Write Once Read Many Oblivious RAM

Bogdan Carbunar, Radu Sion

Abstract—We introduce WORM-ORAM, a first mechanism that combines Oblivious RAM (ORAM) access privacy and data confidentiality with Write Once Read Many (WORM) regulatory data retention guarantees. Clients can outsource their database to a server with full confidentiality and data access privacy, and, for data retention, the server ensures client access WORM semantics. In general *simple confidentiality* and *WORM* assurances are easily achievable e.g., via an encrypted outsourced data repository with server-enforced read-only access to existing records (albeit encrypted). However, this becomes hard when also *access privacy* is to be ensured – when client access patterns are necessarily hidden and the server cannot enforce access control directly. WORM-ORAM overcomes this by deploying a set of zero-knowledge proofs to convince the server that all stages of the protocol are WORM-compliant.

1 Introduction

Regulatory frameworks impose a wide range of policies in finance, life sciences, health-care and the government. Examples include the Gramm-Leach-Bliley Act [1], the Health Insurance Portability and Accountability Act [2] (HIPAA), the Federal Information Security Management Act [3], the Sarbanes-Oxley Act [4], the Securities and Exchange Commission rule 17a-4 [5], the DOD Records Management Program under directive 5015.2 [6], the Food and Drug Administration 21 CFR Part 11 [7], and the Family Educational Rights and Privacy Act [8]. Over 10,000 regulations are believed to govern the management of information in the US alone [9].

A recurrent theme to be found throughout a large part of this regulatory body is the need for assured lifecycle storage of records. A main goal there is to support WORM semantics: once written, data cannot be undetectably altered or deleted before the end of its regulation-mandated life span. This naturally stems from the perception that the primary adversaries are powerful insiders with superuser powers coupled with full access to the storage system. Indeed much recent corporate malfeasance has been at the behest of CEOs and CFOs, who also have the power to order the destruction or alteration of incriminating records [10].

Major storage vendors have responded by offering compliance storage and WORM products, for on-site deployment, including IBM [11], HP [12], EMC [13]. Hitachi Data Systems [14], Zantaz [15], StorageTek [16], Sun Microsystem [17] [18], Network Appliance [19]. and Quantum Inc. [20].

However, as data management is increasingly outsourced to third party "clouds" providers such as Google, Amazon and Microsoft, existing systems simply do not work. When outsourced data lies under the incidence of both mandatory *data retention* regulation and privacy/confidentiality concerns – as it often does in outsourced contexts – new enforcement mechanisms are to be designed.

This task is non-trivial and immediately faces an apparent contradiction. On the one hand, data retention regulation stipulates that, once generated, data records cannot be erased until their "mandated expiration time", *even by their rightful creator* – history cannot be rewritten. On the other hand, access privacy and confidentiality in outsourced scenarios mandate non-disclosure of data and patterns of access thereto to the providers' servers, and can be achieved through "Oblivious RAM" (ORAM) based client-server mechanisms [21], [22]. Yet, by their very nature, existing ORAM mechanisms allow clients unfettered read/write access to the data, including the ability to alter or remove previously written data records – thus directly contradicting data retention requirements.

Basic *confidentiality* and *WORM* assurances are achievable e.g., via traditional systems that could encrypt outsourced data and deploy server-enforced read-only access to data records once written. Yet, when also *access privacy* is to be ensured, client access patterns become necessarily hidden and the server cannot enforce WORM semantics directly.

In this paper we introduce WORM-ORAM, a first mechanism that combines the access privacy and data confidentiality assurances of traditional ORAM with Write Once Read Many (WORM) regulatory data retention guarantees. Clients can outsource their database to a server with full confidentiality and data access privacy, and, for data retention, the server ensures client access WORM semantics, i.e., specifically that client access is append-only: – once a data record has been written it cannot be removed or altered even by its writer.

WORM-ORAM is built around a set of novel efficient zero knowledge (ZK) proofs. The main insight is to allow the client unfettered ORAM access with full privacy to the server-hosted encrypted data set while simultaneously proving to the server *in zero-knowledge* – at all stages of the ORAM access protocol – that no

• *Bogdan Carbunar is with the School of Computing and Information Sciences at the Florida International University, Miami, FL. E-mail: carbunar@gmail.com.*
• *Radu Sion is with the Computer Science Department at the Stony Brook University, Stony Brook, NY. E-mail: sion@cs.stonybrook.edu.*

existing records are overwritten and WORM semantics are preserved.

We show that our solution does not change the computational complexity of existing ORAM implementations. Our implementations show that the end-to-end cost of a read operation is 10s and the amortized cost of a shuffle is 47s. These costs compare favorably with the costs imposed by classic ORAM solutions that do not offer WORM assurances. Future work focuses on reducing these overheads toward true practical efficiency.

2 Related Work

2.1 Oblivious RAM

This paper extends the work of [23] with novel constructions that provide indistinguishability for the read and write accesses, detailed descriptions of essential components such as element expiration and with proofs for the assurances provided by the solution including the zero knowledge properties.

