2015 IEEE Symposium on Security and Privacy

Privacy and Access Control for Outsourced Personal Records

Matteo Maffei, Giulio Malavolta, Manuel Reinert, Dominique Schröder
Saarland University & CISPA
{maffei,malavolta,reinert,schroeder}@cs.uni-saarland.de

***Abstract*—Cloud storage has rapidly become a cornerstone of many IT infrastructures, constituting a seamless solution for the backup, synchronization, and sharing of large amounts of data. Putting user data in the direct control of cloud service providers, however, raises security and privacy concerns related to the integrity of outsourced data, the accidental or intentional leakage of sensitive information, the profiling of user activities and so on. Furthermore, even if the cloud provider is trusted, users having access to outsourced files might be malicious and misbehave. These concerns are particularly serious in sensitive applications like personal health records and credit score systems.**

To tackle this problem, we present GORAM**, a cryptographic system that protects the secrecy and integrity of outsourced data with respect to both an untrusted server and malicious clients, guarantees the anonymity and unlinkability of accesses to such data, and allows the data owner to share outsourced data with other clients, selectively granting them read and write permissions.** GORAM **is the first system to achieve such a wide range of security and privacy properties for outsourced storage. In the process of designing an efficient construction, we developed two new, generally applicable cryptographic schemes, namely, batched zero-knowledge proofs of shuffle and an accountability technique based on chameleon signatures, which we consider of independent interest. We implemented** GORAM **in Amazon Elastic Compute Cloud (EC2) and ran a performance evaluation demonstrating the scalability and efficiency of our construction.**

I. Introduction

Cloud storage has rapidly gained a central role in the digital society, serving as a building block of consumer-oriented applications (e.g, Dropbox, Microsoft SkyDrive, and Google Drive) as well as particularly sensitive IT infrastructures, such as personal record management systems. For instance, credit score systems rely on credit bureaus (e.g., Experian, Equifax, and TransUnion in US) collecting and storing information about the financial status of users, which is then made available upon request. As a further example, personal health records (PHRs) are more and more managed and accessed through web services (e.g., private products like Microsoft HealthVault and PatientsLikeMe in US and national services like ELGA in Austria), since this makes PHRs readily accessible in case of emergency even without the physical presence of the e-health card and eases their synchronization across different hospitals.

Despite its convenience and popularity, cloud storage poses a number of security and privacy issues. The first problem is related to the *secrecy* of user data, which are often sensitive (e.g., PHRs give a complete picture of the health status of citizens) and, thus, should be concealed from the server. A crucial point to stress is that preventing the server from reading user data (e.g., through encryption) is necessary but not sufficient to protect the privacy of user data. Indeed, as shown in the literature [1], [2], the capability to link consecutive accesses to the same file can be exploited by the server to learn sensitive information: for instance, it has been shown that the access patterns to a DNA sequence allow for determining the patient's disease. Hence the *obliviousness* of data accesses is another fundamental property for sensitive IT infrastructures: the server should not be able to tell whether two consecutive accesses concern the same data or not, nor to determine the nature of such accesses (read or write). Furthermore, the server has in principle the possibility to modify client's data, which can be harmful for several reasons: for instance, it could drop data to save storage space or modify data to influence the statistics about the dataset (e.g., in order to justify higher insurance fees or taxes). Therefore another property that should be guaranteed is the *integrity* of user data.

Finally, it is often necessary to share outsourced documents with other clients, yet in a controlled manner, i.e., selectively granting them read and write permissions: for instance, PHRs are selectively shared with the doctor before a medical treatment and a prescription is shared with the pharmacy in order to buy a medicine. *Data sharing* complicates the enforcement of secrecy and integrity properties, which have to be guaranteed not only against a malicious server but also against malicious clients. Notice that the simultaneous enforcement of these properties is particularly challenging, since some of them are in seeming contradiction. For instance, *access control* seems to be incompatible with the obliviousness property: if the server is not supposed to learn which file the client is accessing, how can he check that the client has the rights to do so?

A. Our Contributions

In this work, we present GORAM, a novel framework for privacy-preserving cloud-storage. Users can share outsourced data with other clients, selectively granting them read and write permissions, and verify the integrity of such data. These are hidden from the server and access patterns are oblivious. GORAM is the first system to achieve such a wide range of security and privacy properties for storage outsourcing. More specifically, the contributions of this work are the following:

- We formalize the problem statement by introducing the notion of Group Oblivious RAM (GORAM). GORAM extends the concept of Oblivious RAM [3] (ORAM) [1] by considering multiple, possibly malicious clients, with

[1]ORAM is a technique originally devised to protect the access pattern of software on the local memory and then used to hide the data and the user's access pattern in storage outsourcing services.

DOI 10.1109/SP.2015.28

read and/or write access to outsourced data, as opposed to a single client. We propose a formal security model that covers a variety of security and privacy properties, such as data integrity, data secrecy, obliviousness of access patterns, and anonymity.

- We first introduce a cryptographic instantiation based on a novel combination of ORAM [4], predicate encryption [5], and zero-knowledge (ZK) proofs (of shuffle) [6], [7]. This construction is secure, but building on off-the-shelf cryptographic primitives is not practical. In particular, clients prove to the server that the operations performed on the database are correct through ZK proofs of shuffle, which are expensive when the entries to be shuffled are tuples of data, as opposed to single entries.
- As a first step towards a practical instantiation, we maintain the general design, but we replace the expensive ZK proofs of shuffle with a new proof technique called *batched* ZK proofs of shuffle. A batched ZK proof of shuffle significantly reduces the number of ZK proofs by "batching" several instances and verifying them together. Since this technique is generically applicable in any setting where one is interested to perform a zero-knowledge proof of shuffle over a list of entries, each of them consisting of a tuple of encrypted blocks, we believe that it is of independent interest. This second realization greatly outperforms the first solution and is suitable for databases with relatively small entries, accessed by a few users, but it does not scale to large entries and many users.
- To obtain a scalable solution, we explore some trade-offs between security and efficiency. First, we present a new accountability technique based on chameleon signatures. The idea is to let clients perform arbitrary operations on the database, letting them verify each other's operation a-posteriori and giving them the possibility to blame misbehaving parties. Secondly, we replace the relatively expensive predicate encryption, which enables sophisticated role-based and attribute-based access control policies, with the more efficient broadcast encryption, which suffices to enforce per-user read/write permissions, as required in the personal record management systems we consider. This approach leads to a very efficient solution that scales to large files and thousands of users, with a combined communication-computation overhead of only 7% (resp. 8%) with respect to state-of-the-art, single-client ORAM constructions for reading (resp. writing) on a 1GB storage with 1MB block size (for larger datasets or block sizes, the overhead is even lower).

We have implemented GORAM in Amazon Elastic Compute Cloud (EC2) and conducted a performance evaluation demonstrating the scalability and efficiency of our construction. Although GORAM is generically applicable, the large spectrum of security and privacy properties, as well as the efficiency and scalability of the system, make GORAM particularly suitable for the management of large amounts of sensitive data, such as personal records.

B. Outline

Section II introduces the notion of Group ORAM. We discuss the general cryptographic instantiation in Section III, the accountability-based construction in Section IV, and the efficient scheme with broadcast encryption in Section V. Section VI formalizes the security properties of Group ORAM and Section VII states the security and privacy results. We implemented our system and conducted an experimental evaluation, as discussed in Section VIII. Section IX presents a case study on PHRs. The related work is discussed in Section X. Section XI concludes and outlines future research directions.

Due to space constraints, we postpone the proofs to the long version [8].

II. System Settings

We detail the problem statement by formalizing the concept of Group ORAM (Section II-A), presenting the relevant security and privacy properties (Section II-B), and introducing the attacker model (Section II-C).

A. Group ORAM

We consider a data owner $\mathcal{O}$ outsourcing her database $\mathsf{DB} = d_1, \ldots, d_m$ to the server $\mathcal{S}$. A set of clients $\mathcal{C}_1, \ldots, \mathcal{C}_n$ can accesses parts of the database, as specified by the access control policy set by $\mathcal{O}$. This is formalized as an n-by-m matrix $\mathbf{AC}$, defining the permissions of the clients on the files in the database: $\mathbf{AC}(i, j)$ (i.e., the j-th entry of the i-th row) denotes the access mode for client i on data d_j. Each entry in the matrix is an element of the set $\{\perp, r, rw\}$ of access modes, denoting no access, read access, and write access, respectively.

At registration time, each client $\mathcal{C}_i$ receives a capability cap_i, which gives $\mathcal{C}_i$ access to DB as specified in the corresponding row of $\mathbf{AC}$. Furthermore, we assume the existence of a capability $cap_{\mathcal{O}}$, which grants permissions for all of the operations that can be executed by the data owner only.

In the following we formally characterize the notion of Group ORAM. Intuitively, a Group ORAM is a collection of two algorithms and four interactive protocols, used to setup the database, add clients, add an entry to the database, change the access permissions to an entry, read an entry, and overwrite an entry. In the sequel, we let $\langle A, B \rangle$ denote a protocol between the PPT machines A and B, $|\mathbf{a}|$ the length of the vector $\mathbf{a}$ of access modes, and $\mathbf{a}(i)$ the element at position i in $\mathbf{a}$. In all our protocols $|\mathsf{DB}|$ is equal to the number of columns of $\mathbf{AC}$.

Definition 1 (Group ORAM): A *Group ORAM* scheme is a tuple of (interactive) PPT algorithms $\mathsf{GORAM} = (\mathsf{gen}, \mathsf{addCl}, \mathsf{addE}, \mathsf{chMode}, \mathsf{read}, \mathsf{write})$, such that:

$(cap_{\mathcal{O}}, \mathsf{DB}) \leftarrow \mathsf{gen}(1^\lambda, n)$: The gen algorithm initializes the database $\mathsf{DB} := [\ ]$ and the access control matrix $\mathbf{AC} := [\ ]$, and generates the access capability $cap_{\mathcal{O}}$ of the data owner. The parameter n determines the maximum number of clients. This algorithm returns $(cap_{\mathcal{O}}, \mathsf{DB})$, while $\mathbf{AC}$ is a global variable that maintains a state across the subsequent algorithm and protocol executions.

$\{cap_i, \mathsf{deny}\} \leftarrow \mathsf{addCl}(cap_{\mathcal{O}}, \mathbf{a})$: The addCl algorithm is run by the data owner, who possesses $cap_{\mathcal{O}}$, to register a new client, giving her access to the database as specified by the vector $\mathbf{a}$. If $|\mathbf{a}|$ is equal to the number of columns of $\mathbf{AC}$, $\mathbf{a}$ is appended to $\mathbf{AC}$ as the last row and the algorithm outputs a fresh capability cap_i that is assigned to that row. Otherwise, it outputs deny.

$\{\mathsf{DB'}, \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{addE}}(cap_{\mathcal{O}}, \mathbf{a}, d), \mathcal{S}_{\mathsf{addE}}(\mathsf{DB})\rangle$: This protocol is run by the data owner, who possesses $cap_{\mathcal{O}}$, to append an element d to DB, assigning the vector $\mathbf{a}$ of access modes. If $|\mathbf{a}|$ is equal to the number of rows of $\mathbf{AC}$ then d is appended to DB, $\mathbf{a}$ is appended to $\mathbf{AC}$ as the last column, and the protocol outputs the new database DB'; otherwise it outputs deny.

$\langle \mathcal{C}_{\mathsf{chMode}}(cap_{\mathcal{O}}, \mathbf{a}, j), \mathcal{S}_{\mathsf{chMode}}(\mathsf{DB})\rangle$: This protocol is used by the data owner, who possesses $cap_{\mathcal{O}}$, to change the access permissions for the j-th entry as specified by the vector $\mathbf{a}$ of access modes. If $j \leq |\mathsf{DB}|$ and $|\mathbf{a}|$ is equal to the number of rows of $\mathbf{AC}$, then the j-th column of $\mathbf{AC}$ is replaced by $\mathbf{a}$.

$\{d, \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{read}}(cap_i, j), \mathcal{S}_{\mathsf{read}}(\mathsf{DB})\rangle$: The interactive read protocol is used by the owner of cap_i to read the j-th entry of DB. This protocol returns either $d := \mathsf{DB}(j)$ or deny if $|\mathsf{DB}| < j$ or $\mathbf{AC}(i,j) = \bot$.

$\{\mathsf{DB}', \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{write}}(cap_i, j, d), \mathcal{S}_{\mathsf{write}}(\mathsf{DB})\rangle$: The interactive write protocol is used by the owner of the capability cap_i to overwrite the j-th entry of DB with d. This protocol succeeds and outputs DB' if and only if $\mathbf{AC}(i,j) = rw$, otherwise it outputs deny.

B. Security and Privacy Properties

Here we briefly outline the fundamental security and privacy properties achieved by a Group ORAM. We refer to Section VI for a precise formalization of these properties based on cryptographic games.

Secrecy: clients can only read entries they hold *read* permissions on.

Integrity: clients can only write entries they hold *write* permissions on.

Tamper-resistance: clients, eventually colluding with the server, cannot modify an entry they do not hold *write* permission on without being detected by the data owner.

Obliviousness: the server cannot determine the access pattern on the data given a clients' sequence of operation.

Anonymity: the server and the data owner cannot determine who performed a given operation, among the set of clients that are allowed to perform it.

Accountable Integrity: clients cannot write entries they do not hold *write* permission on without being detected.

C. The Attacker Model

We consider an adversarial model in which the data owner $\mathcal{O}$ is honest, the clients $\mathcal{C}_1, \ldots, \mathcal{C}_n$ may be malicious, and the server $\mathcal{S}$ is assumed to be honest-but-curious (HbC)[2] and not to collude with clients. These assumptions are common in the literature (see, e.g., [9], [10]) and are well justified in a cloud setting, since it is of paramount importance for service providers to keep a good reputation, which discourages them from visibly misbehaving, while they may have an incentive in passively gathering sensitive information given the commercial interest of personal data.

Although we could limit ourselves to reason about all security and privacy properties in this attacker model, we find it interesting to state and prove some of them even in a stronger attacker model, where the server can arbitrarily misbehave. This allows us to characterize which properties unconditionally hold true in our system, i.e., even if the server gets compromised (cf. the discussion in Section VI).

[2]I.e., the server is regarded as a passive adversary, following the protocol but seeking to gather additional information

III. Our Construction (GORAM)

In this section, we first show how to realize a Group ORAM using a novel combination of ORAM, predicate encryption, and zero-knowledge proofs (Section III-A and Section III-B). Since even the usage of the most efficient zero-knowledge proof system still yields an inefficient construction, we introduce a new proof technique called batched ZK proofs of shuffle (Section III-C) and instantiate our general framework with this primitive.

A. Prerequisites

In the following, we describe the database layout, the basic cryptographic primitives, and the system assumptions.

