Dingledine, R., Mathewson, N.: Mixminion: Design of a Type III anonymous remailer protocol. In: Security and Privacy (S&P). pp. 2–15 (2003)
8. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. In: Usenix Security. pp. 303–320 (2004)
9. Dodis, Y., Yampolskiy, A.: A verifiable random function with short proofs and keys. In: Public Key Cryptography-PKC, pp. 416–431. Springer (2005)
10. Franz, M., Williams, P., Carbunar, B., Katzenbeisser, S., Peter, A., Sion, R., Sotakova, M.: Oblivious outsourced storage with delegation. In: Financial Cryptography and Data Security, pp. 127–140. Springer (2012)
11. Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword search and oblivious pseudorandom functions. In: Theory of Cryptography, pp. 303–324. Springer (2005)
12. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) 43(3), 431–473 (1996)
13. Golle, P., Jakobsson, M., Juels, A., Syverson, P.: Universal re-encryption for mixnets. In: Topics in Cryptology–CT-RSA, pp. 163–178. Springer (2004)
14. Goodrich, M.T.: Randomized shellsort: A simple data-oblivious sorting algorithm. Journal of the ACM (JACM) 58(6), 27 (2011)
15. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious RAM simulation with efficient worst-case access overhead. In: ACM Cloud Computing Security Workshop (CCSW). pp. 95–100 (2011)
16. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious RAM simulation. In: SODA. pp. 157–167. SIAM (2012)
17. Jarecki, S., Liu, X.: Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In: Theory of Cryptography, pp. 577–594. Springer (2009)
18. Jinsheng, Z., Wensheng, Z., Qiao, D.: A multi-user oblivious ram for outsourced data (2014)
19. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in) security of hash-based oblivious ram and a new balancing scheme. In: SODA. pp. 143–156. SIAM (2012)
20. Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. In: Theory of Cryptography, pp. 377–396. Springer (2013)
21. Maffei, M., Malavolta, G., Reinert, M., Schröder, D.: Privacy and access control for outsourced personal records. In: Security and Privacy (S&P). pp. 341–358 (2015)
22. Ostrovsky, R.: Efficient computation on oblivious rams. In: Proceedings of the 22nd annual ACM symposium on Theory of computing. pp. 514–523. ACM (1990)
23. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Advances in cryptology–EUROCRYPT. pp. 223–238. Springer (1999)

24. Pinkas, B., Reinman, T.: Oblivious ram revisited. In: Advances in Cryptology–CRYPTO, pp. 502–519. Springer (2010)
25. Prabhakaran, M., Rosulek, M.: Rerandomizable rcca encryption. In: Advances in Cryptology–CRYPTO, pp. 517–534. Springer (2007)
26. Shi, E., Chan, T.H., Stefanov, E., Li, M.: Oblivious RAM with o((logn)3) worst-case cost. In: Advances in Cryptology–ASIACRYPT. pp. 197–214 (2011)
27. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: CCS. pp. 299–310 (2013)
28. The Tor project. `https://www.torproject.org/` (2003), accessed Feb 2016
29. Wang, X., Chan, T.H., Shi, E.: Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In: CCS. pp. 850–861 (2015)
30. Williams, P., Sion, R., Tomescu, A.: PrivateFS: a parallel oblivious file system. In: CCS. pp. 977–988 (2012)

A (Partially Key-Homomorphic) Oblivious PRF

We next present the oblivious PRF (OPRF) construction, based on [17], and mentioned in Section 4.1.

Definition 7 (Oblivious PRF—OPRF [11]). *A 2-party protocol π is an OPRF scheme if there exists some PRF family f_s such that π privately realizes the following functionality:* **Input:** *Client holds an evaluation point μ; Server S holds a key s.* **Output:** *Client outputs $f_s(\mu)$; Server outputs nothing.*

This can be denoted as a secure computation protocol for functionality $\mathcal{F}_{\mathsf{OPRF}}$: $(s, \mu) \to (\perp, f_s(\mu))$.

Our construction is based on the Jarecki-Liu OPRF [17] and presumes a malicious adversary, but a simpler version without zero-knowledge proof systems is suitable against an HbC adversary.

Server S (holder of OPRF keys) and user U (client of OPRF) have public keys $\mathsf{pk_S}$, $\mathsf{pk_U}$ and corresponding private keys $\mathsf{sk_S}$, $\mathsf{sk_U}$. The joint public key is $\mathsf{pk} = \mathsf{pk_S} \cdot \mathsf{pk_U}$. We denote $C_m = \mathsf{E}^+_{\mathsf{pk}}(m)$, $C_m^{(\mathsf{S})} = \mathsf{E}^+_{\mathsf{pk_S}}(m)$, and $C_m^{(\mathsf{U})} = \mathsf{E}^+_{\mathsf{pk_U}}(m)$. K^+ denotes the key generation algorithm of the encryption scheme, D^+ the decryption algorithm. Here, decryption of C_m using $\mathsf{sk_U}$ gives $C_m^{(\mathsf{S})}$, and decryption of C_m using $\mathsf{sk_S}$ gives $C_m^{(\mathsf{U})}$. We need the encryption scheme to satisfy additive homomorphism, verifiable encryption, and verifiable decryption properties as defined in [17, Sec. 2.2]. Encryption scheme satisfying these properties can be instantiated with CSEnc from Section 4.1, accompanied with suitable zero-knowledge proof systems, as specified in [17].

Theorem 5. *Assuming hardness of factoring of safe RSA moduli, a semantically secure encryption scheme on $\mathbb{Z}_n$ which satisfies properties listed above and assuming that each proof (of knowledge) system in Fig. 3 is zero-knowledge and simulation-sound, the protocol in Fig. 3 is a secure computation protocol for functionality $\mathcal{F}_{\mathsf{OPRF}}$.*

Proof. Our proof is similar to the proof of the original OPRF [17].