Oblivious RAM [21] provides access pattern privacy to clients (or software processes) accessing a remote database (or RAM), requiring only logarithmic storage at the client. The amortized communication and computational complexities are $O(log^3 n)$. Due to a large hidden constant factor, the ORAM authors offer an alternate solution with computational complexity of $O(log^4 n)$, that is more efficient for all currently plausible database sizes.

In ORAM, the database is considered a set of n encrypted blocks and supported operations are read(id), and write(id, $newvalue$). The data is organized into $log_4(n)$ levels, as a pyramid. Level i consists of up to 4^i blocks; each block is assigned to one of the 4^i buckets at this level as determined by a hash function. Due to hash collisions each bucket may contain up to $\log n$ blocks.

ORAM Reads/Writes. To obtain the value of block id, the client must perform a read query in a manner that maintains two invariants: (i) it never reveals which level the desired block is at, and (ii) it never looks twice in the same spot for the same block. To maintain (i), the client always scans a single bucket in every level, starting at the top (Level 0, 1 bucket) and working down. The hash function informs the client of the candidate bucket at each level, which the client then scans. *Once the client has found the desired block, the client still proceeds to each lower level, scanning random buckets instead of those indicated by their hash function*. For (ii), once all levels have been queried, the client re-encrypts the query result with a different nonce and places it in the *top* level. This ensures that when it repeats a search for this block, it will locate the block immediately (in a different location), and the rest of the search pattern will be randomized. The top level quickly fills up; how to dump the top level into the one below is described later.

Writes are performed identically to reads in terms of the data traversal pattern, with the exception that the new value is inserted into the top level at the end.

Level Overflow. Once a level is full, it is emptied into the level below. This second level is then re-encrypted and re-ordered, according to a new hash function. Thus, accesses to this new generation of the second level will hence-forth be completely independent of any previous accesses. Each level overflows once the level above it has been emptied 4 times. Any re-ordering must be performed obliviously: once complete, the adversary must be unable to make any correlation between the old block locations and the new locations. A sorting network is used to re-order the blocks.

To enforce invariant (i), note also that all buckets must contain the same number of blocks. For example, if the bucket scanned at a particular level has no blocks in it, then the adversary would be able to determine that the desired block was *not* at that level. Therefore, each re-order process fills all partially empty buckets to the top with *fake* blocks. Since every block is encrypted with a semantically secure encryption function, the adversary cannot distinguish between fake and real blocks.

Pinkas and Reinman introduce in [24] a mechanism to provide $O(\log^2 n)$ oblivious access with only logarithmic client storage. The collision-free Cuckoo hash from [25] is employed to remove a $\log n$ factor of server storage (and eliminate the corresponding query overhead). Data is stored ORAM-like, in a pyramid-shaped set of levels, with queries proceeding downward interactively.

Unfortunately, the premise of this idea is flawed. The construction of the Cuckoo hash function at a given level considers only the data to be stored at a given level, not the data already stored at lower levels. This property is necessary to achieve the desired time complexity. Thus, queries for the lower-level data have a significant chance of sharing locations with data at the current level. Because the Cuckoo hash otherwise avoids collision, any time such an occurrence is observed indicates leaks access information. That is, when the adversary sees two queries access the same pair of hash table locations, it learns that at least one of those queries was, in fact, for lower-level (less recently accessed) data. This immediately violates access privacy. The authors acknowledged this problem.

2.2 Private Information Retrieval

Private Information Retrieval (PIR) [26] 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. [26] proved that in an information theoretic setting, any single-server solution requires $\Omega(n)$ bits of communication. When the information theoretic guarantee is relaxed single-server solutions with better complexities exist; an excellent survey of PIR can be found online [27], [28]. Recently, we have shown [29] 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.

3 Model and Preliminaries

3.1 Deployment and Threat Model

In the deployment model for networked compliance storage, a legitimate client creates and stores records with a (potentially untrusted) remote WORM storage service. These records are to be available later to both the client for read as well as to auditors for audits. Network layer confidentiality is assured by mechanisms such as SSL/IPSec. Without sacrificing generality, we will assume that the data is composed of equal-sized blocks (e.g., disk blocks, or database rows).

At a later time, a previously stored record's existence is regretted and the client will do everything in her power – e.g., attempt to convince the server to remove the record – to prevent auditors from discovering the record. The main purpose of a traditional WORM storage service is to defend against such an adversary.

Moreover, numerous data regulations feature requirements of "secure deletion" of records at the end of their mandated retention periods. Then, in the WORM adversarial model the focus is mainly on preventing clients from "rewriting" history, rather than "remembering" it. Additionally, we prevent the rushed removal of records before their retention periods. Thus, the traditional Write-Once Read-Many (WORM) systems have the following properties:

- Data records may be written by clients to the server once, read many times and not altered for the duration of their life-cycle.
- Records have associated mandatory expiration times. After expiration, they should not be accessible for either audit or read purposes.
- In the case of audits, stored data should be accessible to auditors even in the presence of a non-cooperating client refusing to reveal encryption keys.