Layout of the database. The layout of the database DB follows the one proposed by Stefanov *et al.* [4]. To store N data entries, we use a binary tree T of depth $D = O(\log N)$, where each node stores a bucket of entries, say b entries per bucket. We denote a node at depth d and row index i by $T_{d,i}$. The depth at the root ρ is 0 and increases from top to bottom; the row index increases from left to right, starting at 0. We often refer to the root of the tree as ρ instead of $T_{0,0}$. Moreover, Path-ORAM [4] uses a so-called *stash* as local storage to save entries that would overflow the root bucket. We assume the stash to be stored and shared on the server like every other node, but we leave it out for the algorithmic description. The stash can also be incorporated in the root node, which does not carry b but $b + s$ entries where s is the size of the stash. The extension of the algorithms is straight-forward (only the number of downloaded entries changes) and does not affect their computational complexity. In addition to the database, there is an index structure $\mathcal{LM}$ that maps entry indices i to leaf indices l_i. If an entry index i is mapped in $\mathcal{LM}$ to l_i then the entry with index i can be found in some node on the path from the leaf l_i to the root ρ of the tree. Finally, to initialize the database we fill it with dummy elements.

Cryptographic preliminaries. We informally review the cryptographic building blocks and introduce a few useful notations.

We denote by $\Pi_{\mathsf{SE}} = (\mathsf{Gen}_{\mathsf{SE}}, \mathcal{E}, \mathcal{D})$ a private-key encryption scheme, where $\mathsf{Gen}_{\mathsf{SE}}$ is the key-generation algorithm and $\mathcal{E}$ (resp. $\mathcal{D}$) is the encryption (resp. decryption) algorithm. Analogously, we denote by $\Pi_{\mathsf{PKE}} = (\mathsf{Gen}_{\mathsf{PKE}}, \mathsf{Enc}, \mathsf{Dec})$ a public-key encryption scheme. We also require a publicly available function Rerand to rerandomize public-key ciphertexts. We require that both encryption schemes fulfill the IND-CPA-security property [11].

A predicate encryption scheme [5] $\Pi_{\mathsf{PE}} = (\mathsf{PrGen}, \mathsf{PrKGen}, \mathsf{PrEnc}, \mathsf{PrDec})$ consists of a setup algorithm PrGen, a key-generation algorithm PrKGen, and an encryption (resp. decryption) algorithm PrEnc (resp. PrDec). In a predicate encryption scheme, one can encrypt a message m under an attribute x. The resulting ciphertext can only be decrypted with a secret key that encodes a predicate f such that

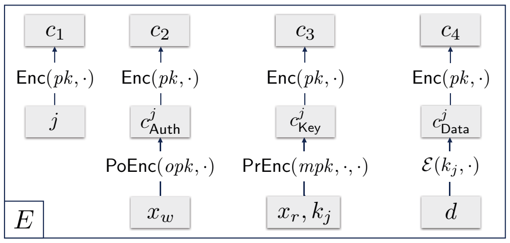


Figure 1. The structure of an entry with index j, payload d, write access regulated by the attribute x_w, and read access regulated by the attribute x_r.

$f(x) = 1$. The choice of predicates determines who can decrypt which ciphertext, which makes predicate encryption a flexible cryptographic tool to enforce access control policies. We further use a predicate-only encryption scheme $\Pi_{\mathsf{PO}} = (\mathsf{PoGen}, \mathsf{PoKGen}, \mathsf{PoEnc}, \mathsf{PoDec})$. The difference from Π_{PE} is that the attribute x is encrypted only. As for public-key encryption, we require rerandomization functions PrRR and PoRR for Π_{PE} and Π_{PO}. We require that both Π_{PE} and Π_{PO} are (selectively) *attribute-hiding* [5]. This security notion says that the adversary learns nothing about the message *and* the associated attribute (except the information that is trivially leaked by the keys that the adversary has).

Intuitively, a zero-knowledge (ZK) proof system $\mathcal{ZKP}$ is a proof system that combines two fundamental properties. The first property, soundness, says that it is (computationally) infeasible to produce a ZK proof of a wrong statement. The second property, zero-knowledge, means that no information besides the validity of the proven statement is leaked. A non-interactive zero-knowledge proof is a zero-knowledge protocol consisting of one message sent by the prover to the verifier. A zero-knowledge proof of knowledge additionally ensures that the prover knows the witnesses to the given statement. We denote by $PK\{(\vec{x}) : F\}$ a zero-knowledge proof of knowledge of the variables in $\vec{x}$ such that the statement F holds. Here, $\vec{x}$ is the set of witnesses, existentially quantified in the statement, and the proof does not reveal any of them. For instance, the proof $PK\{(r) : c = \mathsf{Rerand}(pk, d, r)\}$ shows that two public-key ciphertexts c and d, encrypted with the same public key pk, encrypt the same plaintext, i.e., c is obtained by rerandomizing d with the secret randomness r.

Structure of an entry and access control modes. Abstractly, database entries are tuples of the form $E = (c_1, c_2, c_3, c_4)$ where $c_1, \ldots, c_4$ are ciphertexts obtained using a public-key encryption scheme (see Figure 1). In particular, c_1 is the encryption of an index j identifying the j-th entry of the database; c_2 is the encryption of a predicate-only ciphertext c^j_{Auth}, which regulates the write access to the payload stored at j using the attribute x_w; c_3 is the encryption of a ciphertext c^j_{Key}, which is in turn the predicate encryption of a private key k_j with attribute x_r, regulating the read access; c_4 is the encryption of the ciphertext c^j_{Data}, which is the encryption with the private key k_j of the data d stored at position j in the database. We use the convention that an index $j > |\mathsf{DB}|$ indicates a dummy entry and we maintain the invariant that every client may write each dummy entry.

Algorithm 1 $(cap_{\mathcal{O}}, \mathsf{DB}) \leftarrow \mathsf{gen}(1^\lambda, n)$.

Input: security parameter 1^λ, number of clients n
Output: the capability of the data owner $cap_{\mathcal{O}}$

```
1: $(pk, sk) \leftarrow \mathsf{Gen}_{\mathsf{PKE}}(1^\lambda)$
2: $(opk, osk) \leftarrow \mathsf{PoGen}(1^\lambda, n)$
3: $(mpk, psk) \leftarrow \mathsf{PrGen}(1^\lambda, n)$
4: give $pk$ to the server $\mathcal{S}$
5: initialize DB on $\mathcal{S}$, $\mathcal{ADB} := \{\}$, $cnt_{\mathcal{C}} := 0$, $cnt_E := 0$
6: return $cap_{\mathcal{O}} := (cnt_{\mathcal{C}}, cnt_E, sk, osk, psk)$
```

Intuitively, in order to implement the access control modes $\bot$, r, and rw on a data index j, each client $\mathcal{C}_i$ is provided with a capability cap_i that is composed of three keys, the secret key corresponding to the top level public-key encryption scheme, a secret key for the predicate-only encryption scheme, and a secret key for the predicate encryption scheme. More specifically, if $\mathcal{C}_i$'s mode for j is $\bot$, then cap_i allows for decrypting neither c^j_{Auth} nor c^j_{Key}. If $\mathcal{C}_i$'s mode for j is r, then cap_i allows for decrypting c^j_{Key} but not c^j_{Auth}. Finally, if $\mathcal{C}_i$'s mode for j is rw, then cap_i allows for decrypting both c^j_{Auth} and c^j_{Key}. Intuitively, in order to replace an entry, a client has to successfully prove that she can decrypt the ciphertext c^j_{Auth}.

System assumptions. We assume that each client has a local storage of $O(\log N)$. Notice that the leaf index mapping has size $O(N)$, but the local client storage can be decreased to $O(\log N)$ by applying a standard ORAM construction recursively to it, as proposed by Shi *et al.* [12]. Additionally, the data owner stores a second database $\mathcal{ADB}$ that contains the attributes x_w and x_r associated to every entry in DB as well as predicates f_i associated to the client identities $\mathcal{C}_i$. Intuitively, $\mathcal{ADB}$ implements the access control matrix $\mathbf{AC}$ used in Definition 1. Since also $\mathcal{ADB}$ has size $O(N)$, we use the same technique as the one employed for the index structure. We further assume that clients establish authenticated channels with the server. These channels may be anonymous (e.g., by using anonymity networks [13] and anonymous credentials for the login [14]–[17]), but not necessarily.

B. Description of the Algorithms

Implementation of $(cap_{\mathcal{O}}, \mathsf{DB}) \leftarrow \mathsf{gen}(1^\lambda, n)$ **(Algorithm 1).** Intuitively, the data owner initializes the cryptographic schemes (lines 1.1–1.3) as well as the rest of the infrastructure (lines 1.4–1.5), and finally outputs $\mathcal{O}$'s capability (line 1.6).[3] Notice that this algorithm takes as input the maximum number n of clients in the system, since this determines the size of the predicates ruling access control, which the predicate(-only) encryption schemes are parameterized by.

Implementation of $\{cap_i, \mathsf{deny}\} \leftarrow \mathsf{addCl}(cap_{\mathcal{O}}, \mathbf{a})$ **(Algorithm 2).** This algorithm allows $\mathcal{O}$ to register a new client in the system. Specifically, $\mathcal{O}$ creates a new capability for the new client $\mathcal{C}_i$ according to the given access permission list $\mathbf{a}$ (lines 2.5–2.8). If $\mathcal{O}$ wants to add more clients than n, the maximum

[3]For simplifying the notation, we assume for each encryption scheme that the public key is part of the secret key.

Algorithm 2 $\{cap_i, \mathsf{deny}\} \leftarrow \mathsf{addCl}(cap_{\mathcal{O}}, \mathbf{a})$.

```
Input: the capability of $\mathcal{O}$ $cap_{\mathcal{O}}$ and an access control list $\mathbf{a}$ for the client to be added
Output: a capability $cap_i$ for client $\mathcal{C}_i$ in case of success, deny otherwise
1: parse $cap_{\mathcal{O}}$ as $(cnt_{\mathcal{C}}, cnt_E, sk, osk, psk)$
2: if $|\mathbf{a}| \neq cnt_E$ then return deny
3: end if
4: $cnt_{\mathcal{C}} := cnt_{\mathcal{C}} + 1$
5: compute $f_i$ s.t. the following holds for $1 \leq j \leq |\mathbf{a}|$ and all $(x_{w,j}, x_{r,j}) := \mathcal{ADB}(j)$
       if $\mathbf{a}(j) = \perp$ then $f_i(x_{w,j}) = f_i(x_{r,j}) = 0$
       if $\mathbf{a}(j) = r$ then $f_i(x_{w,j}) = 0$ and $f_i(x_{r,j}) = 1$
       if $\mathbf{a}(j) = rw$ then $f_i(x_{w,j}) = f_i(x_{r,j}) = 1$
6: $\mathcal{ADB} := \mathcal{ADB}[\mathcal{C}_i \mapsto f_i]$
7: $osk_{f_i} \leftarrow \mathsf{PoKGen}(osk, f_i)$, $sk_{f_i} \leftarrow \mathsf{PrKGen}(psk, f_i)$
8: return $cap_i := (sk, osk_{f_i}, sk_{f_i})$
```

Algorithm 3 $\{\mathsf{DB}', \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{addE}}(cap_{\mathcal{O}}, \mathbf{a}, d), \mathcal{S}_{\mathsf{addE}}(\mathsf{DB})\rangle$.

```
Input: the capability of $\mathcal{O}$ $cap_{\mathcal{O}}$, an access control list $\mathbf{a}$ and the data $d$ for the entry to be added
Output: a changed database $\mathsf{DB}'$ on $\mathcal{S}$ in case of success, deny otherwise
1: parse $cap_{\mathcal{O}}$ as $(cnt_{\mathcal{C}}, cnt_E, sk, osk, psk)$
2: if $|\mathbf{a}| \neq cnt_{\mathcal{C}}$ then return deny
3: end if
4: $cnt_E := cnt_E + 1$, $j := cnt_E$, $l_j \leftarrow \{0,1\}^D$, $\mathcal{LM} := \mathcal{LM}[j \mapsto l_j]$
5: let $E_1, \ldots, E_{b(D+1)}$ be the path from $\rho$ to $T_{D,l_j}$ downloaded from $\mathcal{S}$ ($E_i = (c_{1,i}, c_{2,i}, c_{3,i}, c_{4,i})$)
6: let $k$ be such that $\mathsf{Dec}(sk, c_{1,k}) > |\mathsf{DB}|$
7: compute $(x_{w,j}, x_{r,j})$ s.t. the following holds for $1 \leq i \leq |\mathbf{a}|$ and all $f_i := \mathcal{ADB}(\mathcal{C}_i)$
       if $\mathbf{a}(i) = \perp$ then $f_i(x_{w,j}) = f_i(x_{r,j}) = 0$
       if $\mathbf{a}(i) = r$ then $f_i(x_{w,j}) = 0$, $f_i(x_{r,j}) = 1$
       if $\mathbf{a}(i) = rw$ then $f_i(x_{w,j}) = f_i(x_{r,j}) = 1$
8: $\mathcal{ADB} := \mathcal{ADB}[j \mapsto (x_{w,j}, x_{r,j})]$
9: $E_k := (c_{1,k}, c_{2,k}, c_{3,k}, c_{4,k})$ where
       $k_j \leftarrow \mathsf{Gen}_{\mathsf{SE}}(1^\lambda)$                        $c_{1,k} \leftarrow \mathsf{Enc}(pk, j)$
       $c^j_{\mathsf{Auth}} \leftarrow \mathsf{PoEnc}(opk, x_{w,j})$          $c_{2,k} \leftarrow \mathsf{Enc}(pk, c^j_{\mathsf{Auth}})$
       $c^j_{\mathsf{Key}} \leftarrow \mathsf{PrEnc}(mpk, x_{r,j}, k_j)$     $c_{3,k} \leftarrow \mathsf{Enc}(pk, c^j_{\mathsf{Key}})$
       $c^j_{\mathsf{Data}} \leftarrow \mathcal{E}(k_j, d)$                      $c_{4,k} \leftarrow \mathsf{Enc}(pk, c^j_{\mathsf{Data}})$
10: for all $1 \leq \ell \leq b(D+1)$, $\ell \neq k$ do
11:     select $r_\ell$ uniformly at random
12:     $E'_\ell \leftarrow \mathsf{Rerand}(pk, E_\ell, r_\ell)$
13: end for
14: upload $E'_1, \ldots, E'_{k-1}, E_k, E'_{k+1}, \ldots, E'_{b(D+1)}$ to $\mathcal{S}$
```

number she initially decided, she can do so at the price of re-initializing the database. In particular, she has to setup new predicate-and predicate-only encryption schemes, since these depend on n. Secondly, she has to distribute new capabilities to all clients. Finally, for each entry in the database, she has to re-encrypt the ciphertexts c_{Auth} and c_{Key} with the new keys.