Constructing an ideal-world server $\mathsf{SIM_S}$ *from a malicious real server* S^**:* $\mathsf{SIM_S}$ interacts with S^* and $\mathcal{F}_{\mathsf{OPRF}}$, and does the following:

- If S^* succeeds in the proof π_1, then $\mathsf{SIM_S}$ runs the extraction algorithm for π_1 with S^* to extract s, $\hat{s}$, s.t. $\mathsf{pk}_1 = \mathsf{g}^s$, $\mathsf{pk}_2 = \mathsf{g}^{\hat{s}}$.
- $\mathsf{SIM_S}$ simulates the real-world user U as follows: 1. $(\mathsf{pk_U}, \mathsf{sk_U}) \leftarrow \mathsf{K}^+$. 2. $r \leftarrow_R \mathbb{Z}_n^*$, $C_r^{(\mathsf{U})} \leftarrow \mathsf{E}^+_{\mathsf{pk_U}}(r)$. 3. $a \leftarrow_R \mathbb{Z}_n^*$, $C_a^{(\mathsf{S})} \leftarrow \mathsf{E}^+_{\mathsf{pk_S}}(a)$. 4. $C_a^{(\mathsf{S})} \leftarrow \mathsf{E}^+_{\mathsf{pk_S}}(a)$. 5. Send $(\mathsf{pk_U}, C_r^{(\mathsf{U})}, C_a^{(\mathsf{S})})$ and simulate the proof π_2.
- If the proof π_3 verifies, $\mathsf{SIM_S}$ sends s, $\hat{s}$ to $\mathcal{F}_{\mathsf{OPRF}}$. On $\mathsf{SIM_S}$'s inputs $(s, \hat{s})$ and ideal world user $\bar{\mathsf{U}}$'s input μ, $\mathcal{F}_{\mathsf{OPRF}}$ outputs $f_{s,\hat{s}}(\mu)$ to $\bar{\mathsf{U}}$.

Let $\mathcal{D}$ be a distinguisher that controls the server S^*, chooses the input of the user U, and also observes the output of U. We argue that $\mathcal{D}$'s view in the real world (S^*'s view + U's output) and its view in the ideal world (S^*'s view + ideal user $\bar{\mathsf{U}}$'s output) are indistinguishable. To show this, we introduce a series of games $\mathsf{G}_0, \ldots, \mathsf{G}_6$ where G_0 is the real world experiment (S^* interacting with the real user U), G_6 is the ideal world experiment (S^* interacting with $\mathsf{SIM_S}$), and arguing that the views in G_i and G_{i+1} are indistinguishable.

G_1: Same as G_0 except that instead of proving π_2, U simulates it. By zero-knowledge of the π_2, $\mathcal{D}$'s views in G_0 and G_1 are indistinguishable.

G_2: Same as G_1 except that if S^* succeeds in the proof π_1, G_2 runs the extractor algorithm for π_1 with S^* to extract s, $\hat{s}$. By simulation soundness of π_1, G_2 extracts s, $\hat{s}$ with non-negligible probability.

Common input: $(n, \mathsf{g}, \mathsf{pk}_1, \mathsf{pk}_2)$. S's private input: s, $\hat{s}$, s.t. $\mathsf{g}^s = \mathsf{pk}_1$, $\mathsf{g}^{\hat{s}} = \mathsf{pk}_2$.
U's private input: μ.

Step 1 (S). $(\mathsf{pk_S}, \mathsf{sk_S}) \leftarrow \mathsf{K}^+$, $C_s^{(\mathsf{S})} \leftarrow \mathsf{E}^+_{\mathsf{pk_S}}(s)$,
$\pi_1 \leftarrow PoK\{s, \hat{s} | \, C_k^{(\mathsf{S})} \in \mathsf{E}^+_{\mathsf{pk_S}}(s),\ \mathsf{pk}_1 = \mathsf{g}^s,\ \mathsf{pk}_2 = \mathsf{g}^{\hat{s}}\}$. Send $\left(\mathsf{pk_S}, C_s^{(\mathsf{S})}\right), \pi_1$ to U.

Step 2 (U). If π_1 verifies, then $(\mathsf{pk_U}, \mathsf{sk_U}) \leftarrow \mathsf{K}^+$, $r \leftarrow_R \mathbb{Z}_n^*$, $C_r^{(\mathsf{U})} \leftarrow \mathsf{E}^+_{\mathsf{pk_U}}(r)$,
$C_a^{(\mathsf{S})} \leftarrow \left(C_s^{(\mathsf{S})} \cdot \mathsf{E}^+_{\mathsf{pk_S}}(\mu)\right)^r$, $\pi_2 \leftarrow PoK\left\{\mu | \exists r, \text{s.t. } C_r^{(\mathsf{U})} \in \mathsf{E}^+_{\mathsf{pk_U}}(r), C_a^{(\mathsf{S})} \in \left(C_s^{(\mathsf{S})} \cdot \mathsf{E}^+_{\mathsf{pk_S}}(\mu)\right)^r\right\}$.
Send $\left(\mathsf{pk_U}, C_r^{(\mathsf{U})}, C_a^{(\mathsf{S})}, \pi_2\right)$ to S.

Step 3 (S). If π_2 verifies, then $a \leftarrow \mathsf{D}^+_{\mathsf{sk_S}}(C_a^{(\mathsf{S})})$. If $gcd(n, a) \neq 1$, send $\perp$ to U and abort.
$b \leftarrow \hat{s} \cdot (a)^{-1} \bmod n$, $\varphi_{\mathsf{S}} \leftarrow_R \mathbb{Z}_n$, $v_{\mathsf{S}} \leftarrow \mathsf{g}^{\varphi_{\mathsf{S}}}$, $C_{\varphi_{\mathsf{U}}}^{(\mathsf{U})} \leftarrow \left(C_r^{(\mathsf{U})}\right)^b \cdot \mathsf{E}^+_{\mathsf{pk_U}}(-\varphi_{\mathsf{S}})$,

$$\pi_3 \leftarrow P\left\{\begin{array}{c} \exists a, \varphi_{\mathsf{S}}, b, \mathsf{sk_S} \text{ s.t. } a = \mathsf{D}^+_{\mathsf{sk_S}}\left(C_a^{(\mathsf{S})}\right),\ (\mathsf{pk_S}, \mathsf{sk_S}) \in \mathsf{KeyVal} \\ C_{\varphi_{\mathsf{U}}}^{(\mathsf{U})} \in \left(C_r^{(\mathsf{U})}\right)^b \cdot \mathsf{E}^+_{\mathsf{pk_U}}(-\varphi_{\mathsf{S}}),\ v_{\mathsf{S}} = \mathsf{g}^{\varphi_{\mathsf{S}}}, a \cdot b = \hat{s} \bmod n,\ \mathsf{pk}_2 = \mathsf{g}^{\hat{s}} \end{array}\right\}.$$

Send $\left(v_{\mathsf{S}}, C_{\varphi_{\mathsf{U}}}^{(\mathsf{U})}, \pi_3\right)$ to U.