We assume however that client participate correctly in any record expiration protocol. This is reasonable to assume because the regulatory compliance scenario allows clients always to by-pass the server-enforced storage service and store select records elsewhere.

Additionally, when records and their associated access patterns are sensitive they need to be concealed from a curious server that has incentives to illicitly gain information about the stored data and access patterns thereto. The main purpose of WORM-ORAM is to enable WORM semantics while preserving data confidentiality and access pattern privacy. Then, data records have to be encrypted (*confidentiality*) and the server should not distinguish between different read or write operations targeting the same or different data records (*access privacy*). We assume that the server is allowed to distinguish between record expiration and read/write operations. As the regulatory storage provider, the server is the main enforcer of WORM semantics and record expiration. The server is assumed to not collude with clients illicitly desiring to alter their data.

We consider a server S with $O(N)$ storage and a client C with $O(\sqrt{N}\log N)$ local storage. The client stores $O(N)$ items on the server. We denote the regulatory compliance auditor by $\mathcal{A}$.

3.2 Cryptography

We require several cryptographic primitives with all the associated semantic security [30] properties including: a secure, collision-free hash function which builds a distribution from its input that is indistinguishable from a uniform random distribution, a semantically secure cryptosystem (Gen, Enc_k, Dec_k), where the encryption function Enc generates unique ciphertexts over multiple encryptions of the same item, such that a computationally bounded adversary has no non-negligible advantage at determining whether a pair of encrypted items of the same length represent the same or unique items, and a pseudo random number generator whose output is indistinguishable from a uniform random distribution over the output space.

The Decisional Diffie-Hellman (DDH) assumption over a cyclic group G of order q and a generator g states that no efficient algorithm can distinguish between two distributions (g^a, g^b, g^{ab}) and (g^a, g^b, g^c), where a, b and c are randomly chosen from $\mathbb{Z}_q$.

An integer v is said to be a *quadratic residue* modulo an integer n if there exists an integer x such that $x^2 = v \ mod \ n$. Let QRA be the quadratic residuosity predicate modulo n. That is, $QR(v,n) = 1$ if v is a residue mod n and $QR(v,n) = 0$ if v is a quadratic non-residue. Given an odd integer $n = pq$, where p and q are odd primes, the quadratic residuosity (QR) assumption states that given n but not its factorization and an integer v whose Jacobi symbol $(v|n) = 1$ it is difficult to determine whether $QR(v,n)$ is 1 or 0.

Notations:: Let $n = pq$ be a large composite, where p and q are primes. Let $\phi(n)$ denote the Euler totient of n. We will use $x \in_R A$ to denote the random uniform choice of x from the set A. Given a value m, let $\mathcal{P}(m)$ denote the group of permutations over the set $\{0,1\}^m$. Let $\mathbf{k} < |n|$ be a security parameter. Let N denote the set of elements stored in the ORAM. Let $\mathbb{W}_m$ be the universe of all sets of m quadratic residues.

4 Solution Overview

A WORM-ORAM system, consists of two ORAMs (W-ORAM,E-ORAM) and a set of operations (Gen, Enc, Dec, RE, Write, Read, Expire, Shuffle, Audit) that can be used to access the ORAMs. The client needs to store elements at the server while preserving the privacy of its accesses and allowing the server to preserve the data's WORM semantics. W-ORAM serves this purpose: it is used by the client to store $(label, element)$ pairs.

We organize time into epochs: each element stored at the server expires in an integer number of epochs, as determined by the client. The client needs to remember

the expiration time of each element stored in the W-ORAM. The client uses the E-ORAM to achieve this, to store expiration times of labels used to index elements stored in W-ORAM. When queried with a time epoch, E-ORAM provides a list of labels expiring in that epoch. The labels are then used to retrieve the expiring elements from the W-ORAM.

The E-ORAM is stored and accessed as a regular ORAM [22]. It is used as an auxiliary storage structure by the client and it needs not be WORM compliant. The W-ORAM on the other hand stores actual elements and needs to be made WORM compliant. The W-ORAM stores two types of elements: "reals" and "fakes". A real element has a quadratic non-residue component, whereas a fake has a quadratic residue. Each time the ORAM is accessed, elements are re-encrypted to ensure access privacy. The client has then to prove in ZK that (i) an element is real or fake and (ii) a re-encrypted element decrypts to the same cleartext as the original element.