Implementation of $\{\mathsf{DB}', \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{addE}}(cap_{\mathcal{O}}, \mathbf{a}, d), \mathcal{S}_{\mathsf{addE}}(\mathsf{DB})\rangle$ **(Algorithm 3).** In this algorithm, $\mathcal{O}$ adds a new entry that contains the payload d to the database. Furthermore,

Algorithm 4 $(E''_1, \ldots, E''_{b(D+1)}, \pi, [P]) \leftarrow \mathsf{Evict}(E_1, \ldots, E_{b(D+1)}, s, j, k)$.

```
Input: a list of entries $E_1, \ldots, E_{b(D+1)}$, a bit $s$, an index $j$, and a position $k$ in the list
Output: a permuted and rerandomized list of entries $E''_1, \ldots, E''_{b(D+1)}$, a permutation $\pi$, and a proof of shuffle correctness (if $s = 1$)
1: $l_j \leftarrow \{0,1\}^D$, $\mathcal{LM} := \mathcal{LM}[j \mapsto l_j]$
2: compute a permutation $\pi$ s.t. $\pi(k) = 1$ and for all other $\ell \neq k$, $\pi$ pushes $\ell$ down on the path from $\rho$ $(= E_1, \ldots, E_b)$ to the current leaf node $(= E_{bD+1}, \ldots, E_{b(D+1)})$ as long as the index of the $\ell$-th entry still lies on the path from $\rho$ to its designated leaf node.
3: $E'_1, \ldots, E'_{b(D+1)} := E_{\pi^{-1}(1)}, \ldots, E_{\pi^{-1}(b(D+1))}$
4: let $E''_1, \ldots, E''_{b(D+1)}$ be the rerandomization of $E'_1, \ldots, E'_{b(D+1)}$ as described in 3.10–3.13 (including $k$)
5: if $s = 1$ then
6:     $P := PK\left\{ \begin{array}{l} (\pi, r_1, \ldots, r_{b(D+1)}) : \\ \quad \forall \ell.\ E_\ell = \mathsf{Rerand}(pk, E_{\pi^{-1}(\ell)}, r_\ell) \end{array} \right\}$
7:     return $E''_1, \ldots, E''_{b(D+1)}, \pi, P$
8: else
9:     return $E''_1, \ldots, E''_{b(D+1)}, \pi$
10: end if
```

the new entry is protected according to the given access permission list $\mathbf{a}$. Intuitively, $\mathcal{O}$ assigns the new entry to a random leaf and downloads the corresponding path in the database (lines 3.4–3.5). It then creates the new entry and substitutes it for a dummy entry (lines 3.6–3.9). Finally, $\mathcal{O}$ rerandomizes the entries so as to hide from $\mathcal{S}$ which entry changes, and finally uploads the modified path to $\mathcal{S}$ (lines 3.10–3.14).

Eviction. In all ORAM constructions, the client has to rearrange the entries in the database in order to make subsequent accesses unlinkable to each other. In the tree construction we use [4], this is achieved by first assigning a new, randomly picked, leaf index to the read or written entry. After that, the entry might no longer reside on the path from the root to its designated leaf index and, thus, has to be moved. This procedure is called *eviction* (Algorithm 4).

This algorithm assigns the entry to be evicted to a new leaf index (line 4.1). It then locally shuffles and rerandomizes the given path according to a permutation π (lines 4.2–4.4). After replacing the old path with a new one, the evicted entry is supposed to be stored in a node along the path from the root to the assigned leaf, which always exists since the root is part of the permuted nodes. A peculiarity of our setting is that clients are not trusted and, in particular, they might store a sequence of ciphertexts in the database that is not a permutation of the original path (e.g., they could store a path of dummy entries, thereby cancelling the original data).

Integrity proofs. To tackle this problem, a first technical novelty in our construction is, in the read and write protocols, to let the client output the modified path along with a proof of shuffle correctness [18], [7], which has to be verified by the server ($s = 1$, lines 4.6–4.7). As the data owner is assumed to be honest, she does not have to send a proof in the chMode protocol ($s = 0$, line 4.9).

Algorithm 5 $\langle \mathcal{C}_{\mathsf{chMode}}(cap_{\mathcal{O}}, \mathbf{a}, j), \mathcal{S}_{\mathsf{chMode}}(\mathsf{DB})\rangle$.

```
Input: the capability of O $cap_{\mathcal{O}}$, an access control list $\mathbf{a}$, and an index $j$
Output: deny if the algorithm fails
 1: parse $cap_{\mathcal{O}}$ as $(cnt_{\mathcal{C}}, cnt_E, sk, osk, psk)$
 2: if $|\mathbf{a}| \neq cnt_{\mathcal{C}}$ or $j > cnt_E$ then return deny
 3: end if
 4: $l_j := \mathcal{LM}(j)$
 5: let $E_1, \ldots, E_{b(D+1)}$ be the path from $\rho$ to $T_{D,l_j}$ downloaded from $\mathcal{S}$ ($E_i = (c_{1,i}, c_{2,i}, c_{3,i}, c_{4,i})$)
 6: let $k$ be s.t. $\mathsf{Dec}(sk, c_{1,k}) = j$
 7: compute $(x_{w,j}, x_{r,j})$ according to 3.7 and subject to all $f_i$ in $\mathcal{ADB}$, also add them to $\mathcal{ADB}$ (3.8)
 8: $(x'_{w,j}, x'_{r,j}) := \mathcal{ADB}(j)$
 9: let $f$ be s.t. $f(x'_{w,j}) = f(x'_{r,j}) = 1$
        $sk_f \leftarrow \mathsf{PrKGen}(psk, f)$    $c^j_{\mathsf{Key}} \leftarrow \mathsf{Dec}(sk, c_{3,k})$
10:     $k'_j \leftarrow \mathsf{PrDec}(sk_f, c^j_{\mathsf{Key}})$    $c^j_{\mathsf{Data}} \leftarrow \mathsf{Dec}(sk, c_{4,k})$
        $d \leftarrow \mathcal{D}(k'_j, c^j_{\mathsf{Data}})$
11: compute $E'_k$ as in 3.9
12: $(E''_1, \ldots, E''_{b(D+1)}, \pi) := \mathsf{Evict}(E_1, \ldots, E_{k-1}, E'_k, E_{k+1}, \ldots, E_{b(D+1)}, 0, j, k)$
13: upload $E''_1, \ldots, E''_{b(D+1)}$ to $\mathcal{S}$
```

Implementation of $\langle \mathcal{C}_{\mathsf{chMode}}(cap_{\mathcal{O}}, \mathbf{a}, j), \mathcal{S}_{\mathsf{chMode}}(\mathsf{DB})\rangle$ **(Algorithm 5).** In this protocol, $\mathcal{O}$ changes the access mode of the j-th entry in DB according to the new access permission list $\mathbf{a}$. Intuitively, she does so by downloading the path where the entry resides on (lines 5.4–5.5), changing the entry accordingly (lines 5.6–5.11), and uploading a modified and evicted path to the server (lines 5.12–5.13). Naïvely, $\mathcal{O}$ could simply re-encrypt the old key with the new attributes. However, if a client keeps a key for index j locally and his access on j is revoked, then he can still access the payload. Hence, $\mathcal{O}$ also picks a new key and re-encrypts the payload.

Implementation of $\{d, \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{read}}(cap_i, j), \mathcal{S}_{\mathsf{read}}(\mathsf{DB})\rangle$ **(Algorithm 6).** Intuitively, the client downloads the path which index j is assigned to and searches for the corresponding entry (lines 6.4–6.6). She then evicts the downloaded path, subject to the restriction that some dummy entry afterwards resides in the top position of the root node (lines 6.7–6.8). $\mathcal{C}$ uploads the evicted path together with a proof of shuffle correctness to $\mathcal{S}$ who verifies the proof and replaces the old with the new path in case of successful verification (line 6.9).

Obliviousness in presence of integrity proofs. $\mathcal{C}$ could in principle stop here since she has read the desired entry. However, in order to fulfill the notion of obliviousness, the read and write operations must be indistinguishable. In single-client ORAM constructions, $\mathcal{C}$ can make write indistinguishable from read by simply modifying the content of the desired entry before uploading the shuffled path to the server. This approach does not work in our setting, due to the presence of integrity proofs. Intuitively, in read, it would suffice to produce a proof of shuffle correctness, but this proof would not be the same as the one used in write, where one element in the path changes. Hence another technical novelty in our construction is the last part of the read protocol (lines 6.10–6.14), which "simulates" the write protocol despite the presence of integrity proofs. This is explained below, in the context of the write protocol.

Algorithm 6 $\{d, \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{read}}(cap_i, j), \mathcal{S}_{\mathsf{read}}(\mathsf{DB})\rangle$.

```
Input: the capability of the client executing the protocol $cap_i$ and the index $j$ to be read
Output: the data payload $d$ in case of success, deny otherwise
 1: parse $cap_i$ as $(sk, osk_f, sk_f)$
 2: if $j > |\mathsf{DB}|$ then return deny
 3: end if
 4: $l_j := \mathcal{LM}(j)$
 5: let $E_1, \ldots, E_{b(D+1)}$ and $k$ be as in lines 5.5–5.6
 6: extract $d$ from $E_k$ as in line 5.10
 7: let $\ell$ be s.t. $\mathsf{Dec}(sk, c_{1,\ell}) > |\mathsf{DB}|$
 8: $(E''_1, \ldots, E''_{b(D+1)}, \pi, P) := \mathsf{Evict}(E_1, \ldots, E_{b(D+1)}, 1, j, \ell)$
 9: upload $E''_1, \ldots, E''_{b(D+1)}$ and $P$ to $\mathcal{S}$
10: $c^j_{\mathsf{Auth}} \leftarrow \mathsf{Dec}(sk, c''_{2,1})$
11: $P_{\mathsf{Auth}} := PK\left\{(osk_f) : \mathsf{PoDec}(osk_f, c^j_{\mathsf{Auth}}) = 1\right\}$
12: $E'''_1 := (c'''_{1,1}, c'''_{2,1}, c'''_{3,1}, c'''_{4,1})$ where
        $r_1, \ldots, r_4$ are selected uniformly at random
        $c'''_{l,1} \leftarrow \mathsf{Rerand}(pk, c''_{l,1}, r_l)$ for $l \in \{1, 3, 4\}$
        $c'''_{2,1} \leftarrow \mathsf{Enc}(pk, \mathsf{PoRR}(opk, c^j_{\mathsf{Auth}}, r_2))$
13: $P_{\mathsf{Ind}} := PK\left\{(r_1) : c''_{1,1} = \mathsf{Rerand}(pk, c'''_{1,1}, r_1)\right\}$
14: upload $E'''_1$, $P_{\mathsf{Auth}}$, $P_{\mathsf{Ind}}$, and the necessary information to access $c^j_{\mathsf{Auth}}$ to $\mathcal{S}$
```

Algorithm 7 $\{\mathsf{DB}', \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{write}}(cap_i, j, d), \mathcal{S}_{\mathsf{write}}(\mathsf{DB})\rangle$.

```
Input: the capability of the client executing the protocol $cap_i$, the index $j$ to be written, and the payload $d$
Output: deny if the algorithm fails
 1: execute 6.1–6.6
 8: $(E''_1, \ldots, E''_{b(D+1)}, \pi, P) := \mathsf{Evict}(E_1, \ldots, E_{b(D+1)}, 1, j, k)$
 9: execute 6.9–6.11
12: $E'''_1 := (c'''_{1,1}, c'''_{2,1}, c'''_{3,1}, c'''_{4,1})$ where
        $r_1, r_2, r_3$ are selected uniformly at random
        $c'''_{1,1} \leftarrow \mathsf{Rerand}(pk, c''_{1,1}, r_1)$
        $c'''_{2,1} \leftarrow \mathsf{Enc}(pk, \mathsf{PoRR}(opk, c^j_{\mathsf{Auth}}, r_2))$
        $c'''_{3,1} \leftarrow \mathsf{Enc}(pk, \mathsf{PrRR}(mpk, c^j_{\mathsf{Key}}, r_3))$
        $c^j_{\mathsf{Data}} \leftarrow \mathcal{E}(k_j, d)$
        $c'''_{4,1} \leftarrow \mathsf{Enc}(pk, c^j_{\mathsf{Data}})$
13: execute 6.13–6.14
```

Implementation of $\{\mathsf{DB}', \mathsf{deny}\} \leftarrow \langle \mathcal{C}_{\mathsf{write}}(cap_i, j, d), \mathcal{S}_{\mathsf{write}}(\mathsf{DB})\rangle$ **(Algorithm 7).** Firstly, $\mathcal{C}$ reads the element that she wishes to change (line 7.1). Secondly, $\mathcal{C}$ evicts the path with the difference that here the first entry in the root node is the element that $\mathcal{C}$ wants to change, as opposed to a dummy entry like in read (line 7.8). It is important to observe that the shuffle proof sent to the server (line 4.6) is indistinguishable in read and write since it hides both the permutation and the randomness used to rerandomize the entries. So far, we have shown how $\mathcal{C}$ can upload a shuffled and rerandomized path to the server without modifying the content of any entry.

In write, $\mathcal{C}$ can now replace the first entry in the root node with the entry containing the new payload (lines 7.12–7.13).

In read, this step is simulated by rerandomizing the first entry of the root node, which is a dummy entry (line 6.12).

The integrity proofs P_{Auth} and P_{Ind} produced in read and write are indistinguishable (lines 6.11 and 6.13 for both): in both cases, they prove that $\mathcal{C}$ has the permission to write on the first entry of the root node and that the index has not changed. Notice that this proof can be produced also in read, since all clients have write access to dummy entries.

Permanent Entries. Some application scenarios of GORAM might require determined entries of the database not to be modifiable nor deletable, not even by the data owner herself (for instance, in the case of PHRs, the user should not be able to cancel diagnostic results in order to pay lower insurance fees). Even though we did not explicitly describe the construction, we mention that such a property can be achieved by assigning a binary attribute (*modifiable* or *permanent*) to each entry and storing a commitment to this in the database. Every party that tries to modify a given entry, including the data owner, has to provide a proof that the respective attribute is set to *modifiable*. Due to space constraints we omit the algorithm, but it can be efficiently instantiated using El Gamal encryption and Σ-protocols.

C. Batched Zero-Knowledge Proofs of Shuffle

A zero-knowledge proof of a shuffle of a set of ciphertexts proves in zero-knowledge that a new set of ciphertexts contains the same plaintexts in permuted order. In our system the encryption of an entry, for reasonable block sizes, yields in practice hundreds of ciphertexts, which means that we have to perform hundreds of shuffle proofs. These are computable in polynomial-time but, even using the most efficient known solutions (e.g., [7], [19]), not fast enough for practical purposes. This problem has been addressed in the literature but the known solutions typically reveal part of the permutation (e.g., [20]), which would break obliviousness and, thus, are not applicable in our setting.

