Oblivious Outsourced Storage with Delegation

Martin Franz[1], Peter Williams[2], Bogdan Carbunar[3], Stefan Katzenbeisser[1,4], Andreas Peter[4], Radu Sion[2], and Miroslava Sotakova[2]

[1] Center for Advanced Security Research Darmstadt - CASED
[2] Computer Science, Stony Brook University
[3] Applied Research Center, Motorola Labs
[4] Technische Universität Darmstadt

Abstract. In the past few years, outsourcing private data to untrusted servers has become an important challenge. This raises severe questions concerning the security and privacy of the data on the external storage. In this paper we consider a scenario where multiple clients want to share data on a server, while hiding all access patterns. We propose here a first solution to this problem based on Oblivious RAM (ORAM) techniques. Data owners can delegate rights to external new clients enabling them to privately access portions of the outsourced data served by a curious server. Our solution is as efficient as the underlying ORAM constructs and allows for delegated read or write access while ensuring strong guarantees for the privacy of the outsourced data. The server does not learn anything about client access patterns while clients do not learn anything more than what their delegated rights permit.

1 Introduction

As data management is increasingly being outsourced to third party "cloud" providers such as Google, Amazon or Microsoft, enabling secure, distributed access to outsourced data becomes essential. This raises new requirements concerning the privacy of the outsourced data with respect to the external storage, network traffic observers or even collaborators who might have access to parts of the outsourced database. In this scenario, a data owner O outsources his data items to a server S. At a later time, he wishes to delegate read- or write access to individual data items to third party clients $C_1, \ldots, C_n$. Since the data is potentially privacy sensitive, strong confidentiality and privacy guarantees should be in place. Clients should only be able to access those items they are given access to. Moreover, potential adversaries should be unable to derive information from the observed access patterns to the outsourced database. This is necessary, as even the observation that one item is accessed more frequently than others or the fact that one item is accessed by multiple clients, might leak sensitive information about this particular item.

In the special case where the owner is the sole client accessing the data stored on the server, the problem can be solved by applying techniques from Oblivious RAMs [10,21,18]. An ORAM structure preserves not only data confidentiality

G. Danezis (Ed.): FC 2011, LNCS 7035, pp. 127–140, 2012.

but also provides privacy for client data accesses. So far, the problem of hiding access patterns in outsourcing database scenarios containing multiple (distrusted) clients is open. In this paper we show that ORAM techniques can be adapted to this scenario as well. To this end, we introduce a new ORAM feature: delegated access. Data owners can delegate controlled access to their outsourced database to third parties, while preserving full access privacy and data confidentiality. Achieving this turns out to be non-trivial: in addition to preserving the owner's access privacy, we also need to ensure that (i) the server is unable to learn the access patterns of any of the clients, (ii) no client is able to learn or modify any information of items she cannot access and (iii) no client can learn the access patterns of items which she cannot access herself.

1.1 Applications

We now describe several applications that can be built on the constructions we propose in this paper.

Anonymous Banking. The numbered accounts supported by several banks claim to provide user privacy. However, by allowing banks to trace the currency flow and build access pattern statistics, they can be used to learn undesired information, ultimately compromising privacy. The solution proposed in this paper can be used toward preventing such leaks: Account numbers and details are stored as records by the bank and account owners can delegate access rights to other clients as desired. Since the data is accessed obliviously, the bank can learn neither which records are being accessed nor the access rights associated with users.

Oblivious Document Sharing. Document sharing applications such as Google Docs suffer from obvious security and privacy shortcomings. Not only is the central storage able to access the cleartext documents but it can also learn access privileges as well as access patterns and exact contributions from individual users. Our solution is the perfect fit for this problem: Users can store encrypted documents, privately outsource read and write privileges and obliviously and efficiently access desired documents as allowed by their permissions.

Rating Agency Access. Privacy is of paramount importance in financial markets. Public knowledge of investor interest can influence the ratings and prices of company shares in undesired ways. The natural question to ask is, can an investor privately obtain desired information about companies of interest? The solution we provide in this paper answers this question affirmatively. A rating agency maintains an ORAM with records containing ratings and general information for individual companies. Each company owns its own records and can delegate write access to specialized rating assessing companies and, at the same time, an on-demand read-only access to clients that pay to privately access them.

1.2 Contributions

We devise delegated ORAM privacy and security properties, expressing the fact that clients cannot learn any information about items which they are not allowed to access. We provide a first construction of an ORAM with delegation that satisfies this property while preserving the original ORAM privacy properties. The construction relies on a new type of read, write and insert capabilities issued by data owners for items that clients should be able to access. We also show how data owners can efficiently revoke access rights.