In the following we provide a detailed description for each operation mentioned above. We employ the classic ORAM operations described in Section 2.1 as the APIs for building our solution. Specifically, we use $Read_{ORAM}$ to denote the standard ORAM read operation, taking as input an ORAM and a label and returning an element stored under that label along with the list of all elements removed from the ORAM (including the one of interest). $Write_{ORAM}$ is the standard ORAM write operation, which takes as input a label and an element and stores the element indexed under the label. Note that in the standard ORAM implementation, both operations are performed in the same manner. Their operation is only different for the client. Finally, let OS be the standard ORAM re-shuffle operation (see Section 2), which takes as input a level id and generates a pseudo-random permutation of the re-encrypted elements at that level.

5 W-ORAM Element Encryption

We now define the operations for encrypting the elements to be stored in the WORM compliant ORAM.

Gen(k): Generate $p = 2p' + 1$, $q = 2q' + 1$ such that p, p', q, q' are primes. Let $n = pq$. Let G be the cyclic subgroup of order $(p-1)(q-1)$. DDH is believed to be intractable in G [31]. Let g be a generator of G. Let a be a random value and let $d = a^{-1} \ mod \ \phi(n)$. Let k be a random key in a semantically secure symmetric cryptosystem. Gen gives k, n, g, $h = g^a \in G$, p, q, a and d to the client and n, g, h to the server. Gen also gives k, p, q, a and d to the auditor.

Enc((x,T_{exp}),k,g,h,G,f)t: Encrypt an element of value x with expiration time T_{exp}, using the client's view of Gen's output as input parameters. The output of the operation is a tuple $(A, B) \in G \times G$ that can be stored on the W-ORAM. If $f = 0$, Enc generates a "real" W-ORAM element: the first field of such elements is a quadratic non residue, $QR(A, n) = 0$. The tuple is computed as follows. First, generate a random $r \in \{0,1\}^{\mathbf{k}}$ and use it to compute $M(x) = \{E_k(x), T_{exp}, \text{“}real\text{”}, r\}$ where $E_k(x)$ denotes the semantically secure encryption of item x with symmetric key k and "real" is a pre-defined string. The random r is chosen (using trial and failure) such that $QR(M(x), n) = 0$ (quadratic non residue mod n) whose Jacobi symbol is 1. Second, generate a random odd value $b \in_R \{0,1\}^{\mathbf{k}}$ and output the tuple $S(x) \in G \times G$ as

$$S(x) = (A, B) = (M(x)g^{2b}, h^{2b})).$$

$S(x)$ is said to be an "W-ORAM element", whose first field is the "encrypted element" and second field is called the "recovery key". Notice that since $M(x)$ is a QNR, $QR(M(x)g^{2b}, n) = 0$ with (Jacobi symbol) $(M(x)g^{2b}|n) = 1$.

If $f = 1$, Enc generates a "fake" W-ORAM element: the first field of fake elements is a quadratic residue, $QR(A, n) = 1$. To compute a fake element, Enc generates random $s, k \in_R \{0,1\}^{\mathbf{k}}$ and outputs the tuple $(s^2 \ mod \ n, k)$. That is, the first field in the pair is a quadratic residue, however, the "recovery key" is useless – does not recover a meaningful message.

Dec((A,B),d,k): Decrypt a real W-ORAM element, given the secret key $d = a^{-1}$. Compute $M = AB^{-d}$. M has format $\{E, T_{exp}, \text{“}real\text{”}, r\}$. The operation outputs the tuple $Dec_k(E), T_{exp}$.

RE(A,B): Re-encrypt element (A, B). Choose $u \in_R \{0,1\}^{\mathbf{k}}$, called re-encryption factor. Output pair $(A', B') = (Ag^{2u}, Bh^{2u})$. Note that knowledge of the message M encoded in (A, B) is not required. Alternatively, if M is known such that $A = Mg^{2b}$ and $B = h^{2b}$, then output $(A', B') = (Mg^{2u}, h^{2u})$. u can be specified as input parameter: $RE((A, B), u)$.

RE(L): Generalization of $RE((A, B))$, where $L = \{(A_1, B_1), .., (A_m, B_m)\}$ is a list of W-ORAM elements. Choose $\bar{u} = \{u_1, .., u_m\}$, such that $u_i \in_R \{0,1\}^{\mathbf{k}}$. $\bar{u}$ is called the re-encryption vector. Output $L' = \{RE((A_i, B_i), u_i)\}_{i=1..m}$. We also use the notation $L' = L\bar{u}$ and call L' a "correct re-shuffle" of L.

Note that Enc is semantically secure. The proof can be found in [32]. Similarly RE is IND-CPA: given two encryptions (A_0, B_0) and (A_1, B_1) of any two messages and a re-encryption $RE(A_b, B_b)$, $b \in_R \{0,1\}$ of one of the two encryptions, an attacker cannot guess b with non-negligible probability over 1/2.