Step 4 (U). Output $\perp$ if receiving $\perp$ from S, or if π_3 fails.
$\varphi_{\mathsf{U}} \leftarrow \mathsf{D}^+_{\mathsf{sk_U}}\left(C_{\varphi_{\mathsf{U}}}^{(\mathsf{U})}\right)$, $v_{\mathsf{U}} \leftarrow \mathsf{g}^{\varphi_{\mathsf{U}}}$. Output $v_{\mathsf{S}} \cdot v_{\mathsf{U}}$.

Fig. 3. Our (partially key-homomorphic) OPRF construction.

G_3: Same as G_2 except that if the proof π_3 verifies, then G_3 outputs $f_{s,\hat{s}}(\mu) = \mathsf{g}^{\hat{s}/(s+\mu)}$ as the final output (or $\perp$ if $gcd(s+\mu, n) \neq 1$). By simulation soundness of π_3, $\mathcal{D}$'s views in G_2 and G_3 are indistinguishable.

G_4: Same as G_3 except that as long as $gcd(s+\mu, n) = 1$, G_4 does the following: 1. $(\mathsf{pk}_\mathsf{U}, \mathsf{sk}_\mathsf{U}) \leftarrow \mathsf{K}^+$. 2. $a \leftarrow_R \mathbb{Z}_n^*$, $C_a^{(\mathsf{S})} \leftarrow \mathsf{E}^+_{\mathsf{pk}_\mathsf{S}}(a)$. 3. $r \leftarrow a \cdot \hat{s}/(s+\mu)$, $C_r^{(\mathsf{U})} \leftarrow \mathsf{E}^+_{\mathsf{pk}_\mathsf{U}}(r)$. 4. Simulate the proof π_2. The probability that $gcd(s+\mu, n) \neq 1$ is negligible assuming factoring safe RSA moduli is hard. If $gcd(s+\mu, n) = 1$, then the tuple $(\mathsf{pk}_\mathsf{U}, C_r^{(\mathsf{U})}, C_a^{(\mathsf{S})})$ is distributed identically in G_3 and G_4, and so $\mathcal{D}$'s views in these games are indistinguishable.

G_5: Same as G_4 except that value r is replaced by random $r' \in \mathbb{Z}_n^*$. By semantic security of the encryption scheme, $\mathcal{D}$'s views in G_4 and G_5 are indistinguishable. (See G_4 in the first part of the proof of Theorem 1 in [17] for a reduction.)

G_6: G_6 is the ideal world game between SIM_S (with access to S^*), $\mathcal{F}_{\mathsf{OPRF}}$, and the ideal world user $\bar{\mathsf{U}}$. Instead of computing Step 4. in G_5, the simulator SIM_S sends $(s, \hat{s})$ to $\mathcal{F}_{\mathsf{OPRF}}$ in G_6. On inputs $(s, \hat{s})$ from SIM_S and μ from $\bar{\mathsf{U}}$, $\mathcal{F}_{\mathsf{OPRF}}$ computes and sends to $\bar{\mathsf{U}}$ the value $f_{s,\hat{s}}(\mu)$. We have that $\mathcal{D}$'s views in G_5 and G_6 are indistinguishable.

Constructing an ideal-world user SIM_U from a malicious real-world user U^*:
SIM_U interacts with U^* and $\mathcal{F}_{\mathsf{OPRF}}$ and does the following:

- SIM_U picks $(\mathsf{pk}_\mathsf{S}, \mathsf{sk}_\mathsf{S}) \leftarrow_R \mathsf{K}^+$, $s' \leftarrow_R \mathbb{Z}_n^*$, computes $C_s^{\mathsf{S}} \leftarrow \mathsf{E}^+_{\mathsf{pk}_\mathsf{S}}(s')$, sends pk_S and $C_s^{(\mathsf{S})}$ to U^*, and simulates the proof π_1.
- If the proof π_2 verifies, SIM_U runs the extractor algorithm of π_2 with U^* to extract μ and sends it to $\mathcal{F}_{\mathsf{OPRF}}$.
- Getting $v = f_{s,\hat{s}}(\mu)$ from $\mathcal{F}_{\mathsf{OPRF}}$, which computes it on ideal-world server $\bar{\mathsf{S}}$'s inputs $(s, \hat{s})$ and SIM_U's input μ, SIM_U does the following:
  1. If $f_{s,\hat{s}}(\mu) = 1$, then SIM_U sends $\perp$ to U^* and aborts. 2. $\varphi_\mathsf{U} \leftarrow_R \mathbb{Z}_n$. 3. $v_\mathsf{S} \leftarrow v/\mathsf{g}^{\varphi_\mathsf{U}}$. 4. $C_{\varphi_r}^{(\mathsf{U})} \leftarrow \mathsf{E}^+_{\mathsf{pk}_\mathsf{U}}(\varphi_\mathsf{U})$. 5. Send $(v_\mathsf{S}, C_{\varphi_r}^{(\mathsf{U})})$ and simulate the proof π_3.

Let $\mathcal{D}$ be a distinguisher that controls the user U^*, chooses the input of the server S, and also observes the output of S. We show a series of games $\mathsf{G}_0, \ldots, \mathsf{G}_4$, where G_0 is the real world experiment, G_4 is the ideal world experiment, and argue that the views in G_i and G_{i+1} are indistinguishable.

G_1: same as G_0 except that S simulates the proofs π_1 and π_3. By zero-knowledge of these proof systems, $\mathcal{D}$'s views in G_0 and G_1 are indistinguishable.