2 Building Blocks and Related Work

In this section we describe cryptographic primitives used in our solution, the concept of ORAMs and other related work.

2.1 Cryptographic Primitives

In the solution presented in this paper, we make use of the following primitives:

Symmetric encryption. We will use a symmetric encryption scheme to encrypt items which are stored in the database. In particular, we employ an IND-CPA secure, anonymous, verifiable, symmetric encryption scheme (E, D, K) where E and D are the encryption and decryption algorithms and K is the key generation algorithm which outputs the secret key [14].

Signatures. We also use an existentially unforgeable identity-based signature scheme which consists of four algorithms $(\mathbb{G}, \mathbb{K}, \mathbb{S}, \mathbb{V})$. $\mathbb{G}$ outputs public operating parameters as well as a keypair containing a master public key $\mathcal{M}_P$ and a master secret key $\mathcal{M}_S$; $\mathbb{K}_{\mathcal{M}_S}(id)$ outputs a private signing key s_{id} for an identity id; $\mathbb{S}(s_{id}, message)$ and $\mathbb{V}_{\mathcal{M}_P}(signature, id, message)$ are the signature generation and verification algorithms. A concrete instantiation of an identity-based signature scheme can be found in [17].

2.2 Oblivious RAM

Oblivious RAM [10] provides access pattern privacy to clients (or software processes) accessing a remote database (or RAM). The database is considered to be a set of n encrypted pairs of the form $(id, value)$, denoting an item $value$ stored under a searchable tag id, and supports read and write operations. Client access privacy is obtained by maintaining two invariants: (i) never reveal the id values of interest in a query and (ii) never look twice in the same spot for the same id. Since we base our work on the "square root" solution [10], we briefly recall it here:

The "square root" solution. In addition to the n locations reserved for items of the database, the server maintains $2\sqrt{n}$ additional memory locations. $\sqrt{n}$ of them store dummy items (used to preserve access privacy as discussed below).

The remaining $\sqrt{n}$ locations serve as a "cache" buffer. To hide the virtual access pattern, the client first obliviously shuffles the database items together with the $\sqrt{n}$ dummy items, using a permutation chosen uniformly at random. The suggested way to do that is the following: We assign all $m := n + \sqrt{n}$ items a tag, chosen at random from a space of size m^2/ϵ, yielding a collision probability of ϵ. Then the client sorts the items according to their tags obliviously, using a universal sorting network (such as a Batcher network). Once the database is shuffled, $\sqrt{n}$ database accesses are possible by the client before another reshuffle has to take place. To access an item *id*, the client first reads the *entire* buffer. If *id* is not found there, the client retrieves it from the database by performing a binary search for the element indexed by the random tag which was associated to *id* upon the last reshuffle over all $n + \sqrt{n}$ real- and dummy items stored on the server. Notice that the location at which the item has been found does not need to be kept hidden. This is because from the perspective of the server, any database location can potentially store any item. If, on the other hand, *id* is found in the cache buffer, the client retrieves a previously unread *dummy* item. This is necessary to hide from the server whether the desired item *id* was found in the buffer, and thus hide access patterns and inter-query correlation. Finally, the client places the retrieved and re-encrypted item in the cache buffer. When the buffer becomes full, the client obliviously reorganizes the items in the database together with the ones in the cache buffer (while also generating new dummy items), and the process is ready to repeat.

Clearly, from the server's point of view, the database locations are accessed in a random order and each of them at most once. Per each access, the procedure achieves an (amortized) overhead of $O(\sqrt{n}\,\log^2 n)$. As discussed in [10], the result can be optimized to achieve an $O(\sqrt{n}\,\log n)$ computational overhead.

2.3 Related Work

Private Information Retrieval. Another set of existing mechanisms handle access pattern privacy (but *not data confidentiality*) in the presence of *multiple clients*. Private Information Retrieval (PIR) [5] protocols aim to allow (arbitrary, multiple) clients to retrieve information from public or private databases, without revealing to the database servers which records are retrieved.

In initial results, Chor et al. [5] proved that in an information theoretic setting, any single-server solution requires $\Omega(n)$ bits of communication. PIR schemes with only sub-linear communication overheads, such as [5], require multiple non-communicating servers to hold replicated copies of the data. When the information theoretic guarantee is relaxed single-server solutions with better complexities exist; an excellent survey of PIR can be found online [8,9].