6 The E-ORAM

The E-ORAM is a standard ORAM, storing labels indexed under expiration time epochs. The E-ORAM needs to provide C with the means to determine how many and which labels expire at a given time epoch and also to insert a new $(epoch, label)$ pair. This is achieved in the following manner. For each T_{exp} value used to index labels in E-ORAM, a *head* value is used to store the number of labels expiring at T_{exp}: $(T_{exp}, (label, counter))$. $label$ is the first label that was indexed under T_{exp}. Each of the remaining $c - 1$ labels is stored under a unique index: The ith label's index is (T_{exp}, i), that is, the label's

Algorithm 1 E-ORAM: Write new label under expiration time. Enumerate all labels expiring at a given time. V is the list of elements returned by a Read.

```
1.Write(E − ORAM : ORAM, T_exp : int, lbl : id)
2. (e, V) := Read_ORAM(E − ORAM, T_exp);
3. if (e = null) then
4.     e' := E_k(lbl, 1);
5.     Write_ORAM(T_exp, e');
6.     Write_ORAM(null, null);
7. else
8.     (l, c) := D_k(e);
9.     Write_ORAM(T_exp, E_k(l, c + 1));
10.    Write_ORAM((T_exp, c + 1), E_k(lbl));
11.fi
12.end
```

```
12.Enumerate(E − ORAM : ORAM, T_exp : int)
13.L : id[]; #store result labels
14.L := ∅;
15.(e, A) := Read_ORAM(E − ORAM, T_exp);
16.if (e! = null) then
17.  (l, c) := D_k(e);
18.  L := L ∪ l;
19.  for (i := 2; i ≤ c; i + +) do
20.    (e, A) := Read_ORAM(E − ORAM, (T_exp, i));
21.    l := D_k(e);
22.    L := L ∪ l;
23.  od
24.fi
25.return L;
26.end
```

expiration time concatenated with the label's counter at its insertion time.

We now present the most important operations for accessing the E-ORAM, Write and Enumerate.

Write(E-ORAM,T_{exp},label): Record the fact that $label$ expires at time T_{exp} (see Algorithm 1, lines 1-11). Read the element currently stored under T_{exp} (line 2). If no such element exists (line 3), generate an element encoding the fact that this label is the first to be stored under T_{exp} (line 4) and store it on the E-ORAM (line 5). Moreover, run a fake E-ORAM access (line 6), whose purpose will become clear in a few lines. If a label is already stored under T_{exp} (line 7), retrieve that label (l) along with the counter c that specifies how many labels are already expiring (stored in E-ORAM) at T_{exp} (line 8). Note that the read operation performed on line 2 removes this element from the E-ORAM. Since now $c+1$ labels expire at T_{exp}, store label l and the incremented counter in the E-ORAM under T_{exp} (line 9). Finally, store the input $label$ under an index consisting of a unique value: T_{exp} concatenated with $c+1$. This will allow the client to later enumerate all labels expiring at T_{exp} (see next). The reason for the fake E-ORAM write performed in line 6 is to make the two cases indistinguishable to the server: the E-ORAM is always accessed twice, independent of how many elements expire at T_{exp}.

Enumerate(E-ORAM,T_{exp}): Retrive all the labels in E-ORAM that expire at T_{exp} (see Algorithm 1, lines 12-26). First, initialize the result label list (line 13). Then, read the head label stored under T_{exp} along with the counter of labels expiring at T_{exp} (lines 14,16). If such an element exists (line 15), record the head label (line 17). Then, for each of the $c-1$ ($i = 2, .., c$) remaining labels, retrieve their actual value by reading from E-ORAM the element stored under a unique index consisting of T_{exp} concatenated with i. Note that $Enumerate$ removes all labels expiring at T_{exp} from E-ORAM ($Read_{ORAM}$ removes accessed elements).

7 W-ORAM Access Operations

7.1 Generating Labels

Elements in the standard ORAM model are stored as a pair $(label, value)$, where $label$ may denote a memory location or the subject of an e-mail. In our case to prevent the server from launching a dictionary attack, we use the a $Label(label, lkey)$ operation to generate labels. Besides the input $label$, $Label$ also uses a (random) labeling key, which is used to define a pseudo-random function F_{lkey}. The output of $Label$ coincides then with the output of $F_{lkey}(label)$. We now describe the main W-ORAM accessing operations.

7.2 Writing on the Server

Write((W-ORAM,E-ORAM,v,l,T_{exp},params): Store on the server a value v under a label l, with expiration time T_{exp}, using as input also the client's view of Gen's output, $params = k, g, h, G$ (see Algorithm 2 for the pseudo-code of this operation). Generate a new $label$ as described above (line 2) and call Enc to produce a W-ORAM tuple (A_u, B_u) (line 3). Generate a non-interactive zero knowledge proof of $QR(A_u, n) = 0$ (A_u's quadratic non-residuosity). If the proof verifies (line 5) the server inserts the tuple (A_u, B_u) in the top level of the W-ORAM (line 6) and stores $label$ under the tuple's expiration time T_{exp} in E-ORAM (see Section 6). Otherwise, the server aborts the protocol (line 10).