To solve this problem we introduce a new proof technique that we call *batched zero-knowledge proofs of shuffle*, based on the idea of "batching" several instances and verifying them together. Our interactive protocol takes advantage of the homomorphic property of the top layer public-key encryption scheme in order to batch the instances. On a high level, we represent the path, which the client proves the shuffle of, as an n-by-m matrix where n is the number of entries (i.e., the path length) and m is the number of blocks of ciphertexts per entry. The common inputs of the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ are the two matrices $\mathbf{A}$ and $\mathbf{A}'$ characterizing the path stored on the database and the path shuffled and re-randomized by the client, respectively. $\mathcal{P}$ additionally knows the permutation π and the randomnesses $\mathbf{R}$ used for rerandomizing the entries.

Intuitively, the batching algorithm randomly selects a subset of columns (i.e., block indices) and computes the row-wise product of the corresponding blocks for each row. It then computes the proof of shuffle correctness on the resulting single-block ciphertexts. The property we would like to achieve is that modifying even a single block in a row should lead to a different product and, thus, be detected. Notice that naïvely multiplying all blocks together does not achieve the intended property, as illustrated by the following counterexample:

$$\begin{pmatrix} \mathsf{Enc}(pk,3) & \mathsf{Enc}(pk,4) \\ \mathsf{Enc}(pk,5) & \mathsf{Enc}(pk,2) \end{pmatrix} \qquad \begin{pmatrix} \mathsf{Enc}(pk,2) & \mathsf{Enc}(pk,6) \\ \mathsf{Enc}(pk,5) & \mathsf{Enc}(pk,2) \end{pmatrix}$$

In the above matrices, the rows have not been permuted but rather changed. Still, the row-wise product is preserved, i.e., 12 in the first and 10 in the second. Hence, we cannot compute the product over all columns. Instead, as proved in the long version, the intended property can be achieved with probability at least $\frac{1}{2}$ if each column is included in the product with probability $\frac{1}{2}$. Although a probability of $\frac{1}{2}$ is not sufficient in practice, repeating the protocol k times increases the probability to $(1 - \frac{1}{2^k})$.

Algorithm 8 Batched ZK Proofs of Shuffle.

```
Input of 𝒫: A, A′, π, R
Input of 𝒱: A, A′
1: $\mathcal{V}$ randomly selects $\vec{a} \leftarrow \{0,1\}^m$ and sends it to $\mathcal{P}$.
2: $\mathcal{P}$ computes for all $1 \le i \le n$ the partial ciphertext products
       $\theta_i = \prod_{j=1}^{m} a_j \mathbf{A}_{i,j}$ and $\theta'_i = \prod_{j=1}^{m} a_j \mathbf{A}'_{i,j}$
   and the corresponding partial randomness sum
       $r_i = \sum_{j=1}^{m} a_j \mathbf{R}_{i,j}$
   where $a_j$ is the $j$-th bit of $\vec{a}$. $\mathcal{V}$ also computes $\vec{\theta}$ and $\vec{\theta}'$.
3: $\mathcal{V}$ and $\mathcal{P}$ run the protocol for the proof of shuffle correctness [7] on $\vec{\theta}$, $\vec{\theta}'$, $\pi$, and $\vec{r}$.
```

The detailed construction is depicted in Algorithm 8. In line 8.1, $\mathcal{V}$ picks a challenge, which indicates which column to include in the homomorphic product. Upon receiving the challenge, in line 8.2, $\mathcal{P}$ and $\mathcal{V}$ compute the row-wise multiplication of the columns indicated by the challenge. Finally, $\mathcal{V}$ and $\mathcal{P}$ run an off-the-shelf shuffle proof on the resulting ciphertext lists (line 8.3).

It follows from the protocol design that our approach does not affect the completeness of the underlying proof of shuffle correctness, the same holds true for the zero-knowledge and proof of knowledge properties. Furthermore, any malicious prover who does not apply a correct permutation is detected by the verifier with probability at least $1/2$.

Finally, the protocol can be made non-interactive by using the Fiat-Shamir heuristic [21].

To summarize, our new approach preserves all of the properties of the underlying shuffle proof while being significantly more efficient. Our proof system eliminates the dependency of the number of proofs with respect to the block size, making it dependent only on k and on the complexity of the proof itself.

IV. Accountable Integrity (A-GORAM)

In this section we relax the integrity property by introducing the concept of accountability. In particular, instead of letting the server check the correctness of client operations, we develop a technique that allows clients to detect *a posteriori* non-authorized changes on the database and blame the misbehaving party. Intuitively, each entry is accompanied by a tag (technically, a chameleon hash along with the randomness corresponding to that entry), which can only be produced by clients having write access. All clients can verify the validity of such tags and, eventually, determine which client inserted an entry with an invalid tag. This makes the construction more efficient and scalable, significantly reducing the computational complexity both on the client and on the server side, since zero-knowledge proofs are no longer necessary and, consequently, the outermost encryption can be implemented using symmetric, as opposed to asymmetric, cryptography. Such a mechanism is supposed to be paired with a data versioning protocol in order to avoid data losses: as soon as one of the clients detects an invalid entry, the misbehaving party is punished and the database is reverted to the last safe state (i.e., a state where all entries are associated with a valid tag).

A. Prerequisites

In the following, we review some additional cryptographic primitives and explain the structure of the log file.

Cryptographic preliminaries. Intuitively, a chameleon hash function is a randomized collision-resistant hash function that provides a trapdoor. Given the trapdoor it is possible to efficiently compute collisions. A chameleon hash function is a tuple of PPT algorithms $\Pi_{\mathsf{CHF}} = (\mathsf{Gen}_{\mathsf{CHF}}, \mathsf{CH}, \mathsf{Col})$. The setup algorithm $\mathsf{Gen}_{\mathsf{CHF}}$ takes as input a security parameter 1^λ and outputs a key pair (cpk, csk), where cpk is the public key and csk is the secret key. The chameleon hash function CH takes as input the public key cpk, a message m, and randomness r; it outputs a hash tag t. The collision function Col takes as input the secret key csk, a message m, randomness r, and another message m'; it outputs a new randomness r' such that $\mathsf{CH}(cpk, m, r) = \mathsf{CH}(cpk, m', r')$. For our construction, we use chameleon hash functions providing key-exposure freeness [22]. Intuitively, this property states that no adversary is able to find a fresh collision, without knowing the secret key csk, even after seeing polynomially many collisions.

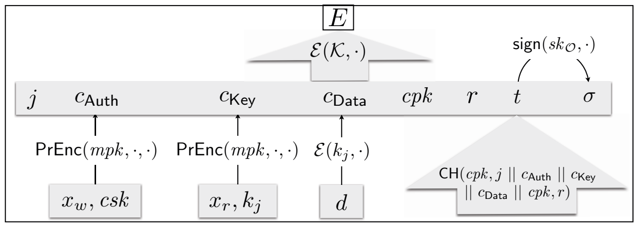
We denote by $\Pi_{\mathsf{DS}} = (\mathsf{Gen}_{\mathsf{DS}}, \mathsf{sign}, \mathsf{verify})$ a digital signature scheme. We require a signature scheme that is existentially unforgeable. Intuitively, this notion ensures that it is infeasible for any adversary to output a forgery (i.e., a fresh signature σ on a message m without knowing the signing key) even after seeing polynomially many valid (σ, m) pairs.

Structure of the log file. We use a log file Log so as to detect who has to be held accountable in case of misbehavior. Log is append-only and consists of the list of paths uploaded to the server, each of them signed by the respective client.

B. Construction

Structure of entries. The structure of an entry in the database is depicted in Figure 2. An entry E is protected by a top-level private-key encryption scheme with a key $\mathcal{K}$ that is shared by the data owner $\mathcal{O}$ and all clients $\mathcal{C}_1, \ldots, \mathcal{C}_n$. Under the encryption, E contains several elements, which we explain below:

- j is the index of the entry;
- c_{Auth} is a predicate encryption ciphertext that encrypts the private key csk of a chameleon hash function under an attribute x_w, which regulates the write access;
- c_{Key} and c_{Data} are unchanged;
- cpk is the public key of a chameleon hash function, i.e., the counterpart of csk encrypted in c_{Auth};


E
$\mathcal{E}(\mathcal{K}, \cdot)$
$\mathsf{sign}(sk_{\mathcal{O}}, \cdot)$
j c_{Auth} c_{Key} c_{Data} cpk r t σ
$\mathsf{PrEnc}(mpk, \cdot, \cdot)$
$\mathsf{PrEnc}(mpk, \cdot, \cdot)$
$\mathcal{E}(k_j, \cdot)$
$\mathsf{CH}(cpk, j \mid\mid c_{\mathsf{Auth}} \mid\mid c_{\mathsf{Key}} \mid\mid c_{\mathsf{Data}} \mid\mid cpk, r)$
x_w, csk
x_r, k_j
d
Figure 2. The structure of an entry in the database.

- r is some randomness used in the computation of t;
- t is a chameleon hash tag, produced by hashing the concatenation of j, c_{Auth}, c_{Key}, c_{Data}, and cpk under randomness r using the public key cpk;
- σ is a signature on the chameleon hash tag t, signed by the data owner $\mathcal{O}$.

Intuitively, only clients with write access are able to decrypt c_{Auth}, and thus to retrieve the key csk required to compute a collision for the new entry d' (i.e., to find a randomness r' such that the chameleon hash t for the old entry d and randomness r is the same as the one for d' and r'). The fundamental observation is that the modification of an entry is performed without changing the respective tag. Consequently, the signature σ is the same for the old and for the new entry. Computing a collision is the only way to make the tag t, originally signed by the data owner, a valid tag also for the new entry d'. Therefore verifying the signature and the chameleon hash suffices to make sure that the entry has been only modified by authorized clients.

Basic Algorithms. The basic algorithms follow the ones defined in Section III-B, except for natural adaptions to the new entry structure. Furthermore, the zero-knowledge proofs are no longer computed and the rerandomization steps are substituted by re-encryptions. Finally, clients upload on the server signed paths, which are stored in the Log.

Entry Verification. We introduce an auxiliary verification function that clients run in order to verify the integrity of an entry. During the execution of any protocol below we maintain the invariant that, whenever a client i (or the data owner himself) parses an entry j that he downloaded from the server, he executes Algorithm 9. If the result is $\bot$, then the client runs $\mathsf{blame}(cap_i, \mathsf{Log}, j)$.

Algorithm 9 The pseudo-code for the verification of an entry in the database which is already decrypted.

```
Input: An entry $(j, c_{\mathsf{Auth}}, c_{\mathsf{Key}}, c_{\mathsf{Data}}, r, cpk, t, \sigma)$ and the verification key $vk_{\mathcal{O}}$ of $\mathcal{O}$.
Output: $\top$ if verification succeeds, $\bot$ otherwise.
1: if $t = \mathsf{CH}(cpk, j \mid\mid c_{\mathsf{Auth}} \mid\mid c_{\mathsf{Key}} \mid\mid c_{\mathsf{Data}} \mid\mid cpk, r)$ and $\top = \mathsf{verify}(\sigma, vk_{\mathcal{O}}, t)$ then
2:     return $\top$
3: else
4:     return $\bot$
5: end if
```

Blame. In order to execute the function $\mathsf{blame}(cap_i, \mathsf{Log}, j)$, the client must first retrieve Log from the server. Afterwards, she parses backwards the history of modifications by decrypting the paths present in the Log. The client stops only when she finds the desired entry indexed by j in a consistent state, i.e., the data hashes to the associated tag t and the signature is valid. At this point the client moves forwards on the Log until she finds an uploaded path where the entry j is supposed to lay on (the entry might be associated with an invalid tag or missing). The signature on the path uniquely identifies the client, whose identity is added to a list L of misbehaving clients. Finally, all of the other clients that acknowledged the changes of the inconsistent entry are also added to L, since they did not correctly verify its chameleon signature.

Discussion. As explained above, the accountability mechanism allows for the identification of misbehaving clients with a minimal computational overhead in the regular clients' operation. However, it requires the server to store a log that is linear in the number of modifications to the database and logarithmic in the number of entries. This is required to revert the database to a safe state in case of misbehaviour. Consequently, the blame algorithm results expensive in terms of computation and communication with the server, in particular for the entries that are not regularly accessed. Nonetheless, blame is supposed to be only occasionally executed, therefore we believe this design is acceptable in terms of service usability. Furthermore, we can require all the parties accessing the database to synchronize on a regular basis so as to verify the content of the whole database and to reset the Log, in order to reduce the storage on the server side and, thus, the amount of data to transfer in the blame algorithm. Such an approach could be complemented by an efficient versioning algorithm on encrypted data, which is however beyond the scope of this work and left as a future work. Finally, we also point out that the accountable-integrity property, as presented in this section, sacrifices the anonymity property, since users have to sign the paths they upload to the server. This issue can be easily overcome by using any anonymous credential system that supports revocation [23].

V. Scalable Solution (S-GORAM)

Even though the personal record management systems we consider rely on simple client-based read and write permissions, the predicate encryption scheme used in GORAM and A-GORAM support in principle a much richer class of access control policies, such as role-based access control (RBAC) or attribute-based access control (ABAC) [5]. If we stick to client-based read and write permissions, however, we can achieve a more efficient construction that scales to thousands of clients. To this end, we replace the predicate encryption scheme with a broadcast encryption scheme [24], which guarantees that a specific subset of clients is able to decrypt a given ciphertext. This choice affects the entry structure as follows (cf. Figure 2):

- c_{Key} is the broadcast encryption of k_j;
- c_{Auth} is the broadcast encryption of csk.

The subset of clients that can decrypt c_{Key} (resp. c_{Auth}) is then set to be the same subset that holds *read* (resp. *write*) permissions on the given entry. By applying the aforementioned modifications on top of A-GORAM, we obtain a much more efficient and scalable instantiation, called S-GORAM, that achieves a smaller constant in the computational complexity (linear in the number of clients). For more details on the performance evaluation and a comparison with A-GORAM, we refer to Section VIII.

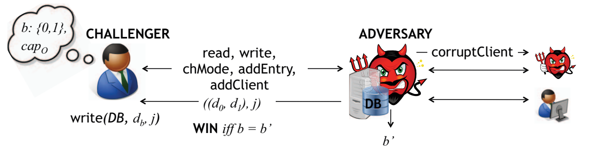


Figure 3. Game for Secrecy.

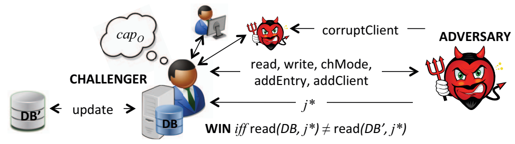

cap_O
CHALLENGER
corruptClient
ADVERSARY
read, write, chMode, addEntry, addClient
j*
DB'
update
DB
WIN iff read(DB, j*) ≠ read(DB', j*)

Figure 4. Game for Integrity.