Recently, Sion and Carbunar [19] showed that due to computation costs, use of existing non-trivial single-server PIR protocols on current hardware is still orders of magnitude more time-consuming than trivially transferring the entire database. Their deployment would in fact *increase* overall execution time, as well as the probability of *forward* leakage, when the present trapdoors become eventually vulnerable – e.g., today's queries will be revealed once factoring of

today's values will become possible in the future. Their result goes beyond existing knowledge of mere "impracticality" under unfavorable assumptions. On real hardware, *no* existing non-trivial single server PIR protocol could have possibly had out-performed the trivial client-to-server transfer of records in the past, and is likely not to do so in the future either. This negative result is due to the fact that on any known past general-purpose Von Neumann hardware, it is simply more expensive to PIR-process one bit of information than to transfer it over a network.

Hardware-aided PIR. The recent advent of tamper-resistant, general-purpose trustworthy hardware such as the IBM 4764 Secure Co-Processor [12] has opened the door to efficiently deploying ORAM privacy primitives for PIR purposes (i.e., for arbitrary public or private data, not necessarily originated by the current client) by deploying such hardware as a trusted server-side client proxy.

Trusted hardware devices however are not a panacea. Their practical limitations pose a set of significant challenges in achieving sound regulatory-compliance assurances. Specifically, heat dissipation concerns under tamper-resistant requirements limit the maximum allowable spatial gate-density. As a result, general-purpose secure coprocessors are significantly constrained in both computation ability and memory capacity, being up to one order of magnitude slower than host CPUs.

Asonov was the first to introduce [1] a PIR scheme that uses a secure CPU to provide (an apparent) $O(1)$ online communication cost between the client and server. However, this requires the secure CPU on the server side to scan portions of the database on every request, indicating a computational complexity cost of $O(n)$, where n is the size of the database.

An ORAM-based PIR mechanism is introduced by Iliev and Smith [13], who deploy secure hardware to achieve a cost of $O(\sqrt{n}\log n)$. This is better than the poly-logarithmic complexity granted by ORAM for the small database sizes they consider. This work is notable as one of the first full ORAM-based PIR setups.

An improved ORAM-based PIR mechanism with $O(n/k)$ cost is introduced in [20], where n is the database size and k is the amount of secure storage. The protocol is based on a careful scrambling of a minimal set of server-hosted items. A partial reshuffle costing $O(n)$ is performed every time the secure storage fills up, which occurs once every k queries. While an improvement, this result is not always practical since the total database size n often remains much larger than the secure hardware size k. For $k = \sqrt{n}$, this yields an $O(\sqrt{n})$ complexity (significantly greater than $O(\log\log n \log n)$ for practical values of n).

In [22] Williams et al. introduced a faster ORAM variant which also features correctness guarantees, with computational complexity costs and storage overheads of only $O(\log n \log\log n)$ (amortized per-query), under the assumption of $O(\sqrt{n})$ temporary client storage. In their work, the assumed client storage is used to speed up the reshuffle process by taking advantage of the predictable nature of a merge sort on uniform random data.

Oblivious Transfer with Access Control. Camenisch et al. [4] study the problem of performing k sequential oblivious transfers (OT) between a client and a server storing N values. The work makes the case that previous solutions tolerate selective failures. A selective failure occurs when the server may force the following behavior in the ith round (for any $i = 1, \ldots, k$): the round should fail if the client requests item j (of the N items) and succeed otherwise. The paper introduces security definitions to include the selective failure problem and then propose two protocols to solve the problem under the new definitions.

Coull et al. [6] propose an access control oblivious transfer problem. Specifically, the server wants to enforce access control policies on oblivious transfers performed on the data stored: The client should only access fields for which it has the credentials. However, the server should not learn which credentials the client has used and which items it accesses.

Camenisch et al. [3] propose another solution that makes use of capabilities to enable clients to obliviously transfer items from a server. Regardless of the outcome of the interaction between a client and a server, the server does not learn which capabilities the client has. Moreover, the client retrieves the item only if it has enough capabilities to do so. Note however that this is different from our solution, since our solution also allows clients to obliviously write/modify items they can access. Thus, an oblivious transfer is not sufficient.