7.3 Reading from the Server

Read(W-ORAM,label): Using as input the W-ORAM and a $label$, return an element of format $(label, x, T_{exp})$ (see Algorithm 3). Perform on W-ORAM a standard ORAM read on the desired $label$ (line 2), returning both the W-ORAM element R of interest and the list L of elements (containing R) removed from the W-ORAM. If $label$ is stored in the W-ORAM (line 3), the client computes $U = (A_u, B_u)$, a re-encryption of R (line 3) and calls ZK-POR to prove in zero knowledge that U is a re-encryption of the only real element in L (line

4). ZK-POR is described in detail in Section 7.3.1. The server verifies in ZK that $QR(A_u, n) = 0$ and also the validity of the ZK-POR proof. If the proofs are valid (line 5), the server inserts U in the first level of the W-ORAM (lines 6-7). The client decrypts the desired element R and returns the result (line 8). If any proof fails (line 9) the server restores the W-ORAM to the state before the start of Read and returns error (lines 10-11).

7.3.1 Zero Knowledge Proof of ORAM Read.

We now present ZK-POR, the zero-knowledge proof of WORM compliance of the read operation performed on the W-ORAM. ZK-POR takes as argument the list L of elements removed from W-ORAM in line 2 of Algorithm 3 and U, the re-encryption of the real element from L. For simplicity of exposition, let us assume that L also contains the elements (scanned but not removed) from the first level of W-ORAM. Let m denote the number of elements in L, $m = O(\log N)$.

Let $L = \{(s_1^2, k_1)$,...,(s_{r-1}^2, k_{r-1}),$S(x_r)$,(s_{r+1}^2, k_{r+1}),.., $(s_m^2, k_m)\}$ where the elements are listed in the order in which they were removed from the W-ORAM. The client is interested in the item from the rth ORAM layer, $R = S(x_r)$. Let $S(x_r) = (M(x_r)g^{2t_r}, h^{2t_r}) = (A_r, B_r)$. Its first field is a quadratic non-residue. All other elements from L are fakes – their first field is a quadratic residue. Let $U = RE(R) = (M(x_r)g^{2u}, h^{2u}) = (A_u, B_u)$ be the re-encryption of $S(x_r)$. The following steps are executed s times between the client and the server.

Step 1: Proof Generation: The client selects a random permutation $\pi \in_R \mathcal{P}(m)$. The client generates $\bar{w} = \{w_1, ..w_m\}$, where each $w_i \in_R \{0,1\}^m$ is odd and generates the proof list $P = \pi(L\bar{w})$. Let $P = \pi\{(s_1^2 g^{2w_1}, k_1 h^{2w_1})$,..,$(A_r g^{2w_r}, B_r h^{2w_r})$, ..,$(s_m^2 g^{2w_m}, k_m h^{2w_m})\}$, where, $(A_r g^{2w_r}, B_r h^{2w_r})$ is a re-encryption of $S(x_r)$. The client sends P to the server. The client locally stores $(w_i, s_i g^{w_i})$, $i = 1..m$. As assumed in the model, The client has $O(\sqrt{N}\log N)$ storage which is sufficient to store $m = O(\log N)$ values.

Step 2: Proof Validation: The server flips a coin b. If b is 0, the client reveals $w_1, .., w_m$. The server verifies that all w_i are odd and $\forall (A_i, B_i) \in L$, $(A_i g^{2w_i}, B_i h^{2w_i}) \in P$. If b is 1, the client sends to the server the values $s_i g^{w_i}$, $i = 1..m, i \neq r$ along with the value $\Gamma = (t_r + w_r - u)$. Note that given $s_i^2 \ mod \ n$ and n's factorization, the client can easily recover s_i. The server verifies first that $(s_i g^{w_i})^2$, $i = 1..m, i \neq r$ occurs in the first field of exactly one tuple in P. That is, $m-1$ of the elements from P are fakes. The server then verifies that $(A_r g^{2w_r}, B_r h^{2w_r}) = RE((A_u, B_u), \Gamma)$. If any verification fails, the server outputs "error" and stops.

Analysis: We now present the following results, whose proofs can be found in [32].

Theorem 1: A correct execution of Read from W-ORAM has $O(\log N)$ complexity.

Theorem 2: ZK-POR is a zero knowledge proof system of Read $\in$ WORM. That is, Read is WORM compliant.

8 Access Indistinguishability

The solution previously described allows the server to distinguish between read and write operations. In this section we solve this problem, by creating a single operation, Access, that can be used to both read and write on the W-ORAM.