G_2: same as G_1 except that if the proof π_2 verifies, G_2 runs the extractor algorithm for π_2 with U^* to extract μ. By simulation soundness of π_2, G_2 extracts μ with non-negligible probability.

G_3: same as G_2 except that it does the following after extracting μ: 1. $v = f_{s,\hat{s}}(\mu)$; if $v = 1$, send $\perp$ to U^* and abort. 2. $\varphi_\mathsf{U} \leftarrow_R \mathbb{Z}_n$, $v_\mathsf{S} \leftarrow v/\mathsf{g}^{\varphi_r}$. 3. $C_{\varphi_\mathsf{U}}^{(\mathsf{U})} \leftarrow \mathsf{E}^+_{\mathsf{pk}_\mathsf{U}}(\varphi_\mathsf{U})$. 4. Send $\left(v_\mathsf{S}, C_{\varphi_\mathsf{U}}^{(\mathsf{U})}\right)$ and simulate the proof π_3.

For the same arguments made in G_3 of the second part of the proof of Theorem 1 in [17], $\mathcal{D}$'s views in G_2 and G_3 are indistinguishable.

G_4: same as G_3 except that when v is to be computed, SIM_U sends the extracted μ to $\mathcal{F}_{\mathsf{OPRF}}$ (which also gets inputs $(s, \hat{s})$ from the ideal world server $\bar{\mathsf{S}}$)

and gets the value $v = f_{s,\hat{s}}(\mu)$ from the ideal functionality instead. We have that the views in G_3 and G_4 are indistinguishable. □

B Proof of $\mathsf{AnonRAM}_{\mathsf{polylog}}$

Theorem 6. $\mathsf{AnonRAM}_{\mathsf{polylog}}$ *provides access privacy against HbC adversaries controlling* S^* *in the random oracle model, when instantiated with a semantically secure universally re-randomizable encryption (UREnc) scheme, and a semantically secure additively homomorphic encryption and a secure (partially key-homomorphic) oblivious PRF schemes for appropriate compatible domains.*

Before proving Theorem 6, we state several facts and lemmas.

Fact 7. *During* User*, a user* U_i *modifies at most one block at level* $\ell > 1$*, and this block is a real block belonging to* U_i *and it is replaced with a dummy block by* U_i*.*

Based on Fact 7, after User is executed, the total number of dummy and real blocks at level $\ell > 1$ remains unchanged and these blocks are located at the same locations.

Fact 8. *After* User *is executed, the number of dummy or real blocks is increased by one at level 1.*

Lemma 1. *The number of non-empty blocks at any level* ℓ *is determined only by the current value* t*, and not by access pattern.*

Proof. It follows from Facts 7 and 8, and the invariant of Reshuffle from levels 1 to ℓ into $\ell + 1$ for some ℓ: all non-empty blocks from levels 1 to ℓ are moved to level $\ell + 1$, and Reshuffle does not introduce any new dummy or real blocks.

Lemma 2. *Before reshuffle levels* 1 *to* ℓ *into level* $\ell + 1$*, level* $\ell + 1$ *is empty.*

Proof. Analogously to the proof of Lemma 1 in [20].

Proof of Theorem 6. We argue that locations of U_i's real blocks and dummy locations used by U_i are not known to S^* (and therefore cannot be distinguished). First, the adversary cannot decrypt any part of block except for the block value of the blocks belonging to $\mathcal{U}^*$, so S^* merely observes the access locations used by U_i in User and can detect whether the real blocks of corrupted used has been changed by U_i. Since U_i does not change the blocks of other users while executing User (Fact 7), the only thing that could help the adversary is the locations accessed by U_i during User. And second, after Reshuffle, the server S^* does not know the dummy locations, and the adversary learns dummy locations used by the compromised users. Dummy locations used by U_i in User, on the other hand, are known only to U_i itself and the tag server T, since these locations are computed by T and sent to U_i in User, and they depend on private input $(\mu_{\mathcal{D}} + t)$ known only to T.

We define a series of games $\mathsf{G}_1, \ldots, \mathsf{G}_{10}$, where G_1 denotes an execution $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_0, \mathsf{HbC})$, and G_{10} an execution $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_1, \mathsf{HbC})$ for any two HbC-compliant access patterns AP_0, AP_1, and show that the views of the adversary in these games are indistinguishable.

We briefly describe the games. G_1 is the real experiment with AP_0. In G_2, an uncorrupted user U_i does not overwrite a found real block with the dummy block, but just re-randomizes the real block and writes to the first level the dummy block with the pre-tag computed for that level. In other words, the user re-randomizes *all* blocks read in Steps 2-4 of User. In G_3, the user always sets τ to the value corresponding to the dummy pre-tag in Step 3b of User. In G_4, the user sends random inputs to OPRF evaluation in Step 3a. In G_5, there is unique user $\mathsf{U}' \notin \mathcal{U}^*$ who performs accesses instead of any other user $\mathsf{U}_i \notin \mathcal{U}^*$. The remaining games $\mathsf{G}_6, \ldots, \mathsf{G}_{10}$ form a counter part for AP_1 (in the reversed order) such that $\mathsf{G}_5 = \mathsf{G}_6$. Below, we present reductions from G_1 to G_{10}.

G_2 - same as G_1 except for the changes in User presented below. We define series of games $\mathsf{G}_1^0, \ldots, \mathsf{G}_1^{|AP|}$ with $\mathsf{G}_1 := \mathsf{G}_1^{|AP|}$ and $\mathsf{G}_2 := \mathsf{G}_1^0$, where the games $\mathsf{G}_1^{\mathbf{i}+1}$ and $\mathsf{G}_1^{\mathbf{i}}$ differ in the following: if the $(\mathbf{i}+1)$-th access is made by some uncorrupted U_i, do not change the location and content of the real block in $\mathsf{G}_1^{\mathbf{i}}$, and instead add a dummy block to the first level. We can show that the views of the adversary in these games are indistinguishable by a reduction to DDH. For that we need to introduce additional intermediate games as tools that look as follows:

a. Change the encrypted value $(g^a, m \cdot g^{x_i a}, g^b, g^{x_i b})$ in a game G to some random value as $(g^a, \tilde{m} \cdot g^{x_i a}, g^b, g^{x_i b})$ for some random $\tilde{m} \in G_q$ in a game G'. We show that the views of the adversary in the games G and G' are indistinguishable by reduction to DDH: given a DDH tuple $(X, Y, Z) = (g^{\mathbf{x}}, g^{\mathbf{y}}, g^{\mathbf{z}}$ *or* $g^{\mathbf{xy}})$, we construct a distinguisher $\mathcal{D}$ against DDH as follows. $\mathcal{D}$ runs $\mathsf{Setup}(\lambda)$ of $\mathsf{AnonRAM}_{\mathsf{polylog}}$ where instead of generating the U_i's secret key x_i for UREnc, it sets $g^{x_i} = X$, and simulates the uncorrupted user U_i without x_i based on the second part of UREnc ciphertext. Then $\mathcal{D}$ replaces a challenge ciphertext with $(Y, Z \cdot m, g^b, X^b)$. If (X, Y, Z) is a true DH tuple, then the game proceeds as G, otherwise, i.e. if (X, Y, Z) is a random tuple, the game proceeds as in G'.

b. Change the ownership of a ciphertext $(g^a, m \cdot g^{xa}, g^b, g^{xb})$ under the key $x \in \mathbb{Z}_q$ in game G to some random key $\tilde{x} \in \mathbb{Z}_q$ and a random message $\tilde{m} \in G_q$ as $(g^a, \tilde{m} \cdot g^{\tilde{x}a}, g^b, g^{\tilde{x}b})$ in a game G'. We show that the views of the adversary in these two games are indistinguishable by a reduction to DDH: given a DDH tuple $(X, Y, Z) = (g^{\mathbf{x}}, g^{\mathbf{y}}, g^{\mathbf{z}}$ *or* $g^{\mathbf{xy}})$, we construct a distinguisher $\mathcal{D}$ against DDH as follows. $\mathcal{D}$ runs $\mathsf{Setup}(\lambda)$ of $\mathsf{AnonRAM}_{\mathsf{polylog}}$ and instead of generating U_i's secret key x_i for UREnc, it sets $X = g^{x_i}$ and simulates the uncorrupted user U_i without x_i. Then, $\mathcal{D}$ replaces the challenge ciphertext with $(g^a, X^a \cdot m, Y, Z)$. If (X, Y, Z) is a true DH tuple, then the game proceeds as G, otherwise, i.e. if (X, Y, Z) is a random tuple, the game proceeds as in G'.

These tools are applied to the block value, since they are encrypted using UREnc. We just mention that we also need the first tool (for changing the encrypted value) for the pre-tag part, which is encrypted using the semantically

secure encryption scheme on $\mathbb{Z}_n$. Having all these tools in place, we define intermediate games between $\mathsf{G}_1^{\mathbf{i}+1}$ and $\mathsf{G}_1^{\mathbf{i}}$, so that the views in these games are indistinguishable for the adversary.

G_3 - same as G_2, except that pre-tags in OPRF evaluations performed by U_i are replaced with dummy pre-tags computed and sent by T to U_i. Specifically, we define series of games $\mathsf{G}_2^0, \ldots, \mathsf{G}_2^{|AP_0|\cdot L}$ with $\mathsf{G}_2 := \mathsf{G}_2^{|AP_0|\cdot L}$ and $\mathsf{G}_3 := \mathsf{G}_2^0$, where games $\mathsf{G}_2^{\mathbf{i}+1}$ and $\mathsf{G}_2^{\mathbf{i}}$ differ in the following: if the $(\mathbf{i}+1)$-th OPRF evaluation is performed by U_i, set τ corresponding to the dummy pre-tag computed in Step 3a, in $\mathsf{G}_2^{\mathbf{i}}$.

Let E be the event that there exist two OPRF evaluations with at least one of them belonging to an honest user U_i, such that the OPRF is evaluated using the same input and the same key. We show that the probability of this event is negligible. In $\mathsf{G}_2^{\mathbf{i}+1}$, pre-tag is evaluated via OPRF using a storage server's key $(s, \hat{s})$ and some input $(\mu_{\mathcal{D}}+t)$ or (μ_i+j). In the former case, t is incremented every new access, therefore the OPRF evaluation will be pseudo-random (the special property of OPRF w.r.t. the second key applies only if the inputs to OPRF are the same), and so will be the tag computed via h_ℓ. In the latter case, i.e. if OPRF is evaluated using input (μ_i+j), the inputs to OPRF at specific level ℓ can be the same, however the keys used by S^* will be different with overwhelming probability. The reason lies in the mechanics of User and Reshuffle. Fix level $\ell > 1$. Assume $(\mu_i + j)$ was used as input to OPRF at level ℓ for some t. This only could happen if the real block was located at the level ℓ or at one of the next levels. Regardless of that level, once read, the real block will be moved by U_i to the first level. In any subsequent accesses, U_i will send $(\mu_i + j)$ as input to OPRF for level ℓ only if the real block is located at level ℓ or one of the next levels; denote the time (access counter) when this event happened as $t' > t$. The real block can be moved from lower levels to one of the next levels only by performing Reshuffle. In particular, there should have been Reshuffle of all levels from 1 to $\ell-1$ into level ℓ at time $t^* < t'$ and $t^* \geq t$. But when level ℓ is involved into Reshuffle, the storage server S^* generates a fresh second key for OPRF. The probability to draw some $\hat{s}'$ that already has been used as the second key to OPRF in the past is negligible, so is the probability of E. The specific property of OPRF is "neglected" since the adversary observes the tags, i.e. the output of h, which is modelled as a random oracle. Hence, the views of the adversary in games $\mathsf{G}_2^{\mathbf{i}+1}$ and $\mathsf{G}_2^{\mathbf{i}}$ are indistinguishable.