3 Model

Let O be a database owner and S be a server that stores the database. In its simplest form, the database is stored as a set of n pairs, $D = \{(id_1, v_1), \ldots, (id_n, v_n)\}$, where id denotes a unique identifier and v is the value stored under it. We will assume that the data owner knows all the IDs of items stored in his database (or has a an efficient way to compute them directly when needed). A set of *clients* $\mathcal{C} = \{C_1, \ldots, C_c\}$ is given access to items from D. In our approach, the database owner O delegates the rights to access items by handing out certain capabilities. We focus on the management of individual items, where a client is provided with access to a single item at a time. While this model can be extended to handle multiple items (e.g., request access to a contiguous range between id_1 and id_2 or to all items in a table column whose values exceeds v_1), we prefer our model for simplicity of exposition. We further assume that each client has a secure communication channel to communicate with the data owner. We give a construction for an oblivious database *D-ORAM* which supports the following operations:

- *Setup()*: Operation called by the owner to generate the initial *D-ORAM*.
- *Store(id, v)*: Operation that allows the owner to insert a new (id, v) pair into the *D-ORAM*.
- *DStore(id, C, ctr, ctr_C)*: Operation that allows the owner to insert a new dummy item for client C into the *D-ORAM*.

- *Delegate(C,id,op)*: Delegate to a client C the right to access an item *id* with operation *op* (*Read*, *Write* or *Insert*).
- *Read(id,cap)*: Access the value of *id*, thereby using capability *cap*.
- *Write(id,newV,cap)*: Modify the value stored under *id* to *newV*, thereby using capability *cap*.
- *Insert(id,v,cap)*: Insert a value v under *id* using capability *cap*.
- *Reshuffle(ctr)*: Reshuffle the *D-ORAM*, where *ctr* stores the number of reshuffles performed so far on the *D-ORAM*.

We note that it is further possible to revoke access rights; this can be done efficiently by changing the item key k_{id}. After changing k_{id}, the data owner sends the new key to all clients who were allowed to access the item and were not revoked access rights. To efficiently distribute the new key, we suggest to use broadcast encryption (see Section 5.2).

In our analysis, we assume an honest but curious server. The server is trusted to run any protocol correctly, while trying to collect additional information (access patterns or values accessed). We further assume the clients to be purely malicious: They can try to read items they cannot access, modify items even if they only have the right to read them, or learn about the access patterns of other clients. However, we guarantee these strong constraints only for items for which no permissions were given to a corrupted client. Note that this is a natural assumption: It is impossible to prevent a malicious client from publishing an item's content via other channels or to reveal that an item has been accessed. Furthermore we assume that the owner is trusted – he knows which clients can access which items, and he has full control over the database if desired.

Before building a delegated ORAM, we need to define its security properties. In order to achieve security goals against the server, these need to capture all the security guarantees offered by the standard, single-client ORAM. We therefore require the *D-ORAM* to satisfy the security properties against a curious server as outlined in [10] and the following security properties against malicious clients:

- *Access Security*: An *D-ORAM* offers *Access Security*, if no client can read or write an item $id \in$ *D-ORAM* without having proper capabilities.
- *Access Privacy*: We say that an item *id* in *D-ORAM* has been compromised, if there exists a corrupted client C_M with access to *id*. An *D-ORAM* offers *Access Privacy*, if for any item $id \in$ *D-ORAM*, which has not been compromised, no client without access to *id* can tell with non-negligible probability whether the item has been accessed, or not.

4 The Delegated ORAM Solution

Our solution is built on the "square root" ORAM variant described in Section 2 and relies on a novel use of capabilities. The data owner O issues a capability allowing a client C to access a certain item in the *D-ORAM*. Recall that in the square root ORAM, the database stores $n + \sqrt{n}$ items, where n items are "real"

and $\sqrt{n}$ items are dummy values. Thus, in our solution, each client is assumed to have access to $\sqrt{n}$ dummy or private items – real items that no other client can access. This increases storage at S by $c\sqrt{n}$, where c is the number of clients. In the following we will make extensive use of a buffer called "cache" buffer of size $\sqrt{n}$ stored at S. The buffer starts empty.

Each item *id* stored in the *D-ORAM* has a key associated with it, denoted by k_{id}. The item is stored encrypted with k_{id}, providing confidentiality from S. O either stores all keys or is able to compute them on demand (e.g., using a private, general purpose database key and a pseudo-random generator). Each item is stored in the *D-ORAM* as a $(tag, v, keybox)$ triplet, where *tag* is a public pseudo random string (derived from *id* as shown below) used to retrieve the item from the *D-ORAM*, *keybox* is an encrypted version of the item key k_{id} which will be used during the reshuffle and v denotes an encryption of the actual database item. The latter includes the item *id*, the actual value stored under this id, a version number for this item (which will be incremented upon each write operation) as well as a signature which allows to verify that the item-value was written correctly. C is allowed to access an item *id* only if it knows k_{id}. Thus, a capability for *id* needs to include k_{id}.