VI. Security and Privacy for Group ORAM

In order to prove the security of our constructions, we formalized the security and privacy properties of a Group ORAM by cryptographic games, which are intuitively introduced below. The formal definitions can be found in the appendix.

A. Security and Privacy of Group ORAM

Secrecy. Intuitively, a Group ORAM preserves the secrecy of outsourced data if no party is able to deduce any information about the content of any entry she does not have access to. We formalize this intuition through a cryptographic game, which is illustrated in Figure 3. Intuitively, the challenger initializes an empty database locally and it hands it over to the adversary so as to give him the possibility to adaptively and arbitrarily fill the content of the database. Additionally, the adversary is given the possibility of spawning and corrupting a polynomial number of clients, allowing him to perform operations on the database on their behalf. Hence, this property is proven in a strong adversarial model, without placing any assumption on the server's behavior. At some point of the game the adversary outputs two data and a database index, the challenger flips a coin and it randomly inserts either one of the two payloads in the desired database entry. In order to make the game not trivial, it must be the case that the adversary should not have corrupted any client that holds read permission on such index. We define the adversary to win the game if he correctly guesses which of the two entries has been written. Since the adversary can always randomly guess, we define the system to be *secrecy-preserving* if the adversary cannot win the game with probability non-negligibly greater than $\frac{1}{2}$.

Integrity. A Group ORAM preserves the integrity of its entries if none of the clients can modify an entry to which she does not have write permissions. The respective cryptographic game is depicted in Figure 4. Intuitively, the challenger initializes an empty database DB and a copy DB', providing the adversary with the necessary interfaces to fill the content of DB and to generate and corrupt clients. Every time the adversary queries an interface, the challenger interacts with the respective client playing the server's role and additionally executes locally the

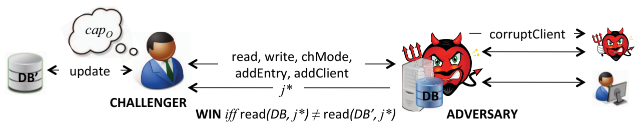

cap_O
DB'
update
CHALLENGER
read, write, chMode, addEntry, addClient
j^*
WIN *iff* read(*DB*, j^*) ≠ read(*DB'*, j^*)
corruptClient
DB
ADVERSARY

Figure 5. Game for Tamper-resistance.

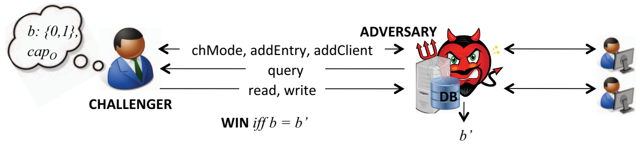


Figure 6. Game for Obliviousness.

same operation on DB' in an honest manner. Note that here the adversary cannot directly operate on the database but he can only operate through the clients: this constraint reflects the honesty assumption of the server. At some point of the execution, the adversary outputs an index of the database (that none of his corrupted clients can write on) and the challenger compares the two entries stored in DB and DB', if they are not the same we say that the adversary wins the game. Since that would imply that a client could potentially be able to modify the content of an entry she does not have access to, we say that the system is *integrity-preserving* if any possible adversary cannot win the game with non-negligible probability.

Tamper-resistance. Intuitively, a Group ORAM is tamper-resistant if the server, even colluding with a subset of malicious clients, is not able to convince an honest client about the integrity of some maliciously modified data. Notice that this property refers to a strong adversarial model, where the adversary may arbitrarily misbehave and collude with clients. Naturally, tamper-resistance holds true only for entries which none of the corrupted clients had ever access to. The respective cryptographic game is depicted in Figure 5. The game is described exactly as in the previous definition except for the fact that the database is this time handed over to the adversary at the very beginning of the experiment so as to allow him to operate directly on it. The challenger maintains a local copy of the database where it performs the same operations that are triggered by the adversary but in an honest way. The winning conditions for the adversary are the same as stated above and we say that the system is *tamper-resistant* if no adversary can win this game with probability greater than a negligible value. Note that there exists a class of attacks where the adversary wins the game by simply providing an old version of the database, which are inherent to the cloud storage setting. We advocate the usage of standard techniques to deal with this kind of attacks (e.g., a gossip protocol among the clients for versioning of the entries [25]) and hence, we rule them out in our formal analysis by implicitly assuming that the information provided by the adversary are relative to the most up to date version of the database that he possesses locally.

Obliviousness. Intuitively, a Group ORAM is oblivious if the server cannot distinguish between two arbitrary query sequences which contain read and write operations. In the cryptographic game depicted in Figure 6, the adversary holds the database on his side and he gets access to the interfaces needed to adaptively and arbitrarily insert content in the database. Thus the server may arbitrarily misbehave but it is not allowed to collude with clients: the adversary can only spawn a polynomial number of them, but he cannot corrupt them. In this game the challenger offers an additional interface where the adversary can input two arbitrary queries, i.e., on behalf of arbitrary clients and on arbitrary indices of the database. This interface can be used by the adversary polynomially many times, thus creating the two query sequences at the core of the obliviousness definition. In the beginning of the game the challenger flips a coin and then it always executes either one of the two queries, depending on the outcome of the initial random coin. In order to win, the adversary has to tell the value of the random coin of the challenger, thus distinguishing which query sequence has been executed. This would then mean that the adversary has been able to link some access to a specific memory location, hence we say that a system is *oblivious* if the adversary does not win the game with probability non-negligibly greater than $\frac{1}{2}$.

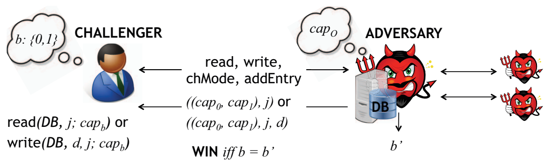

b: {0,1}
CHALLENGER
cap_O
ADVERSARY
read, write, chMode, addEntry
((cap_0, cap_1), j) or
((cap_0, cap_1), j, d)
read(*DB*, j; cap_b) or
write(*DB*, d, j; cap_b)
WIN *iff* $b = b'$
DB
b'

Figure 7. Game for Anonymity.

Anonymity. A Group ORAM is anonymity-preserving if the data owner cannot efficiently link a given operation to a client, among the set of clients having access to the queried index. In the cryptographic game depicted in Figure 7, the setting is equivalent to the secrecy definition except that the challenger also hands over the capability of the data owner to the adversary. Clearly the adversary does not need to corrupt clients since he can spawn them by himself. Additionally, the challenger provides the adversary with an interface that he can query with an operation associated with two arbitrary capabilities. To make the game not trivial, it must hold that both of the capabilities hold the same read and write permissions on the entry selected by the adversary. Based on some initial randomness, the challenger always executes the desired command with either one of the two capabilities and the adversary wins the game if and only if he can correctly determine which capability has been selected. Since this would imply a de-anonymization of the clients, we say that the system is *anonymity-preserving* if the adversary cannot win the game with probability non-negligibly greater than $\frac{1}{2}$.

Accountable Integrity. The server maintains an audit log Log, which holds the evidence of client operations on the database DB. Specifically, each path uploaded to the server as a result of an eviction procedure is signed by the client and appended by the server to the log. After detecting an invalid or missing entry with index j, the client retrieves Log from the server and performs the algorithm $\mathsf{blame}(cap_i, \mathsf{Log}, j)$. The output is a list of identities, which correspond to misbehaving parties.

We define the accountable integrity property through a cryptographic game, illustrated in Figure 8. The game is the same as the one for integrity, except for the winning condition,

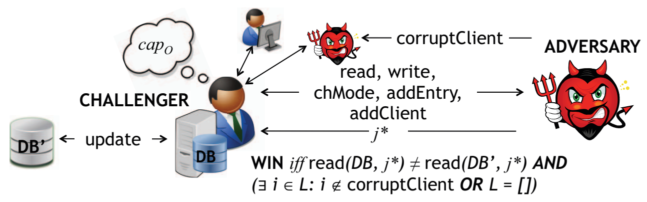

cap_O
CHALLENGER
DB'
update
DB
corruptClient
ADVERSARY
read, write, chMode, addEntry, addClient
j^*
WIN *iff* read(DB, j^*) $\neq$ read(DB', j^*) AND ($\exists$ $i \in L$: $i \notin$ corruptClient OR $L = []$)

Figure 8. Game for Accountable Integrity.

Property	Server	Collusion
Secrecy	malicious	✓
(Accountable) Integrity	HbC	✗
Tamper-resistance	malicious	✓
Obliviousness	malicious	✗
Anonymity	malicious	✓

Table I: Security and privacy properties together with their minimal assumptions.

which is adjusted according to the accountability requirements. Intuitively, the adversary wins the game if he manages to modify the entry in the index he provided and the challenger is not able to identify at least one of the corrupted clients that contributed to modify that entry or it erroneously blames some honest party. This means that the blaming procedure always returns at least one of the misbehaving parties and never an honest one. The literature defines this notion of accountability as *fairness* (never blame honest parties) and *completeness* (blame at least one dishonest party) [26]. We say that a system preserves *accountable integrity* if the adversary cannot win the game with more than negligible probability.

Discussion. Table I summarizes the security and privacy properties presented in this section, along with the corresponding assumptions. The HbC assumption is in fact only needed for integrity, since the correctness of client operations is checked by the server, thus avoiding costly operations on the client side. We will see in Section IV that the HbC assumption is still needed for the accountable integrity property, since the server maintains a log of accesses, which allows for blaming misbehaving parties. All other properties hold true even if the server is malicious as long as it does not collude with clients. Furthermore, secrecy, tamper-resistance, and anonymity hold true even if the server is malicious and colludes with clients. The non-collusion assumption is due to the obliviousness property, which is meant to protect the access patterns from the server. Extending this property to non-authorized clients and devising corresponding enforcement mechanisms is beyond the scope of this paper and left as an interesting future work.

VII. Security and Privacy Results

In this section, we show that the Group ORAM instantiations presented in Section III, in Section IV, and in Section V achieve the security and privacy properties stated in Section VI-A. The proofs are reported in the technical report [8]. A brief overview of the properties guaranteed by each construction is shown in Table II. As previously discussed, dropping the computationally expensive integrity checks in favor of an accountability mechanism is crucial to achieve efficiency. It follows that A-GORAM and S-GORAM provide accountable integrity as opposed to integrity and tamper resistance. Having an accountable system trivially implies the loss of anonymity, as defined in Section VI-A, although it is still possible to achieve pseudonym-based anonymity by employing anonymous credentials. The other privacy properties of our system, namely secrecy and obliviousness, are fulfilled by all of our instantiations. Moreover, by replacing predicate encryption with broadcast encryption (S-GORAM), we sacrifice the possibility to enforce ABAC policies, although we can still handle client-based read/write permissions.

Property	G.	A-G.	S-G.
Secrecy	✓	✓	✓
Integrity	✓	Accountable	Accountable
Tamper-resistance	✓	✗	✗
Obliviousness	✓	✓	✓
Anonymity	✓	✗	✗
Access control	ABAC	ABAC	R/W

Table II: Security and privacy properties achieved by each construction where G. stands for GORAM.

Before we present the security and privacy results, we start with a soundness result for the batched ZK proof of shuffle.

Theorem 1 (Soundness): Let $\mathcal{ZKP}$ be a zero-knowledge proof system for a proof of shuffle correctness. Then the batched ZK proof of shuffle defined in Algorithm 8 is sound with probability at least $1/2$.

The following theorems characterize the security and privacy properties achieved by each cryptographic instantiation presented in this paper.

*Theorem 2 (*GORAM*):* Let Π_{PE} and Π_{PO} be an attribute-hiding predicate and predicate-only encryption scheme, Π_{PKE} (resp. Π_{SE}) be a CPA-secure public-key (resp. private-key) encryption scheme, and $\mathcal{ZKP}$ be a zero-knowledge proof system. Then GORAM achieves *secrecy*, *integrity*, *tamper-resistance*, *obliviousness*, and *anonymity*.

*Theorem 3 (*A-GORAM*):* Let Π_{PE} be an attribute-hiding predicate encryption scheme, Π_{SE} be a CPA-secure private-key encryption scheme, Π_{DS} be an existentially unforgeable digital signature scheme, and Π_{CHF} be a collision-resistant, key-exposure free chameleon hash function. Then A-GORAM achieves *secrecy*, *accountable integrity*, and *obliviousness*.

*Theorem 4 (*S-GORAM*):* Let Π_{BE} be an adaptively secure broadcast encryption scheme, Π_{SE} be a CPA-secure private-key encryption scheme, Π_{DS} be an existentially unforgeable digital signature scheme, and Π_{CHF} be a collision-resistant, key-exposure free chameleon hash function. Then S-GORAM achieves *secrecy*, *accountable integrity*, and *obliviousness*.

VIII. Implementation and Experiments

In this section, we present the concrete instantiations of the cryptographic primitives that we previously described (Section VIII-A), we study their asymptotic complexity (Section VIII-B), describe our implementation (Section VIII-C), and discuss the experimental evaluation (Section VIII-D).

A. Cryptographic Instantiations

Private-key and public-key encryption. We use AES [27] as private-key encryption scheme with an appropriate message padding. Furthermore, we employ the El Gamal encryption scheme [28] for public-key encryption as it fulfills all properties that we require for GORAM, i.e., it is rerandomizable and supports zero-knowledge proofs.

Encryption schemes for access control. We utilize the predicate encryption scheme introduced by Katz *et al.* [5]. Its ciphertexts are rerandomizable and we also show them to be compatible with the Groth-Sahai proof system [6]. Concerning the implementation, the predicate encryption scheme by Katz *et al.* [5] is not efficient enough since it relies on elliptic curves on composite-order groups. In order to reach a high security parameter, the composite-order setting requires us to use much larger group sizes than in the prime-order setting, rendering the advantages of elliptic curves practically useless. Therefore, we use a scheme transformation proposed by David Freeman [29], which works in prime-order groups and is more efficient.

For implementing S-GORAM we use an adaptively secure broadcast encryption scheme by Gentry and Waters [24].

Zero-knowledge proofs. We deploy several non-interactive zero-knowledge proofs. For proving that a predicate-only ciphertext validly decrypts to 1 without revealing the key, we use Groth-Sahai non-interactive zero-knowledge proofs[4] [6]. More precisely, we apply them in the proofs created in line 6.11 (read and write, see Algorithm 6 and Algorithm 7). We employ plaintext-equivalence proofs (PEPs) [19], [30] for the proofs in line 6.13. Furthermore, we use a proof of shuffle correctness [7] and batched shuffle proofs in lines 6.8 and 7.8.

Chameleon hashes and digital signatures. We use a chameleon hash function by Nyberg and Rueppel [22], which has the key-exposure freeness property. We combine the chameleon hash tags with RSA signatures [31].