G_4 - same as G_3, except that uncorrupted users use random inputs to OPRF. Specifically, we define intermediate games $\mathsf{G}_3^0, \ldots, \mathsf{G}_3^{|AP_0|\cdot L}$ with $\mathsf{G}_3 := \mathsf{G}_3^{|AP_0|\cdot L}$ and $\mathsf{G}_4 := \mathsf{G}_3^0$, where $\mathsf{G}_3^{\mathbf{i}+1}$ and $\mathsf{G}_3^{\mathbf{i}}$ differ in the following: if the $(\mathbf{i}+1)$-th OPRF evaluation is performed by U_i, the input to OPRF is replaced by a random value. The output of OPRF is not used subsequently by U_i in $\mathsf{G}_3^{\mathbf{i}+1}$. So if Adv can distinguish between these two games, it can break the underlying assumptions of OPRF.

G_5 - same as G_4, except that all accesses performed by uncorrupted users are replaced with "equivalent" accesses performed by some fixed uncorrupted user U'. Note that uncorrupted users in G_4 do not use their secret inputs in OPRF

evaluation. The uncorrupted users re-randomize the blocks they have read, and introduce the dummy block to the first level. The views of the adversary in these two games are indistinguishable provided that communication channels between users and servers are anonymous.

The remaining games, $\mathsf{G}_6, \ldots, \mathsf{G}_{10}$ are defined analogously to $\mathsf{G}_1, \ldots, \mathsf{G}_5$, in reversed order, so that G_{10} corresponds to an experiment with AP_1, and whenever AP_0 is mentioned in $\mathsf{G}_1 - \mathsf{G}_5$, it should be replaced with AP_1 in $\mathsf{G}_{10} - \mathsf{G}_6$. We have $\mathsf{G}_5 = \mathsf{G}_6$. Hence the views in games G_1 and G_{10} are indistinguishable. □

Theorem 9. $\mathsf{AnonRAM}_{\mathsf{polylog}}$ *provides access privacy against HbC adversaries controlling* T^* *in the random oracle model, when instantiated with a semantically secure universally re-randomizable encryption (UREnc) scheme, and a semantically secure additively homomorphic encryption and a secure (partially key-homomorphic) oblivious PRF schemes for appropriate compatible domains.*

Proof. There are several arguments for the proof. First, while accessing S, a corrupted user from $\mathcal{U}^*$ reads and writes some blocks. Since S re-randomizes the block before sending it to users, the adversary cannot detect whether a particular bucket was touched or not, by an uncorrupted user. So, Adv has no information about tags used by an uncorrupted user, even though T^* knows pre-tags for dummy blocks in User.

Second, in $\mathsf{Reshuffle}$, T^* can observe pre-tags and identify empty blocks, but only with help of S since pre-tags are encrypted under the joint key TS, while the users' data (the value of block) are protected by semantic security of underlying UREnc. Before pre-tags become accessible to T^* in $\mathsf{Reshuffle}$, S preliminarily changes pre-tags (Step 2 of $\mathsf{Reshuffle}$) and shuffles array A. Intuitively, T^* should not learn the link between pre-tags observed in any two reshuffles, since the storage server S draws a new random second key to OPRF for the level ℓ, into which all the blocks from levels below are moved, every time $\mathsf{Reshuffle}$ is performed.

Finally, after pre-tags become accessible, T^* can identify empty cells, however, the number of empty blocks is array A in $\mathsf{Reshuffle}$ does not depend on the access pattern (Lemma 1). Let Adv denote an adversary who corrupts T^* and a subset of users $\mathcal{U}^*$. The goal is to show that Adv cannot distinguish between $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_0, \mathsf{HbC})$, and $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_1, \mathsf{HbC})$ for any two compliant access patterns AP_0, AP_1 significantly better than pure guessing.

We define a series of games $\mathsf{G}_1, \ldots, \mathsf{G}_{10}$, where G_1 denotes an execution $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_0, \mathsf{HbC})$, and G_{10} an execution $\mathsf{Exec}(\mathcal{AR}, \mathsf{Adv}, AP_1, \mathsf{HbC})$ for any two compliant access patterns AP_0, AP_1, and show that the views of the adversary in these games are indistinguishable. The games are defined in the same way as in the proof of Theorem 6, however reductions from G_1 to G_2 and from G_2 to G_3 proceed differently, while other reductions remain without changes. Below, we present reductions from G_1 to G_{10}.

G_2 - same as G_1 except for the changes in User: if the access is made by an uncorrupted user U_i, then in Steps 2 and 3b, re-randomize the found block, and write the dummy block in Step 5.

We introduce an intermediate game $\bar{\mathsf{G}}_1$ to apply the changes made in the game G_2 into two steps. In the first step (in $\bar{\mathsf{G}}_1$ compared to G_1), the changes affect only the block value, while the pre-tag remains as in G_1. And in the second step (in G_2 compared to $\bar{\mathsf{G}}_1$), pre-tags are changed, i.e. if User is performed by an uncorrupted user, then send $\mathsf{E}^+_{\mathcal{D}}(\theta_{\mathcal{D}})$ in Step 5 for $\theta_{\mathcal{D}}$ computed in Step 2, and re-randomize the pre-tag in Step 3b regardless of the value of **found**. The reason for this split is the fact that T^* observes pre-tags in $\mathsf{Reshuffle}$.

Reduction from G_1 to $\bar{\mathsf{G}}_1$ is done similarly to the reduction from G_1 to G_2 in the proof of Theorem 6 (except that pre-tags are not altered). For reduction from $\bar{\mathsf{G}}_1$ to G_2, assume w.l.o.g. that there is a list of all initial $M \cdot N$ real pre-tags and $M \cdot N$ dummy pre-tags. Each pre-tag is an element of G_n, so we can represent pre-tags as a list of $2M \cdot N$ distinct (probability of collision is negligible) group elements: $(g_1, \ldots, g_{2M \cdot N})$. The $\mathbf{i}$-th element of the list corresponds to a specific input to OPRF: $\mu_{\mathcal{D}} + (\mathbf{i} - N \cdot M)$ if $\mathbf{i} \geq M \cdot N$, and $\mu_{1+\mathbf{i}/M} + (\mathbf{i} \bmod M)$ otherwise. In Step 2 of $\mathsf{Reshuffle}$, w.l.o.g. we may assume that the list is changed by S to $(g_1^r, \ldots, g_{2M \cdot N}^r)$ for some random r, and T^* observes a subset of it ordered randomly. There are $|AP|/2$ such lists, and we refer to them as to the lists of pre-tags.