Tag Generation. Each item *id* in the database (including the dummy values) is assigned a tag, chosen pseudorandomly. Note that, if these tags were chosen uniformly at random, after each reshuffle O would have to notify each client C of the new tags assigned to items (including the dummy ones) it can access. To avoid this, we compute the tags as $t(id) := h(id, ctr, k_{id})$, where h is a publicly known pseudo random function, *ctr* is a counter, which counts the number of reshuffle operations performed so far, and k_{id} is the secret key corresponding to item *id*. This ensures that clients allowed to access item *id* (i.e., that know k_{id}) will be able to determine its tag after a reshuffle.

Keeping track of the tags for dummy items is done similarly. Each client maintains a personal counter ctr_C, indicating the number of unused dummy items. Using a unique client dummy password d_C, the current value of the counter *ctr* (which counts the number of reshuffle operations) and the personal counter ctr_C we compute the tags as $h(d_C, ctr, ctr_C)$. The passwords d_C will be unique for each client, known to only client and the data owner.

4.1 D-ORAM Operations

The *Setup* operation is called by the owner before the first *D-ORAM* operation is performed to populate the RAM.

Setup(). Initially, O calls the operation $\mathbb{G}$ of an identity based signature scheme, which outputs a master secret $\mathcal{M}_S$ and a public master key $\mathcal{M}_P$. He further sets $ctr := 0$ and chooses a symmetric key k_O which will be used exclusively by O. Next, he calls the *Store* and *DStore* $n + c\sqrt{n}$ times, once for each data or dummy item that needs to be stored in the *D-ORAM*. Notice that O will add these items in random order to the *D-ORAM* (to hide from the server which of them are dummy items). Furthermore, he allocates an empty cache buffer.

Store(id, v, C, ctr)**.** *Store* is executed by O when a new data item is to be inserted into the *D-ORAM*. O generates a secret key $k_{id} \in \{0,1\}^k$ and uses $\mathbb{K}_{\mathcal{M}_S}$ to generate a private key s_{id} for the value id. Further, O generates the tag $t(id)$ and outputs $(t(id), ev, E_{k_O}(k_{id}))$, where $ev = E_{k_{id}}(id, v, 0, \mathbb{S}(s_{id}, (v, 0))$ is the encryption of item id with version number 0. The value $E_{k_O}(k_{id})$ will help O to recover the decryption key when presented with the encrypted item. Finally, O asks the server to insert this tuple in the database.

DStore(id, C, ctr, ctr_C)**.** The owner executes this operation to insert a dummy item for some client $C = C_i$. *DStore* generates a tag $tag = h(d_C, ctr, ctr_C)$ for client C and counters ctr, ctr_C and a string s having the same distribution as the output of $E(\cdot)$. Finally, the triplet $(tag, s, E_{k_O}(\text{“}dummy\text{”}, C, ctr))$ is added to the *D-ORAM*.

We now describe the capability issuing operation, performed by O.

Delegate(C, id, op)**.** This operation outputs a capability *cap* which can be used in *Read*, *Write* or *Insert* operations. First generate the value k_{id} just like in the *Store* operation. Next, if $op =$ *Read*, output the tuple (id, k_{id}) and return. If $op =$ *Write*, generate the secret signing key s_{id} (just like in *Store*), output (id, k_{id}, s_{id}) and return. If op=*Insert*, output $(id, k_{id}, s_{id}, E_{k_O}(k_{id}))$ and return.

The *Read*, *Write* and *Insert* operations behave similarly to their basic ORAM variant. They are executed by a client C.