B. Computational Complexity

The computational and communication complexity of our constructions, for both the server and the client, is $O((B + G)\log N)$ where N is the number of the entries in the database, B is the block size of the entries in the database, and G is the number of clients that have access to the database. $O(B \log N)$ originates from the ORAM construction and we add $O(G \log N)$ for the access structure. Hence, our solution only adds a small overhead to the standard ORAM complexity. The client-side storage is $O(B \log N)$, while the server has to store $O(BN)$ many data.

C. Java Implementation

We implemented the four different versions of GORAM in Java (GORAM with off-the-shelf shuffle proofs and batched shuffle proofs, A-GORAM, and S-GORAM). Furthermore, we also implemented A-GORAM and S-GORAM on Amazon EC2. For zero-knowledge proofs, we build on a library [32] that implements Groth-Sahai proofs [6], which internally relies on jPBC/PBC [33], [34].

Cryptographic setup. We use MNT curves [35] based on prime-order groups for primes of length 224 bits. This results in 112 bits of security according to different organizations [36]. We deploy AES with 128 bit keys and we instantiate the El Gamal encryption scheme, the RSA signature scheme, and the chameleon hash function with a security parameter of 2048 bits. According to NIST [36], this setup is secure until 2030.

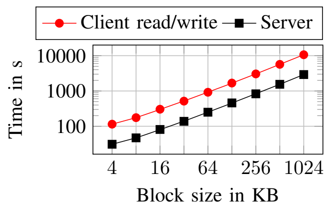
D. Experiments

We evaluated the four different implementations. As a first experiment, we measured the computation times on client and server for the read and write operation for the constructions without accountable integrity. We performed these experiments on an Intel Xeon with 8 cores and 2.60GHz in order to show the efficiency gained by using batched shuffle proofs instead of off-the-shelf zero-knowledge proofs of shuffle correctness. We vary different parameters: the database size from 1GB to 1TB, the block size from 4KB to 1MB, the number of clients from 1 to 10, the number of cores from 1 to 8, and for batched shuffle proofs also the number of iterations k from 1 to 128. We fix a bucket size of 4 since Stefanov *et al.* [4] showed that this value is sufficient to prevent buckets from overflowing.

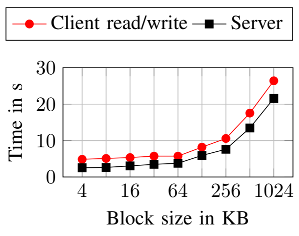
The second experiment focuses on the solution with accountability. Here we measure also the overhead introduced by our realization with respect to a state-of-the-art ORAM construction, i.e., the price we pay to achieve a wide range of security and privacy properties in a multi-client setting. Another difference from the first experiment is the hardware setup. We run the server side of the protocol in Amazon EC2 and the client side on a MacBook Pro with an Intel i7 and 2.90GHz. We vary the parameters as in the previous experiment, except for the number of clients which we vary from 1 to 100 for A-GORAM and from 1 to 10000 for S-GORAM, and the number of cores which are limited to 4. In the experiments where the number of cores is not explicitly varied, we use the maximum number of cores available.

Discussion. The results of the experiments are reported in Figure 9–14 and Table III. As shown in Figure 9a, varying the block size has a linear effect in the construction without batched shuffle proofs. As expected, the batched shuffle proofs improve the computation time significantly (Figure 9b). The new scheme even seems to be independent of the block size, at least for block sizes less than 64KB. This effect is caused by the parallelization. Still, the homomorphic multiplication of the public-key ciphertexts before the batched shuffle proof computation depends on the block size (line 8.2). Figure 9c and Figure 9d show the results for A-GORAM and S-GORAM. Since the computation time is in practice almost independent of the block size, we can choose larger block sizes in the case of databases with large files, thereby allowing the client to read (resp. write) a file in one shot, as opposed to running multiple read (resp. write) operations. We identify a minimum computation time for 128KB as this is the optimal trade-off between the index map size and the path size. The server computation

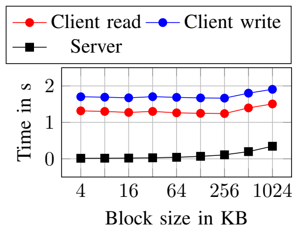
[4]Groth-Sahai proofs are generally not zero-knowledge. However, in our case the witnesses fulfill a special equation for which they are zero-knowledge.



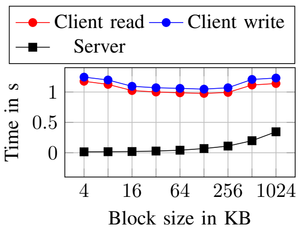
(a) GORAM.



(b) GORAM with batched shuffle proofs and k=3.

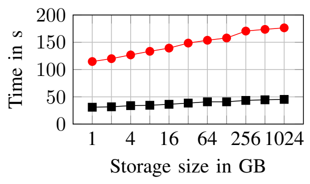


(c) A-GORAM.

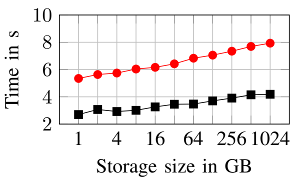


(d) S-GORAM.

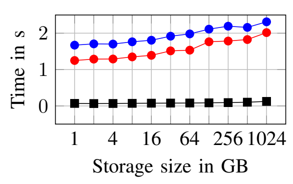
Figure 9. The average execution time for the read and write protocol on client and server for varying B where $BN = 1$GB and $G = 4$.



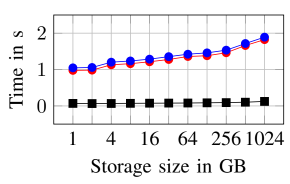
(a) GORAM with $B = 4$KB.



(b) GORAM with batched shuffle proofs, $B = 4$KB, and $k = 4$.

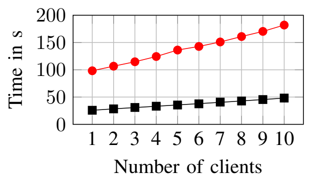


(c) A-GORAM with $B = 128$KB.

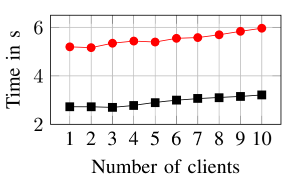


(d) S-GORAM with $B = 128$KB.

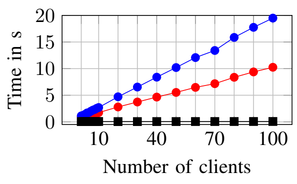
Figure 10. The average execution time for the read and write protocol on client and server for varying BN where $G = 4$.



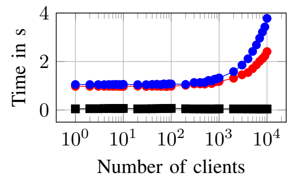
(a) GORAM with $B = 4$KB.



(b) GORAM with batched shuffle proofs, $B = 4$KB, and $k = 4$.



(c) A-GORAM with $B = 128$KB.



(d) S-GORAM with $B = 128$KB.

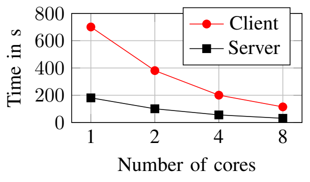
Figure 11. The average execution time for the read and write protocol on client and server for varying G where $BN = 1$GB.

time is low and varies between 15ms and 345ms, while client operations take less than 2 seconds for A-GORAM and less than 1.3 seconds for S-GORAM. As we obtained the best results for 4KB in the experiments for GORAM and 128KB for the others, we use these block sizes in the sequel.

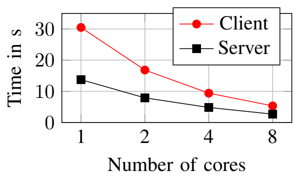
The results obtained by varying the storage size (Figure 10) and the number of clients (Figure 11) prove what the computational complexity suggests. Nevertheless, it is interesting to see the tremendous improvement in computation time between GORAM with and without batched shuffle proofs. The results obtained by varying the iteration time of the batched shuffle proof protocol are depicted in Figure 13 and we verify the expected linear dependency. Smaller values of k are more efficient but higher values give a better soundness probability. If we compare A-GORAM and S-GORAM in Figure 11c and Figure 11d we can see that S-GORAM scales well to a large amount of users as opposed to A-GORAM. The good scaling behavior is due to the used broadcast encryption scheme: it only computes a constant number of pairings independent of the number of users for decryption while the opposite holds for predicate encryption. Nevertheless, we identify a linear growth in the times for S-GORAM, which arises from the linear number of exponentiations that are computed. For instance, in order to write 128KB in a 1GB storage that is used by 100 users, A-GORAM needs about 20 seconds while S-GORAM only needs about 1 second. Even when increasing the number of users to 10000, S-GORAM requires only about 4 seconds, a time that A-GORAM needs for slightly more than 10 users.

Figure 12 shows the results obtained by varying the number of cores. In GORAM most of the computation, especially the zero-knowledge proof computation, can be easily parallelized. We observe this fact in both results (Figure 12a and Figure 12b). In the efficient construction we can parallelize the top-level encryption and decryption, the verification of the entries, and the predicate ciphertext decryption. Also in this case parallelization significantly improves the performance (Figure 12c and Figure 12d). Notice that we run the experiments in this case for 20 clients, as opposed to 4 as done for the other constructions, because the predicate ciphertext decryption takes the majority of the computation time and, hence, longer ciphertexts take longer to decrypt and the parallelization effect can be better visualized.

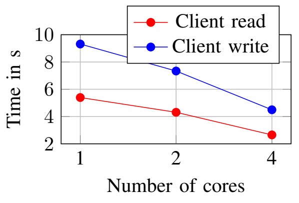
Finally, Table III compares S-GORAM with the underlying Path-ORAM protocol. Naturally, since Path-ORAM only uses symmetric encryption, no broadcast encryption, and no verification with chameleon signatures, the computation time



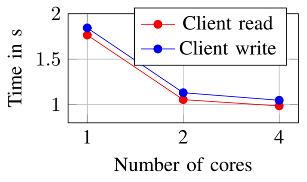
(a) GORAM with $B = 4$KB, and $G = 4$.



(b) GORAM with batched shuffle proofs, $B = 4$KB, $G = 4$, and $k = 4$.



(c) A-GORAM with $BN = 1$GB, $B = 128$KB, and $G = 20$.



(d) S-GORAM with $BN = 1$GB, $B = 128$KB, and $G = 20$.

Figure 12. The average execution time for the read and write protocol on client and server for a varying number of cores where $BN = 1$GB.

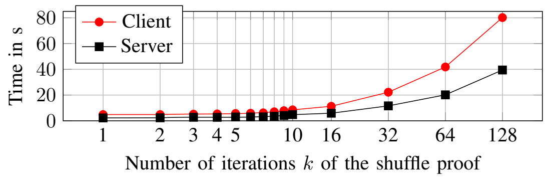
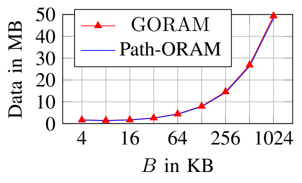


Figure 13. The average execution time for the read and write protocol on client and server for GORAM with batched shuffle proofs and varying k where $BN = 1$GB, $B = 8$KB, and $G = 4$.



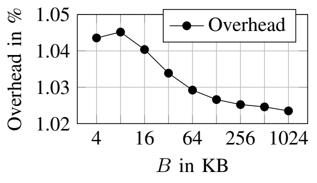


Figure 14. The up-/download amount of data compared between Path-ORAM [4] and S-GORAM for varying B while $BN = 1$GB and $G = 4$.

is much lower. However, the bottleneck of both constructions is actually the amount of data that has to be downloaded and uploaded by the client (Figure 14). The time required to upload and download data may take much more time than the computation time, given today's bandwidths. Here the overhead is only between 1.02% and 1.05%. For instance, assuming a mobile client using LTE (100Mbit/s downlink and 50Mbit/s uplink in peak) transferring 2 and 50 MB takes 480ms and 12s, respectively. Under these assumptions, considering a block size of 1MB, we get a combined computation and communication overhead of 8% for write and 7% for read, which we consider a relatively low price to pay to get a wide range of security and privacy properties in a multi-client setting.

IX. Case Study: Personal Health Records

We briefly discuss a potential application of GORAM, namely, a privacy-preserving personal health record (PHR) management system. As the patient should have the control of her own record, the patient is the data owner. The server is some cloud storage provider, which may be chosen by the patient or directly by the state for all citizens (e.g., ELGA in Austria). The healthcare personal (doctors, nurses, pharmacies, and so on) constitutes the clients.

We discuss now possible real-world attacks on PHRs and how the usage of GORAM prevents them. One typical threat is the cloud provider trying to learn customer information (e.g., to sell it or to use it for targeted advertising). For instance, as previously discussed, monitoring the accesses to DNA sequences would allow the service provider to learn the patient's disease: these kinds of attacks are not possible because of obliviousness and data secrecy. Another possible attack could be a pharmacy that tries to increase its profit by changing a prescription for a cheap medicine into one that prescribes an expensive medicine. However, in GORAM pharmacies would not have write access to prescriptions, and hence, these cannot be changed or, in A-GORAM, the misbehaving pharmacy can be blamed by the data owner. A common procedure in order to sign a contract with a health insurance is the health check. The patient might want to hide health information from the insurance in order to get a lower fee. To this end, the patient could simply try to drop this information. Dropping of entries in the database is, however, either prevented by making such documents permanent or, in A-GORAM, by letting the insurance, who sees that some documents are missing, blame the patient. Using the backup strategy, the missing documents can be restored.

Finally, we think that GORAM with batched shuffle proofs (even more so A-GORAM and S-GORAM) is a practical solution for the management of today's PHRs, since they are of rather small size. For instance, the data today stored in e-health cards is at most 128KB. The current trend is to store the remaining medical information (e.g., DNA information) on an external server, which can be accessed by using the card. This is exactly our setting, except that we allow for accessing PHRs even without the card, which is crucial in emergency situations. DNA information takes approximately 125MB[5] [37] and all our constructions offer an adequate performance for databases of a few gigabytes, with A-GORAM and S-GORAM performing better for the retrieval of large amounts of data, thanks to the possibility of using larger block sizes.

Scheme	Client Read	Client Write	Server
S-GORAM	0.981s	1.075s	0.068s
Path-ORAM	0.042s	0.042s	0.002s

Table III: Comparison of the computation times between Path-ORAM [4] (single-client!) and S-GORAM on 1GB storage size, 128KB block size and 100 clients.

[5]The actual DNA sequence takes about 200GB but one usually shares only the mutations, i.e., the differences of the considered genome to the average human genome. These mutations are only 0.1% of the overall sequence.