We define series of games $\bar{\mathsf{G}}_1^{0,0}, \ldots, \bar{\mathsf{G}}_1^{0,|AP|/2}, \bar{\mathsf{G}}_1^{1,0}, \ldots, \bar{\mathsf{G}}_1^{1,|AP|/2}, \ldots, \bar{\mathsf{G}}_1^{2M \cdot N,0}$, $\ldots$, $\bar{\mathsf{G}}_1^{2M \cdot N,|AP|/2}$ with $\bar{\mathsf{G}}_1 := \bar{\mathsf{G}}_1^{2M \cdot N,|AP|/2}$ and $\mathsf{G}_2 := \bar{\mathsf{G}}_1^{0,0}$, where the games $\bar{\mathsf{G}}_1^{\mathbf{i},\mathbf{j}+1}$ and $\bar{\mathsf{G}}_1^{\mathbf{i},\mathbf{j}}$ differ in the following: for the $\mathbf{i}$-th element of the $\mathbf{j}$-th list of pre-tags, if this element corresponds to the dummy input or to the input of uncorrupted user, replace the corresponding value with a random value, and related OPRF evaluations (for $t = 2\mathbf{j}$ and $t = 2\mathbf{j}+1$ and inputs w.r.t. $\mathbf{i}$-th element of the list) are responded by S with random values. We show that views of in $\bar{\mathsf{G}}_1^{\mathbf{i},\mathbf{j}+1}$ and $\bar{\mathsf{G}}_1^{\mathbf{i},\mathbf{j}}$ are indistinguishable to the adversary by a reduction to composite DDH. Let $\hat{s}_{\mathbf{j}}$ denote the second OPRF key used in the $\mathbf{j}$-th list of pre-tags. Given $(X, Y, Z) = (\mathsf{g}^x, \mathsf{g}^y, \mathsf{g}^z)$, we construct a distinguisher $\mathcal{D}$ as follows: on input μ corresponding to the $\mathbf{i}$-th element of the $\mathbf{j}$-th list, set $\mathsf{g}^{1/(s+\mu)} = X$, so that $f_{s,\hat{s}}(\mu) = X^{\hat{s}}$ for any $\hat{s} \neq \hat{s}_{\mathbf{j}}$, and $f_{s,\hat{s}_{\mathbf{j}}}(\mu) = Z$, where $\mathsf{g}^{\hat{s}_{\mathbf{j}}} = Y$. For other inputs $\mu' \neq \mu$, $f_{s,\hat{s}_{\mathbf{j}}}(\mu') = Y^{1/(\hat{s}+\mu')}$ (respectively, OPRF is simulated as $f_{1,\hat{s}_{\mathbf{j}}}(\mu')$ with $\mathsf{g} \leftarrow Y$). If (X, Y, Z) is a DH tuple, then the game proceeds as $\bar{\mathsf{G}}_1^{\mathbf{i},\mathbf{j}+1}$, otherwise, if (X, Y, Z) is a random tuple, the game proceeds as $\bar{\mathsf{G}}_1^{\mathbf{i},\mathbf{j}}$. Since dummy pre-tags and the pre-tags of uncorrupted users observed by T^* are replaced with random pre-tags, the views of the adversary in $\bar{\mathsf{G}}_1$ and G_2 are indistinguishable.

G_3: same as G_2, except that pre-tags in OPRF evaluations performed by an uncorrupted user U_i are replaced with dummy pre-tags computed and sent by T^* to U_i. Since the adversary does not learn tags used by uncorrupted users in User, the views in these games are indistinguishable.

Reductions from G_3 to G_5 are the same as in the proof of Theorem 6. Games $\mathsf{G}_6, \ldots, \mathsf{G}_{10}$ are defined analogously to $\mathsf{G}_1, \ldots, \mathsf{G}_5$, in reversed order, with the change that whenever AP_0 is mentioned, it is replaced with AP_1. We have that $\mathsf{G}_5 = \mathsf{G}_6$, and so the views of the adversary in games G_1 and G_{10} are indistinguishable. □

Proof of Theorem 3 . We have to show that $\mathsf{AnonRAM}_{\mathsf{polylog}}$ construction provides indistinguishability of access patterns in the two following cases: a) collusion of honest-but-curious subset of users $\mathcal{U}^*$ and the storage server, b) collusion of honest-but-curious subset of users $\mathcal{U}^*$ and the tag server T^*. In either case, we have to construct a simulator that simulates the execution of an access pattern (for uncorrupted users) without knowing it, such that the adversary is not able to distinguish between the real or simulated execution. Note that the probability of overflow in Step 8 of Reshuffle algorithm is determined by n balls to n bins experiment and therefore is small ($\leq 1/n$). As in GO-ORAM, the overflow itself does not help the adversary. The theorem follows from Theorems 6 and 9. □

C Detailed Description of $\mathbf{AnonRAM}^{\mathbf{M}}_{\mathbf{polylog}}$

In the following, we present the changes that have to be made in order to make $\mathsf{AnonRAM}_{\mathsf{polylog}}$ secure against malicious users.

Changes in User**.** We elaborate on ZK proof system, in which a user proves specific relations between the old (stored at S at the moment before the user started User) and the new ciphertexts (sent from the user to S during User).

We collect all the blocks read and written by U_i into the array of $w := (t \bmod 2)+(L-1)\beta$ blocks, $B_1, \ldots, B_w$, the modified array of w blocks, $\hat{B}_1, \ldots, \hat{B}_w$, and an extra new block $\hat{B}_0$. Each block B_k is the pair (γ_k, v_k), where $\gamma_k := (\gamma_k[1], \gamma_k[2])$ denotes encryption of the pre-tag, and $v_k := (v_k[1], v_k[2], v_k[3], v_k[4])$ encryption of the value. During User, the tag server T computes L pre-tags for the dummy blocks for each level, and the user computes a pre-tag for the new block (in the proof, we are interested only in aforementioned $L+1$ pre-tags). These pre-tags are sent (computed) to (by) the user in Steps 2 and 3a (Step 5) of User. In the modified version of User, the encryptions of dummy pre-tags are also sent to S, so that S could verify ZK proofs sent by the user to S at the end of User. Let $\gamma_{\mathcal{D}_\ell}$ denote the encrypted pre-tag for the dummy block computed for level $\ell \in [1, L]$, and γ_0 denote the dummy pre-tag computed in Step 5 of User. Let $v_{\mathcal{D}} := \mathsf{E}^*_{\mathsf{TS}}(\text{“}dummy\text{”})$ denote the public ciphertext initialized in Setup.