Read(id, cap)**.** Given a capability $cap = (id, k_{id})$, scan all cache buffer items starting from the most recently added. Retrieve each element as $(value, keybox)$. Decrypt each element *value* using the key k_{id}. If any decryption has the format (id, v, ver, sig), then check that $\mathbb{V}_{\mathcal{M}_P}(sig, id, (v, ver))$ verifies correctly. Note that since we are using an identity based signature scheme, each signature can be verified by using the value id and the master public key $\mathcal{M}_P$. If the check does not verify, discard the item and continue with the next item. If no correct item is found, compute the tag $t(id)$ and request the item with this tag from the *D-ORAM* database, obtain element $(tag, value, keybox)$ and decrypt its second field *value*. If the desired element had been found in the cache buffer, request the next unused dummy item from the *D-ORAM* database and obtain $(td, s, keybox)$ – discarding values td and s immediately while storing the value *keybox*. If all verifications pass and the decryption has been performed correctly, use the value v as the actual item value. Finally, re-encrypt the message $m = E_{k_{id}}(id, v, ver, sig)$ using k_{id}. It is necessary to re-encrypt this message before storing the item back into the buffer, to hide from the server whether it was found in the buffer or had been retrieved from the main database. Insert the result into the cache buffer along with the value *keybox* of the item which was requested from the main *D-ORAM* database (note that we always use the value *keybox* derived from the main database – real or dummy item – in order to keep all *D-ORAM* accesses indistinguishable). Output v and return.

Write$(id, newV, cap)$**.** Given a capability $cap = (k_{id}, id, s_{id})$, proceed as in *Read*, except that the value appended to the cache buffer is $E_{k_{id}}(id, newV, ver+1, \mathbb{S}(s_{id}, (newV, ver+1))$, where ver is the version number of the most recent item id when scanning the buffer and the main database.

Insert(id, v, cap)**.** For a given capability $cap = (id, k_{id}, s_{id}, E_{k_O}(k_{id}))$, append the tuple $(m, keybox) = (E_{k_{id}}(id, v, 0, \mathbb{S}(s_{id}, (v, 0)), E_{k_O}(k_{id}))$ to the cache buffer. Note that, by adding items in this manner, the server will notice when an insert has occurred. This problem can be prevented by adding sufficiently many dummy items to the initial ORAM and replacing them with real items whenever *Insert* is called. In fact, this can as well be done incrementally: If it is known that the database on average grows by k items per epoch, the data owner can add additional k dummy items during each re-shuffle operation. This efficiently hides the time the *Insert* occurred. To support incremental inserts, some minor adaptions to our construction are necessary. In particular, the form of dummy elements needs to be changed slightly to allow the data owner to recover the correct value *keybox* during the reshuffle.

Reshuffle(ctr)**.** The database *Reshuffle* operation is performed by O. The reshuffle is performed in five steps:
Step 1: Use $E(\cdot)$ to encrypt each item in the *D-ORAM* (including the buffer) with a fresh session key k_{rs}, used exclusively to perform the reshuffle. Note that this essentially works like a second layer of encryption for items in the database. Thus, in steps *2 - 4* we will always assume that items are first decrypted using k_{rs} when accessed by the data owner, and encrypted again before stored back on the server.
Step 2 (Clean the Cache): O verifies the validity of each updated item: items which fail to verify correctly should never appear in the main database. Download each element $(v, keybox)$ from the buffer, starting at the last inserted item. Perform the following actions:

- If $D_{k_O}(keybox) = k_{id}$ and $D_{k_{id}}(v)$ correctly decrypts and verifies to a valid item id, continue by scanning the earlier items in the buffer. Mark items with the same id for deletion.
- If $D_{k_O}(keybox)$ contains "dummy", scan the earlier items in the buffer until a valid value *keybox*, containing the correct key k_{id}, is found. Update the *keybox* encryption to $E_{k_O}(k_{id})$ and continue.
- Otherwise (e.g. if the signature can not be verified or if no valid key k_{id} could be determined), mark the item as invalid and continue.

Step 3: Read each item stored in the *D-ORAM*, generate a new tag $t(id)$ for it and store it back.
Step 4: Perform the re-shuffle operation as described in the basic square-root ORAM solution [10], i.e. obliviously update the database items' values according to the buffers, re-encrypt them using the corresponding key k_{id}, and obliviously permute the database locations.
Step 5: Decrypt each item in the *D-ORAM* with the session key k_{rs}.

4.2 Security Analysis

We discuss the security of our construction as introduced in Section 3.

Security against a curious server. The proof is identical to the one for ORAM with only one client [10]. It is easy to verify that, from the server's point of view, every time the *D-ORAM* is accessed, all operations are performed in precisely the same way. In particular, in the reshuffle the steps $1, 2, 3$ and 5 are performed in the same deterministic way in each epoch, while step 4 consists of the reshuffle used in the single client ORAM [10]. All values the server can see are pseudo random and therefore do not reveal information to the server.