X. Related Work

Privacy-preserving outsourced storage. Oblivious RAM (ORAM) [3] is a technique originally devised to protect the access pattern of software on the local memory and thus to prevent the reverse engineering of that software. The observation is that encryption by itself prevents an attacker from learning the content of any memory cell but monitoring how memory is accessed and modified may still leak a great amount of sensitive information. Recent advances in ORAM show that it is efficient enough to hide the data and the user's access pattern in storage outsourcing services [2], [12], [38]–[45].

While a few ORAM constructions guarantee the integrity of user data [46], [47], none of them is suitable to share data with potentially distrustful clients. Goodrich *et al.* [48] studied the problem of multi-client ORAM, but their attacker model does not include malicious, and potentially colluding, clients. Furthermore, their construction does not provide fine-grained access control mechanisms, i.e., either all members of a group have access to a certain data, or none has. Finally, this scheme does not allow the clients to verify the data integrity.

The fundamental problem in existing ORAM constructions is that all clients must have access to the ORAM key, which allows them to read and potentially disrupt the entire database.

A few recent works have started to tackle this problem. Franz *et al.* have introduced the concept of delegated ORAM [49]. The idea is to encrypt and sign each entry with a unique set of keys, initially only known to the data owner: giving a client the decryption key (resp. the decryption and signing keys) suffices to grant read (resp. write) access to that entry. This solution, however, has two drawbacks that undermine its deployment in practice. First, the data owner has to periodically visit the server for checking the validity of the signatures accompanying the data to be inserted in the database (thus tracking the individual client accesses) and reshuffling the ORAM according to the access history in order to enable further unlinkable ORAM accesses. Furthermore, revoking access for a single client requires the data owner to change (and distribute) the capabilities of all other users that have access to that file.

Huang and Goldberg have recently presented a protocol for outsourced private information retrieval [50], which is obtained by layering a private information retrieval (PIR) scheme on top of an ORAM data layout. This solution is efficient and conceals client accesses from the data owner, but it does not give clients the possibility to update data. Moreover, it assumes ℓ non-colluding servers, which is due to the usage of information theoretic multi-server PIR.

De Capitani di Vimercati *et al.* [51] proposed a storage service that uses selective encryption as a means for providing fine-grained access control. The focus of their work is to study how indexing data in the storage can leak information to clients that are not allowed to access these data, although they are allowed to know the indices. The authors do, however, neither consider verifiability nor obliviousness, which distinguishes their storage service from ours.

Finally, there have been a number of works leveraging trusted hardware to realize ORAM schemes [52], [53] including some in the multi-client setting [54], [55]. We, however, intentionally tried to strive for a solution without trusted hardware, only making use of cryptographic primitives.

Verifiable outsourced storage. Verifying the integrity of data outsourced to an untrusted server is a research problem that has recently received increasing attention in the literature. Schröder and Schröder introduced the concept of verifiable data streaming (VDS) and an efficient cryptographic realization thereof [56], [57]. In a verifiable data streaming protocol, a computationally limited client streams a long string to the server, who stores the string in its database in a publicly verifiable manner. The client has also the ability to retrieve and update any element in the database. Papamathou *et al.* [58] proposed a technique, called streaming authenticated data structures, that allows the client to delegate certain computations over streamed data to an untrusted server and to verify their correctness. Other related approaches are proofs-of-retrievability [59]–[62], which allow the server to prove to the client that it is actually storing all of the client's data, verifiable databases [63], which differ from the previous ones in that the size of the database is fixed during the setup phase, and dynamic provable data possession [64]. All the above do not consider the privacy of outsourced data. While some of the latest work has focused on guaranteeing the confidentiality of the data [65], to the best of our knowledge no existing paper in this line of research takes into account obliviousness.

Personal Health Records. Security and privacy concerns seem to be one of the major obstacles towards the adoption of cloud-based PHRs [66], [67], [68]. Different cloud architectures have been proposed [69], as well as database constructions [70], [71], in order to overcome such concerns. However, none of these works takes into account the threat of a curious storage provider and, in particular, none of them enforces the obliviousness of data accesses.

XI. Conclusion and Future Work

This paper introduces the concept of Group ORAM, which captures an unprecedented range of security and privacy properties in the cloud storage setting. The fundamental idea underlying our instantiation is to extend a state-of-the-art ORAM scheme [4] with access control mechanisms and integrity proofs while preserving obliviousness. To tackle the challenge of devising an efficient and scalable construction, we devised a novel zero-knowledge proof technique for shuffle correctness as well as a new accountability technique based on chameleon signatures, both of which are generically applicable and thus of independent interest. We showed how GORAM is an ideal solution for personal record management systems.

As a future work, we intend to relax the assumptions on the server behavior, under which some of the security and privacy properties are proven, developing suitable cryptographic techniques. A further research goal is the design of cryptographic solutions allowing clients to learn only limited information (e.g., statistics) about the dataset.

Acknowledgments

This work was supported by the German research foundation (DFG) through the Emmy Noether program, by the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy and Accountability (CISPA), and by an Intel Early Career Faculty Honor Program Award. Finally, we thank the reviewers for their helpful comments.

References

[1] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation," in *NDSS'12*. Internet Society, 2012.

[2] B. Pinkas and T. Reinman, "Oblivious RAM Revisited," in *CRYPTO'10*, ser. LNCS. Springer, 2010, pp. 502–519.

[3] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.

[4] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," in *CCS'13*. ACM, 2013.

[5] J. Katz, A. Sahai, and B. Waters, "Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products," in *EUROCRYPT'08*. Springer, 2008, pp. 146–162.

[6] J. Groth and A. Sahai, "Efficient Noninteractive Proof Systems for Bilinear Groups," *SIAM J. Comp.*, vol. 41, no. 5, pp. 1193–1232, 2012.

[7] S. Bayer and J. Groth, "Efficient Zero-Knowledge Argument for Correctness of a Shuffle," in *EUROCRYPT'12*, ser. LNCS. Springer, 2012, pp. 263–280.

[8] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Privacy and Access Control for Outsourced Personal Records," Cryptology ePrint Archive, Report 2015/224, 2015, http://eprint.iacr.org/.

[9] I. E. Akkus, R. Chen, M. Hardt, P. Francis, and J. Gehrke, "Non-tracking Web Analytics," in *CCS'12*. ACM, 2012, pp. 687–698.

[10] R. Chen, I. E. Akkus, and P. Francis, "SplitX: High-Performance Private Analytics," in *SIGCOMM'13*. ACM, 2013, pp. 315–326.

[11] S. Goldwasser and S. Micali, "Probabilistic Encryption & How To Play Mental Poker Keeping Secret All Partial Information," in *STOC'82*. ACM, 1982, pp. 365–377.

[12] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM With $O((\log n)^3)$ Worst-Case Cost," in *ASIACRYPT'11*, ser. LNCS. Springer, 2011, pp. 197–214.

[13] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," in *USENIX'04*. USENIX Association, 2004, pp. 303–320.

[14] M. Maffei, K. Pecina, and M. Reinert, "Security and Privacy by Declarative Design," in *CSF'13*. IEEE Press, 2013, pp. 81–96.

[15] M. Backes, S. Lorenz, M. Maffei, and K. Pecina, "Anonymous Webs of Trust," in *PETS'10*, ser. LNCS. Springer, 2010, pp. 130–148.

[16] M. Backes, M. Maffei, and K. Pecina, "Automated Synthesis of Privacy-Preserving Distributed Applications," in *NDSS'12*. Internet Society, 2012.

[17] F. Baldimtsi and A. Lysyanskaya, "Anonymous Credentials Light," in *CCS'13*. ACM, 2013, pp. 1087–1098.

[18] D. L. Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Comm. ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[19] M. Jakobsson and A. Juels, "Millimix: Mixing in Small Batches," DIMACS, Tech. Rep. 99-33, 1999.

[20] M. Jakobsson, A. Juels, and R. L. Rivest, "Making Mix Nets Robust for Electronic Voting by Randomized Partial Checking," in *USENIX'02*. USENIX Association, 2002, pp. 339–353.

[21] A. Fiat and A. Shamir, "How to Prove Yourself: Practical Solutions to Identification and Signature Problems," in *CRYPTO'86*. Springer, 1987, pp. 186–194.

[22] G. Ateniese and B. de Medeiros, "On the Key Exposure Problem in Chameleon Hashes," in *SCN'04*, ser. LNCS. Springer, 2004, pp. 165–179.

[23] J. Camenisch, M. Kohlweiss, and C. Soriente, "An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials," in *PKC'09*, ser. LNCS. Springer, 2009, pp. 481–500.

[24] C. Gentry and B. Waters, "Adaptive Security in Broadcast Encryption Systems (with Short Ciphertexts)," in *EUROCRYPT'09*, ser. LNCS. Springer, 2009, pp. 171–188.

[25] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance," in *PODC'87*. ACM, 1987, pp. 1–12.

[26] R. Küsters, T. Truderung, and A. Vogt, "Accountability: Definition and Relationship to Verifiability," in *CCS'10*. ACM, 2010, pp. 526–535.

[27] J. Daemen and V. Rijmen, *The Design of Rijndael, AES - The Advanced Encryption Standard*. Springer, 2002.

[28] T. El Gamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," in *CRYPTO'84*, ser. LNCS. Springer, 1985, pp. 10–18.

[29] D. M. Freeman, "Converting Pairing-Based Cryptosystems from Composite-Order Groups to Prime-Order Groups," in *EUROCRYPT'10*, ser. LNCS. Springer, 2010, pp. 44–61.

[30] C. P. Schnorr, "Efficient Identification and Signatures for Smart Cards," in *CRYPTO'89*, ser. LNCS. Springer, 1989, pp. 239–252.

[31] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Comm. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[32] J. Backes, S. Lorenz, and K. Pecina, "Zero-knowledge Library," online at github.com/peloba/zk-library.

[33] A. D. Caro, "jPBC - Java Library for Pairing Based Cryptography," online at http://gas.dia.unisa.it/projects/jpbc/.

[34] B. Lynn, "PBC - C Library for Pairing Based Cryptography," online at http://crypto.stanford.edu/pbc/.

[35] A. Miyaji, M. Nakabayashi, and S. Takano, "Characterization of Elliptic Curve Traces under FR-Reduction," in *ICISC'00*, ser. LNCS, vol. 2015. Springer, 2001, pp. 90–108.

[36] BlueKrypt, "Cryptograhpic Key Length Recommendation," online at www.keylength.com.

[37] R. J. Robinson, "How big is the human genome?" Online at https://medium.com/precision-medicine/how-big-is-the-human-genome-e90caa3409b0.

[38] B. Carbunar and R. Sion, "Regulatory Compliant Oblivious RAM," in *ACNS'10*, ser. LNCS. Springer, 2010, pp. 456–474.

[39] M. Ajtai, "Oblivious RAMs Without Cryptographic Assumptions," in *STOC'10*. ACM, 2010, pp. 181–190.

[40] M. T. Goodrich and M. Mitzenmacher, "Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation," in *ICALP'11*, ser. LNCS. Springer, 2011, pp. 576–587.

[41] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly Secure Oblivious RAM Without Random Oracles," in *TCC'11*, ser. LNCS. Springer, 2011, pp. 144–163.

[42] E. Stefanov and E. Shi, "Multi-Cloud Oblivious Storage," in *CCS'13*. ACM, 2013, pp. 247–258.

[43] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "PHANTOM: Practical Oblivious Computation in a Secure Processor," in *CCS'13*. ACM, 2013, pp. 311–324.

[44] E. Stefanov and E. Shi, "ObliviStore: High Performance Oblivious Cloud Storage," in *S&P'13*. IEEE Press, 2013, pp. 253–267.

[45] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam, "Verifiable Oblivious Storage," in *PKC'14*, ser. LNCS. Springer, 2014, pp. 131–148.

[46] P. Williams, R. Sion, and B. Carbunar, "Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage," in *CCS'08*. ACM, 2008, pp. 139–148.

[47] E. Stefanov, E. Shi, and D. Song, "Towards Practical Oblivious RAM," in *NDSS'12*. Internet Society, 2012.

[48] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation," in *SODA'12*. SIAM, 2012, pp. 157–167.

[49] M. Franz, C. Carbunar, R. Sion, S. Katzenbeisser, M. Sotakova, P. Williams, and A. Peter, "Oblivious Outsourced Storage with Delegation," in *FC'11*. Springer, 2011, pp. 127–140.

[50] Y. Huang and I. Goldberg, "Outsourced Private Information Retrieval with Pricing and Access Control," in *WPES'13*. ACM, 2013.

[51] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Private Data Indexes for Selective Access to Outsourced Data," in *WPES'11*. ACM, 2011, pp. 69–80.

[52] M. Backes, A. Kate, M. Maffei, and K. Pecina, "ObliviAd: Provably Secure and Practical Online Behavioral Advertising," in *S&P'12*. IEEE Press, 2012, pp. 257–271.

[53] A. Kate, M. Maffei, P. Moreno-Sanchez, and K. Pecina, "Privacy Preserving Payments in Credit Networks," in *NDSS'15*. Internet Society, 2015.

[54] A. Iliev and S. W. Smith, "Protecting Client Privacy with Trusted Computing at the Server," *IEEE Security and Privacy*, vol. 3, no. 2, pp. 20–28, Mar. 2005.

[55] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman, "Shroud: Ensuring Private Access to Large-scale Data in the Data Center," in *FAST'13*. USENIX Association, 2013, pp. 199–214.

[56] D. Schröder and H. Schröder, "Verifiable Data Streaming," in *CCS'12*. ACM, 2012, pp. 953–964.

[57] D. Schröder and M. Simkin, "VeriStream - A Framework for Verifiable Data Streaming," in *FC'15*. Springer, 2015.

[58] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi, "Streaming Authenticated Data Structures ," in *EUROCRYPT'13*, 2013.

[59] H. Shacham and B. Waters, "Compact Proofs of Retrievability," in *ASIACRYPT'08*, ser. LNCS. Springer, 2008, pp. 90–107.

[60] D. L. G. Filho and P. S. L. M. Barreto, "Demonstrating Data Possession and Uncheatable Data Transfer," Cryptology ePrint Archive, Report 2006/150, 2006, http://eprint.iacr.org/.

[61] T. Schwarz and E. L. Miller, "Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage," 2006.

[62] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels, "Iris: A Scalable Cloud File System with Efficient Integrity Checks," Cryptology ePrint Archive, Report 2011/585, 2011, http://eprint.iacr.org/.

[63] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable Delegation of Computation Over Large Datasets," in *CRYPTO'11*, ser. LNCS. Springer, 2011, pp. 111–131.

[64] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic Provable Data Possession," in *CCS'09*. ACM, 2009, pp. 213–222.

[65] M. van Dijk, A. Juels, A. Oprea, R. L. Rivest, E. Stefanov, and N. Triandopoulos, "Hourglass Schemes: How to Prove That Cloud Files Are Encrypted," in *CCS'12*. ACM, 2012, pp. 265–280.