The proof consists of $w + 1$ individual parts. For each $k \in \{1, \ldots, w\}$, the individual k-th part looks as described in Figure 4. The ZK proof for the individual part consists of six components. The first two components in the proof correspond to re-randomization of the block B_k: component 1 states that $\hat{B}_k$ encrypts the same pre-tag as B_k, and component 2 states that $\hat{B}_k$ and B_k encrypt the same block value under the same key.

The next four components correspond to the case when a real block is found by the user. The user then replaces that block with the dummy block and stores an updated value of the real block as $\hat{B}_0$. Specifically:

- component 3 states that the user knows the secret key, using which she can decrypt the value part of B_k;
- component 4 states that $\hat{B}_k$ encrypts the same pre-tag as $\gamma_{\mathcal{D}_\ell}$;
- component 5 states that the block value of $\hat{B}_k$ is a re-randomization of $v_{\mathcal{D}}$;

$$
P\left\{\begin{array}{ll}\exists r_1, r_2, r_3 \text{ s.t. } \hat{\gamma}_k = (\gamma_k[1] \cdot \mathsf{g}^{r_1}, \gamma_k[2] \cdot (\mathsf{pk}_{\mathsf{TS}})^{r_1}) & 1 \\ \hat{v}_k = (v_k[1](v_k[3])^{r_2}, v_k[2](v_k[4])^{r_2}, (v_k[3])^{r_3}, (v_k[4])^{r_3}) & 2\end{array}\right\}
$$

$$
\vee
$$

$$
PoK\left\{\begin{array}{ll}x| \exists\ r_4, r_5, r_6, r_7, r_8 \text{ s.t. } v_k[4] = (v_k[3])^x & 3 \\ \hat{\gamma}_k = (\gamma_{\mathcal{D}_\ell}[1] \cdot \mathsf{g}^{r_4}, \gamma_{\mathcal{D}_\ell}[2] \cdot (\mathsf{pk}_{\mathsf{TS}})^{r_4}), & 4 \\ \hat{v}_k = (v_{\mathcal{D}}[1](v_{\mathcal{D}}[3])^{r_5}, v_{\mathcal{D}}[2](v_{\mathcal{D}}[4])^{r_5}, (v_{\mathcal{D}}[3])^{r_6}, (v_{\mathcal{D}}[4])^{r_6}) & 5 \\ (\hat{v}_0[3], \hat{v}_0[4]) = ((v_k[3])^{r_8}, (v_k[4])^{r_8}) & 6\end{array}\right\}
$$

Fig. 4. Part of zero-knowledge proof system for $\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{polylog}}$ relationship between blocks B_k and $\hat{B}_k$ at level ℓ, represented as ciphertexts (γ_k, v_k) and $(\hat{\gamma}_k, \hat{v}_k)$ respectively, and a new block $\hat{B}_0$, represented as $(\hat{\gamma}_0, \hat{v}_0)$. Encryption of pre-tag for the dummy block associated with $\hat{B}_k$ is denoted as $\gamma_{\mathcal{D}_\ell}$. Encryption of the constant dummy value is denoted as $v_{\mathcal{D}}$.

- component 6 states that the value of $\hat{B}_0$ is encrypted under the same key as the user's ciphertext v_k.

Note that we do not restrict the value of $\hat{B}_0$, since the user may want to update the value of her cell to some other value, different from the value of B_k.

The last, $(w+1)$-th, part of ZK proof system is $P\{\bigvee_{k=0}^{w} \exists r_k \text{ s.t. } (\hat{v}_k[3], \hat{v}_k[4]) = ((v_{\mathcal{D}}[3])^{r_k}, (v_{\mathcal{D}}[4])^{r_k})\}$. It states that the value in at least one of the new blocks is encrypted under the same key as the dummy value $v_{\mathcal{D}}$. This part of ZK proof system tolerates a malicious user who just re-randomizes all blocks she has read (and thus proving only components 1-3 in Figure 4) and introduces a new block of an honest user. In this case, the newly introduced block $\hat{B}_0$ has to be encrypted using the shared key $\mathsf{pk}_{\mathsf{TS}}$.

The user sends $w+1$ proofs to the server S at the end of User command. The server rejects if at least one of the proofs does not verify.

Changes in $\mathsf{Reshuffle}$**.** $\mathsf{Reshuffle}$ has to be modified as follows: if T observes two or more blocks with same pre-tag, it replaces them with empty blocks. To this end, T performs an additional preparation step: OSort array A by pre-tags in the ascending order, then Scan A and replace any consecutive blocks having the same pre-tag with empty blocks. This modification to $\mathsf{Reshuffle}$ circumvents a malicious scenario in which too many blocks have the same pre-tag (such blocks would be mapped to the same tag and cause a bucket overflow with high probability).

D Postponed Proof for $\mathsf{AnonRAM}^{\mathbf{M}}_{\mathsf{polylog}}$

Proof Sketch of Theorem 4. The argument for integrity is based on ZK proof systems introduced in Section 4.7. They ensure that a user can modify the block value only if she knows the secret key which is required for decryption of the value. Since all the users generate their keys independently, the probability that a malicious user from $\mathcal{U}^*$ knows the secret key of any uncorrupted user is negligible.

The employed ZK proof system also ensures that the introduced to the first level block is encrypted either under the same key as one of the blocks, for which the user has proven the knowledge of the secret key, or under the joint servers' key. Thus, the probability that the adversary changes during User any block belonging to an honest user, or introduces a new block of an honest user, is negligible.

The privacy properties are preserved based on security analysis of $\mathsf{AnonRAM}_{\mathsf{polylog}}$ secure against HbC adversaries. □