Security against malicious clients. The construction achieves D-ORAM *Access Security*: First, notice that an unauthorized attempt to overwrite an item can be detected and the original item's value retrieved by any other client who is allowed to access this item. Even if a corrupted client knew k_{id}, the unforgeability property of the signature scheme ensures that without knowing s_{id}, he cannot produce a valid signature of a new item's value v. Hence, if a client finds that supposedly a new item's value is not correctly signed, he simply uses the last one that passes the signature verification in his computation. Also, in the reshuffle phase, the owner ensures that the items are updated to their correctly signed values. Notice further that IND-CPA security of the encryption scheme guarantees confidentiality for each item, in the sense that no collusion of clients without the capability to read this item can learn its value.

Furthermore, the solution also provides D-ORAM *Access Privacy*: To show that a client, who cannot access any of the items in a set, learns nothing about the computation on them, we use the same argument as in the case of the server: In case that no malicious client compromised the privacy of item id, every access to this item is indistinguishable from a random access to the *D-ORAM* for all clients who are not able to access item id.

4.3 Complexity

Using a Batcher network to shuffle a database containing n regular and $d = c\sqrt{n}$ dummy items requires $O((n+d)\log^2(n+d))$ comparisons. In addition, a reshuffle requires $O(n+d)$ operations (encrypt/decrypt each item once, update the buffer). When manipulating one item, $O(\sqrt{n})$ items need to be read. Hence, between two reshuffles, $O((n+d)\log^2(n+d) + n + d) = O((n+d)\log^2(n+d))$ operations are needed. We therefore get the amortized complexity to be of order $O(\frac{n+d}{\sqrt{n}}\log^2(n+d)) = O((\sqrt{n}+c)\log^2(n+d))$, where we assume that c is a small constant compared to $\sqrt{n}$.

5 Discussion

5.1 Beyond a Curious Server

While above we considered the case of a curious yet otherwise honest server, here we discuss also some insights into malicious server behavior.

It makes little sense to handle outright denial of service behavior at this level, as the server has many natural avenues at his disposal to restrict service, including simply shutting down. More interesting to explore are attacks in which the server illicitly and *undetectably* manages to satisfy its curiosity by behaving incorrectly. We distinguish a set of scenarios, some of which are discussed in the following.

Fork Consistency. The server may attempt to partition the set of clients, and maintain separate versions of the database state (buffer, main database) for each partition. This partitioning attack has been examined in previous literature; if there are non-inter-communicating asynchronous clients, the best that can be guaranteed is fork consistency [15]. This is not as weak of a guarantee as it may appear, as once the provider has created a partition, the provider must block all future communication between partitioned clients, or else the partition will be immediately detected.

Altering Responses. Additionally, the server may attempt to substitute messages and previously read data for new requests. This can naturally be addressed by a combination of minimal client state based mechanisms that can checksum the server responses. For instance, the client could deploy Merkle tree based approaches coupled with item versioning to defend against such attacks. As numerous existing efforts already addressed such mechanisms we chose not to detail them here.

Timing Attacks. In such attacks, the server measures the time intervals taken by a client to parse the buffers and to access the ORAM database. This might enable the server to learn which type of operation was performed, and/or where the desired item was found. We suggest to prevent this attack by introducing the requirement that each access to an item stored on the server (buffer or database) takes the same time. This can be achieved by each client using additional timers to "uniformize" inter-request times.

5.2 Efficient Access Right Updates Using Broadcast Encryption Schemes

In the protocol construction, we omitted the details of access right updates. A naive solution to revoke the access rights of a set of clients to a particular item, is to change the item's secret key k_{id} and broadcast a new key, encrypted with the public keys of all clients in the target set. If there are c clients, this solution potentially requires to encrypt the item's secret key with $\Theta(c)$ public keys.

A more efficient way to solve the problem is to use *broadcast encryption schemes* [2,7,16,11]. The main idea of broadcast encryption schemes is to associate keys to the subsets of clients and represent any set of privileged clients as a union of these subsets. In the naive solution, each client is given a unique key. A better result can be achieved by building a binary tree of keys with clients representing its leaves, and give each client all private keys on the path from the corresponding leaf to the root. The privileged set of clients is then covered by a set of subtrees and the (public key, private key)-pairs in the roots of these

subtrees are used for encryption/decryption of an item's content. This scheme is still inefficient if access right updates involve revoking small sets of clients. For instance, to revoke a single client, $\log c$ subtrees might be needed to cover all the remaining clients.

A better performance is achieved in [16,11]. Following the approach of Halevy and Shamir [11], any set of r clients can be revoked by the owner, broadcasting only $O(r)$ (at most $4r$) encryptions. In this scheme, each client is given $O(\log^{1+\epsilon} c)$ private keys ($O(\log^{3/2} c)$ in the basic case of practical interest) and performs $O(\log c)$ decryption operations.