[66] I. Carrión Señor, L. J. Fernández-Alemán, and A. Toval, "Are Personal Health Records Safe? A Review of Free Web-Accessible Personal Health Record Privacy Policies," *J. Med. Int. Res.*, vol. 14, no. 4, 2012.

[67] D. Daglish and N. Archer, "Electronic Personal Health Record Systems: A Brief Review of Privacy, Security, and Architectural Issues," *World Congress on Privacy, Security, Trust and the Management of e-Business*, pp. 110–120, 2009.

[68] K. T. Win, W. Susilo, and Y. Mu, "Personal Health Record Systems and Their Security Protection," *J. Med. Sys.*, vol. 30, no. 4, pp. 309–315, 2006.

[69] H. Löhr, A.-R. Sadeghi, and M. Winandy, "Securing the e-Health Cloud," in *IHI'10*. ACM, 2010, pp. 220–229.

[70] M. Li, S. Yu, K. Ren, and W. Lou, "Securing Personal Health Records in Cloud Computing: Patient-Centric and Fine-Grained Data Access Control in Multi-owner Settings," in *SECURECOMM'10*, 2010.

[71] P. Korde, V. Panwar, and S. Kalse, "Securing Personal Health Records in Cloud using Attribute Based Encryption," *Int. J. Eng. Adv. Tech.*, 2013.

APPENDIX
FORMAL DEFINITIONS

A. Secrecy

Definition 2 (Secrecy): A Group ORAM GORAM = (gen, addCl, addE, chMode, read, write) *preserves secrecy*, if for every PPT adversary $\mathcal{A}$ the following probability is negligible in the security parameter λ:

$$\left|\Pr[\mathsf{ExpSec}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, 1) = 1] - \Pr[\mathsf{ExpSec}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, 0) = 1]\right|$$

where $\mathsf{ExpSec}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, b)$ is the following game:

Setup: The challenger runs $(cap_{\mathcal{O}}, \mathsf{DB}) \leftarrow \mathsf{gen}(1^\lambda)$, sets $\mathbf{AC} := []$, and runs a black-box simulation of $\mathcal{A}$ to which it hands over DB.

Queries: The challenger provides $\mathcal{A}$ with interactive interfaces addCl, addE, chMode, read, write, and corCl that $\mathcal{A}$ may query adaptively and in any order. Each round $\mathcal{A}$ can query exactly one interface. These interfaces are described below:

(1) On input $\mathsf{addCl}(\mathbf{a})$ by $\mathcal{A}$, the challenger executes $\mathsf{addCl}(cap_{\mathcal{O}}, \mathbf{a})$ locally and stores the capability cap_i returned by the algorithm.

(2) On input $\mathsf{addE}(\mathbf{a}, d)$ by $\mathcal{A}$, the challenger executes $\langle \mathcal{C}_{\mathsf{addE}}(cap_{\mathcal{O}}, \mathbf{a}, d), \mathcal{S}_{\mathsf{addE}}(\mathsf{DB})\rangle$ in interaction with $\mathcal{A}$, where the former plays the role of the client while the latter plays the role of the server.

(3) On input $\mathsf{chMode}(\mathbf{a}, j)$ by $\mathcal{A}$, the challenger executes $\langle \mathcal{C}_{\mathsf{chMode}}(cap_{\mathcal{O}}, \mathbf{a}, j), \mathcal{S}_{\mathsf{chMode}}(\mathsf{DB})\rangle$ in interaction with $\mathcal{A}$.

(4) On input $\mathsf{corCl}(i)$ by $\mathcal{A}$, the challenger hands over the capability cap_i related to the i-th client in the access control matrix $\mathbf{AC}$.

(5) On input $\mathsf{read}(i, j)$ by $\mathcal{A}$, the challenger executes $\langle \mathcal{C}_{\mathsf{read}}(cap_i, j), \mathcal{S}_{\mathsf{read}}(\mathsf{DB})\rangle$ in interaction with $\mathcal{A}$.

(6) On input $\mathsf{write}(i, j, d)$ by $\mathcal{A}$, the challenger executes $\langle \mathcal{C}_{\mathsf{write}}(cap_i, j, d), \mathcal{S}_{\mathsf{write}}(\mathsf{DB})\rangle$ in interaction with $\mathcal{A}$.

Challenge: Finally, $\mathcal{A}$ outputs $(j, (d_0, d_1))$, where j is an index denoting the database entry on which $\mathcal{A}$ wants to be challenged and (d_0, d_1) is a pair of entries such that $|d_0| = |d_1|$. The challenger accepts the request only if $\mathbf{AC}(i, j) = \bot$, for every i corrupted by $\mathcal{A}$ in the query phase. Afterwards it invokes $\langle \mathcal{C}_{\mathsf{write}}(cap_{\mathcal{O}}, j, d_b), \mathcal{S}_{\mathsf{write}}(\mathsf{DB})\rangle$ in interaction with $\mathcal{A}$.

Output: In the output phase $\mathcal{A}$ still has access to the interfaces except for addCl on input $\mathbf{a}$ such that $\mathbf{a}(j) \neq \bot$; corCl on input i such that $\mathbf{AC}(i, j) \neq \bot$; and chMode on input $\mathbf{a}, i$ with $\mathbf{a}(i) \neq \bot$ for some previously corrupted client i. Eventually, $\mathcal{A}$ stops, outputting a bit b'. The challenger outputs 1 if and only if $b = b'$.

B. Integrity

Definition 3 (Integrity): A Group ORAM GORAM = (gen, addCl, addE, chMode, read, write) *preserves integrity*, if for every PPT adversary $\mathcal{A}$ the following probability is negligible in the security parameter:

$$\Pr[\mathsf{ExpInt}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda) = 1]$$

where $\mathsf{ExpInt}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda)$ is the following game:

Setup: The challenger runs $(cap_{\mathcal{O}}, \mathsf{DB}) \leftarrow \mathsf{gen}(1^\lambda)$, sets $\mathbf{AC} := []$, and runs a black-box simulation of $\mathcal{A}$. Furthermore, the challenger initializes a second database $\mathsf{DB}' := []$ which is managed locally.

Queries: The challenger provides $\mathcal{A}$ with the same interfaces as in Definition 2, which $\mathcal{A}$ may query adaptively and in any order. Since DB is maintained on the challenger's side, the queries to addE, chMode, read and write are locally executed by the challenger. Furthermore, the challenger updates DB' locally for all affecting interface calls.

Challenge: Finally, the adversary outputs an index j^* which he wants to be challenged on. If there exists a capability cap_i provided to $\mathcal{A}$ with $\mathbf{AC}(i, j^*) = rw$, the challenger aborts. Otherwise it runs $d^* \leftarrow \langle \mathcal{C}_{\mathsf{read}}(cap_{\mathcal{O}}, j^*), \mathcal{S}_{\mathsf{read}}(\mathsf{DB}) \rangle$ locally.

Output: It outputs 1 if and only if $d^* \neq \mathsf{DB}'(j^*)$.

C. Tamper Resistance

Definition 4 (Tamper resistance): A Group ORAM $\mathsf{GORAM} = (\mathsf{gen}, \mathsf{addCl}, \mathsf{addE}, \mathsf{chMode}, \mathsf{read}, \mathsf{write})$ is *tamper-resistant*, if for every PPT adversary $\mathcal{A}$ the following probability is negligible in the security parameter:

$$\Pr[\mathsf{ExpTamRes}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda) = 1]$$

where $\mathsf{ExpTamRes}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda)$ is the following game:

Setup: The challenger runs the Setup phase as in Definition 2. Furthermore, it forwards DB to $\mathcal{A}$ and initializes a second database DB' which is managed locally.

Queries: The challenger provides $\mathcal{A}$ with the same interfaces as in Definition 2, which $\mathcal{A}$ may query adaptively and in any order. Furthermore, the challenger updates DB' locally for all affecting interface calls.

Challenge: Finally, the adversary outputs an index j^* which he wants to be challenged on. If there exists a capability cap_i that has ever been provided to $\mathcal{A}$ such that $\mathbf{AC}(i, j^*) = rw$, then the challenger aborts. The challenger runs $d^* \leftarrow \langle \mathcal{C}_{\mathsf{read}}(cap_{\mathcal{O}}, j^*), \mathcal{S}_{\mathsf{read}}(\mathsf{DB}) \rangle$ in interaction with $\mathcal{A}$.

Output: It outputs 1 if and only if $d^* \neq \mathsf{DB}'(j^*)$.

D. Obliviousness

Definition 5 (Obliviousness): A Group ORAM $\mathsf{GORAM} = (\mathsf{gen}, \mathsf{addCl}, \mathsf{addE}, \mathsf{chMode}, \mathsf{read}, \mathsf{write})$ is *oblivious*, if for every PPT adversary $\mathcal{A}$ the following probability is negligible in the security parameter:

$$\Big| \Pr[\mathsf{ExpObv}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, 1) = 1] - \Pr[\mathsf{ExpObv}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, 0) = 1] \Big|$$

where $\mathsf{ExpObv}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, b)$ is the following game:

Setup: The challenger runs $(cap_{\mathcal{O}}, \mathsf{DB}) \leftarrow \mathsf{gen}(1^\lambda)$ as in Definition 2 and it forwards DB to $\mathcal{A}$.

Queries: The challenger provides $\mathcal{A}$ with the same interfaces as in Definition 2 except corCl, which $\mathcal{A}$ may query adaptively and in any order. Furthermore, $\mathcal{A}$ is provided with the following additional interface:

(1) On input $\mathsf{query}(\{(i_0, j_0), (i_0, j_0, d_0)\}, \{(i_1, j_1), (i_1, j_1, d_1)\})$ by $\mathcal{A}$, the challenger checks whether $j_0 \leq |\mathsf{DB}|$, $j_1 \leq |\mathsf{DB}|$, and i_0, i_1 are valid clients. Furthermore, it checks that the operations requested by $\mathcal{A}$ are allowed by $\mathbf{AC}$. If not it aborts. Otherwise it executes $\langle \mathcal{C}_{\mathsf{read}}(cap_{i_b}, j_b), \mathcal{S}_{\mathsf{read}}(\mathsf{DB}) \rangle$ or $\langle \mathcal{C}_{\mathsf{write}}(cap_{i_b}, j_b, d_b), \mathcal{S}_{\mathsf{write}}(\mathsf{DB}) \rangle$ depending on the input, in interaction with $\mathcal{A}$. Here the challenger plays the role of the client and $\mathcal{A}$ plays the role of the server.

Output: Finally, $\mathcal{A}$ outputs a bit b'. The challenger outputs 1 if and only if $b = b'$.

E. Anonymity

Definition 6 (Anonymity): A Group ORAM $\mathsf{GORAM} = (\mathsf{gen}, \mathsf{addCl}, \mathsf{addE}, \mathsf{chMode}, \mathsf{read}, \mathsf{write})$ is *anonymity-preserving*, if for every PPT adversary $\mathcal{A}$ the following probability is negligible in the security parameter:

$$\Big| \Pr[\mathsf{ExpAnon}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, 1) = 1] - \Pr[\mathsf{ExpAnon}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, 0) = 1] \Big|$$

where $\mathsf{ExpAnon}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda, b)$ is the following game:

Setup: The challenger runs $(cap_{\mathcal{O}}, \mathsf{DB}) \leftarrow \mathsf{gen}(1^\lambda)$ and it forwards $cap_{\mathcal{O}}$ and DB to $\mathcal{A}$.

Queries: The challenger provides $\mathcal{A}$ with read and a write interactive interfaces that $\mathcal{A}$ may query adaptively and in any order. Each round $\mathcal{A}$ can query exactly one interface. The interfaces are described below:

(3) On input $\mathsf{read}(cap_i, j)$ by $\mathcal{A}$, the challenger executes $\langle \mathcal{C}_{\mathsf{read}}(cap_i, j), \mathcal{S}_{\mathsf{read}}(\mathsf{DB}) \rangle$ in interaction with $\mathcal{A}$, where the former plays the role of the server and the latter plays the role of the client.

(4) On input $\mathsf{write}(cap_i, j, d)$ by $\mathcal{A}$, the challenger executes $\langle \mathcal{C}_{\mathsf{write}}(cap_i, j, d), \mathcal{S}_{\mathsf{write}}(\mathsf{DB}) \rangle$ in interaction with $\mathcal{A}$, where the former plays the role of the server and the latter plays the role of the client.

Challenge: $\mathcal{A}$ outputs $((cap_{i_0}, cap_{i_1}), \{j, (j, d)\})$, where (cap_{i_0}, cap_{i_1}) is a pair of capabilities, j is an index denoting the database entry on which $\mathcal{A}$ wishes to be challenged, and d is some data. The challenger checks whether $\mathbf{AC}(i_0, j) = \mathbf{AC}(i_1, j)$: if not, then it aborts, otherwise it executes $\langle \mathcal{C}_{\mathsf{read}}(cap_{i_b}, j), \mathcal{S}_{\mathsf{read}}(\mathsf{DB}) \rangle$ or $\langle \mathcal{C}_{\mathsf{write}}(cap_{i_b}, j, d), \mathcal{S}_{\mathsf{write}}(\mathsf{DB}) \rangle$ in interaction with $\mathcal{A}$.

Output: Finally, $\mathcal{A}$ outputs a bit b'. The challenger outputs 1 if and only if $b = b'$.

F. Accountability

Definition 7 (Accountability): A Group ORAM $\mathsf{GORAM} = (\mathsf{gen}, \mathsf{addCl}, \mathsf{addE}, \mathsf{chMode}, \mathsf{read}, \mathsf{write}, \mathsf{blame})$ is *accountable*, if for every PPT adversary $\mathcal{A}$ the following probability is negligible in the security parameter:

$$\Pr[\mathsf{ExpAcc}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda) = 1]$$

where $\mathsf{ExpAcc}^{\mathcal{A}}_{\mathsf{GORAM}}(\lambda)$ is the following game:

Setup: The challenger runs the Setup phase as in Definition 3.

Queries: The challenger runs the Query phase as in Definition 3.

Challenge: Finally, the adversary outputs an index j^* which he wants to be challenged on. If there exists a capability cap_i provided to $\mathcal{A}$ such that $\mathbf{AC}(i, j^*) = rw$, then the challenger aborts. The challenger runs $d^* \leftarrow \langle \mathcal{C}_{\mathsf{read}}(cap_{\mathcal{O}}, j^*), \mathcal{S}_{\mathsf{read}}(\mathsf{DB}) \rangle$ and $L \leftarrow \mathsf{blame}(cap_{\mathcal{O}}, \mathsf{Log}, j^*)$ locally.

Output: It outputs 1 if and only if $d^* \neq \mathsf{DB}'(j^*)$ and $\exists\, i \in L$ that has not been queried by $\mathcal{A}$ to the interface $\mathsf{corCl}(\cdot)$ or $L = []$.