Access((W-ORAM,E-ORAM,v,l,T_{exp},params): If Access=Write, use l to generate a new $label$ (as described in Section 7.1) and insert $label$ under T_{exp} in the E-ORAM (using the Write operation described in Section 6). If Access=Read, perform a fake Write on the E-ORAM, consisting of three random accesses to the E-ORAM (one for a read and two for writes, see Section 6). Then, access all the elements in the top level of the W-ORAM and access and remove one element from each subsequent level. If Access=Write, all removed elements have to be fakes. If Access=Read, one of them is real (unless the read element was found in the top level). Let $L = \{(s_1^2, k_1)$,...,(s_{r-1}^2, k_{r-1}),$S(x_r)$, (s_{r+1}^2, k_{r+1}),..,$(s_m^2, k_m)\}$ be the list of elements accessed in the W-ORAM, where $S(x_r)$ may be the real element accessed by a Read or a fake if accessed by a Write. Then, generate two elements R and N and send them to the server. If Access=Write, R=$RE(S(x_r))$ and N=$Enc((v, T_{exp}), k, g, h, G, 0)$ is the element to be written. If Access=Read, $R = RE(S(x_r))$ and N=$Enc((null, null), k, g, h, G, 1)$ is a fresh fake (see Section 5). The zero knowledge proof then proceeds exactly as ZK-POR.

The following result shows that reads and writes performed using Access are indistinguishable. The proof can be found in [32].

Theorem 3: The server cannot decide whether an Access operation is a Read or a Write with probability significantly larger than 1/2.

9 Shuffling the W-ORAM

When the $l-1$th level of W-ORAM stores more than 4^{l-1} elements, due to element insertions occurring during Read operation, the level needs to be spilled over into level l. Let $T[l]$ denote the list of elements stored in the W-ORAM at the l-th level. The l-th level then needs to be filled with fakes. The fakes are needed to ensure that subsequent Read accesses will not run out of fakes (see [22] for more details). The l-th level then needs to be obliviously permuted, using only $O(\sqrt{N}\log N)$ client space. Let $T^{new}[l]$ denote the re-shuffled l-th level elements. Due to the WORM semantics, the client also needs to prove that the reshuffle is correct: (i) $T^{new}[l]$ is a re-encryption of the old $T[l]$ and (ii) $|T^{new}[l]| - |T[l]| - |T[l-1]|$ elements from $T^{new}[l]$ are fakes. Shuffle performs this operation.

Shuffle(W-ORAM,l): Uses as input the W-ORAM and the index of a level to reshuffle the corresponding level (see Algorithm 4). First, spill the content of level $l-1$ into level l (lines 3-6) and compute an oblivious permutation of the new level l. Then, build its ZK proof

Algorithm 2 W-ORAM: Write value v expiring at T_{exp}.

```
1.Write(W − ORAM : ORAM, E − ORAM : ORAM,
        v : string, l : id, T_exp : int)
2.  label := newLabel(l, lkey);
3.  (A_u, B_u) := Enc(label, v, T_exp, params);
4.  ZKP := getQNRProof(A_u, n);
5.  if (verify(ZKP, A_u) = 1) then
6.      T_0 := getLevel(W − ORAM, 1);
7.      insert(T_0, (A_u, B_u));
8.      Write(E − ORAM, T_exp, label);
9.  else
10.     return error;
11.fi
12.end
```

Algorithm 3 W-ORAM: Read $label$.

```
1.Read(W − ORAM : ORAM, label : id)
2.  (R, L) := Read_ORAM(W − ORAM, label);
3.  U := (A_u, B_u) := RE(R);
4.  Proof := ZK − POR(L, U);
5.  if (verifyQNR(A_u, n)
    & verify(Proof, L, U)) then
6.      T_0 := getLevel(W − ORAM, 1);
7.      insert(T_0, U);
8.      return Dec(R, d, k);
9.  else
10.     undo(W − ORAM);
11.     return error;
12.fi end
```

of correctness, ZK-PRS, detailed in the following (see Algorithm 4, lines 7-38 for pseudo-code).

9.1 Zero Knowledge Proof of Re-Shuffle.

Similar to ZK-POR (see Section 7.3.1), ZK-PRS consists of s rounds executed by the client and the server. During each round, a proof list P_j is built by the client (line 14 of Algorithm 4). P_j has the same number of elements as $T^{new}[l]$, $O(N)$. The client builds the list $T^{new}[l]$ and each of the s proofs P in the following steps. Initially, $T^{new}[l]$ and each proof list P_j is stored as an empty list at the server. The client generates a symmetric key k for the (G, E, D) cryptosystem.

Step 1: Element Re-Encryption: First, the client takes each element from $T[l]$ and stores a re-encrypted version in $T^{new}[l]$ and in each proof P_j (lines 7-13). That is, for each element $S_i = (A_i, B_i) \in T[l]$ (stored at the server), the client generates fresh random odd values $u_i, w_i \in \{0,1\}^{\mathbf{k}}$ (lines 9 and 12) and produces one element S'_i to be inserted in $T^{new}[l]$ (line 10) $S'_i = E_k(A_i g^{2u_i}, B_i h^{2u_i})$ and one element P to be inserted in P_j (line 13) $P = E_k(A_i g^{2w_i}, B_i h^{2w_i}, "mv", \Gamma_1[i], \Gamma_2[i])$ where $\Gamma_1[i] = -w_i$ and $\Gamma_2[i] = (u_i - w_i)$. The string "mv" denotes that this proof element corresponds to an element from $T[l]$ moved to $T^{new}[l]$.