6 Conclusions

In this paper we study the problem of delegating access to an outsourced private database to multiple clients, while preserving the access privacy of all involved entities. Our solution extends existing ORAM flavors with the notion of capabilities, allowing data owners to delegate and revoke permissions and clients to privately read, write and insert items. We show that our solution provides reasonable security guarantees and protects the privacy of the involved parties. We further note that more efficient versions of ORAM can be constructed based on the so called "poly-log"-solution [10]. While in this paper we provide a basic ORAM solution which allows to give access to the ORAM to multiple parties, it might be interesting to investigate whether similar solutions could be applied to the more efficient "poly-log"-solution. We leave these, as well as further optimizations, as future work.

Acknowledgments. We thank Ian Goldberg for insightful discussions and the anonymous reviewers of FC'11 for helpful comments. Sion and Sotakova were supported by NSF through awards 0937833, 0845192, and 0803197, as well as by CA Technologies, Xerox, IBM and Microsoft Research. This work was supported by CASED (http://www.cased.de), the German Research Foundation (DFG) and DAAD.

References

1. Asonov, D. (ed.): Querying Databases Privately. LNCS, vol. 3128. Springer, Heidelberg (2004)
2. Berkovits, S.: How to Broadcast a Secret. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 535–541. Springer, Heidelberg (1991)
3. Camenisch, J., Dubovitskaya, M., Neven, G.: Oblivious transfer with access control. In: CCS 2009: Proceedings of the 16th ACM Conference on Computer and Communications Security (2009)
4. Camenisch, J.L., Neven, G., Shelat, A.: Simulatable Adaptive Oblivious Transfer. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 573–590. Springer, Heidelberg (2007)
5. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: IEEE Symposium on Foundations of Computer Science, pp. 41–50 (1995)

6. Coull, S., Green, M., Hohenberger, S.: Controlling Access to an Oblivious Database Using Stateful Anonymous Credentials. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 501–520. Springer, Heidelberg (2009)
7. Fiat, A., Naor, M.: Broadcast Encryption. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 480–491. Springer, Heidelberg (1994)
8. Gasarch, W.: A WebPage on Private Information Retrieval, `http://www.cs.umd.edu/~gasarch/pir/pir.html`
9. Gasarch, W.: A survey on private information retrieval, `http://citeseer.ifi.unizh.ch/gasarch04survey.html`
10. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM 43, 431–473 (1996)
11. Halevy, D., Shamir, A.: The LSD Broadcast Encryption Scheme. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 47–60. Springer, Heidelberg (2002)
12. IBM. IBM 4764 PCI-X Cryptographic Coprocessor (2007), `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`
13. Iliev, A., Smith, S.W.: Private information storage with logarithmic-space secure hardware. In: Proceedings of i-NetSec 2004: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems, pp. 201–216 (2004)
14. Jarecki, S., Shmatikov, V.: Handcuffing Big Brother: an Abuse-Resilient Transaction Escrow Scheme. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 590–608. Springer, Heidelberg (2004)
15. Li, J., Krohn, M., Mazières, D., Shasha, D.: Secure Untrusted Data Repository (SUNDR). In: OSDI 2004, pp. 121–136 (2004)
16. Naor, D., Naor, M., Lotspiech, J.B.: Revocation and Tracing Schemes for Stateless Receivers. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 41–62. Springer, Heidelberg (2001)
17. Paterson, K.G.: Id-based signatures from pairings on elliptic curves. Electronics Letters 38, 1025–1026 (2002)
18. Pinkas, B., Reinman, T.: Oblivious ram revisited. In: Proceedings of the 30th International Cryptology Conference (2010) (to appear)
19. Sion, R., Carbunar, B.: On the Computational Practicality of Private Information Retrieval. In: Proceedings of the Network and Distributed Systems Security Symposium (2007); Stony Brook Network Security and Applied Cryptography Lab Tech Report 2006-06
20. Wang, S., Ding, X., Deng, R.H., Bao, F.: Private Information Retrieval using Trusted Hardware. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 49–64. Springer, Heidelberg (2006)
21. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: CCS 2008: Proceedings of the 15th ACM Conference on Computer and Communications Security (2008)
22. Williams, P., Sion, R., Carbunar, B.: Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In: ACM Conference on Computer and Communication Security CCS (2008)