Step 2: Fake Insertion: The client adds f fake elements (lines 14-22). For each of the f fakes to be inserted in $T^{new}[l]$, the client generates two random values $s_i, k_i \in_R \{0,1\}^{\mathbf{k}}$ (line 16), $i = 1..f$, where w_i is odd. The client then adds an element $E_k(s_i^2, k_i)$ in $T^{new}[l]$ (lines 17-18). It then generates a random value $w_i \in_R \{0,1\}^{\mathbf{k}}$ (line 20) for each proof list P_j and appends an element $E_k(s_i^2 g^{2w_i}, k_i h^{2w_i}, "add", \Gamma_1[i], \Gamma_2[i])$ to P_j (lines 21-22). $\Gamma_1[i] = s_i g^{w_i}$, $\Gamma_2[i] = (u_i - w_i)\ mod\ \phi(N)$ and the string "add" denotes that this proof element is a fake added to level l.

Note that $\Gamma_1[i]$ and $\Gamma_2[i]$ are used to keep track of the correspondence between the ith element of each P_j and its re-encryptions in $T[l]$ and $T^{new}[l]$ after the list reshuffle step (see next).

Step 3: List Reshuffle: At the end of the set generation step, the client and the server have a one-to-one correspondence between each element in $T^{new}[l]$, each element in each P_j and each element in $T[l]$. The client then calls the oblivious scramble, OS, procedure using $T^{new}[l]$ and each P_j as inputs (lines 23-25). During the OS call, elements read from $T^{new}[l]$ and P are decrypted (using k) and re-encrypted before being written back. Due to the semantic security properties of the encryption scheme employed, at the end of the OS, the server can no longer map elements from $T[l]$ to elements in the reshuffled $T^{new}[l]$ and P_j sets.

Step 4 - Decryption: The client reads each element from the reshuffled $T^{new}[l]$ list, decrypts the element and writes it back in-place (lines 26-28). The client reads each element from each proof list P_j, decrypts it and writes back $(A_i g^{2w_i}, B_i h^{2w_i}, E_k(str, \Gamma_1[i], \Gamma_2[i]))$, where str is either "mv" or "add" (lines 29-32).

Step 5 - Proof Verification: The server verifies each proof list P_j (lines 34-37). If any verification fails, restore the W-ORAM to the state at the beginning of the operation and return error (lines 36-37). Each verification, for a proof list P, works as follows.

The server flips a coin b. If $b = 0$, the server asks the client to prove that P is a valid reshuffle of $T[l]$ and all the remaining elements in P are fakes. For this, the client reads each element of P, $(A_i g^{2w_i}, B_i h^{2w_i}, E_k(str, \Gamma_1[i], \Gamma_2[i]))$, retrieves $\Gamma_1[i]$ and sends to the server, $A_i g^{2w_i}$, $B_i h^{2w_i}$, str and $\Gamma_1[i]$. If $str =$ "mv", the server first verifies that indeed $\Gamma_1[i]$ is an odd number, then verifies that $RE((A_i g^{2w_i}, B_i h^{2w_i}), \Gamma_1[i])$ appears in $T[l]$ exactly once. If $str =$ "add", the server verifies that $\Gamma_1[i]^2$ is the first field of exactly one tuple in $T^{new}[l]$. If at the end of this step the client has proved that $|T[l]|$ elements from $T^{new}[l]$ are re-encryptions of the elements from $T[l]$ and that f elements from $T^{new}[l]$ are fakes, the server continues. Otherwise it outputs "error" and stops.

If $b = 1$, the client needs to prove that P is a valid reshuffle of $T^{new}[l]$. For this, the client reads each element from P, recovers $\Gamma_2[i]$ and sends to the server

Algorithm 4 Shuffle of level l.

```
1.Shuffle(W − ORAM : ORAM, l : int)
2. T^new[l] : string[]#new level l array

#spill T[l − 1] into T[l]
3. T[l − 1] := getLevel(W − ORAM, l − 1);
4. T[l] := getLevel(W − ORAM, l);
5. T[l] := T[l − 1] ∪ T[l];
6. T[l − 1] := ∅;
#re − encrypt elements from T[l]
7. for (i := 1; i ≤ |T[l]|; i + +) do
8.     e := T[l][i];
9.     u[i] := genRandom();
10.    T^new[l][i] := E_k(RE(e, u[i]));
11.    for (j := 1; j ≤ s; j + +) do
12.        w[i] := genRandom();
13.        P_j[i] := E_k(RE(e, w[i]),
               "mv", u[i], u[i] − w[i]);
#add fakes
14.f := fakeCount(T[l]);
15.for (i := 1; i ≤ f; i + +) do
16.    (s[i], k[i]) := genRandom();
17.    e := (s[i]^2, k[i]);
18.    append(T^new[l], E_k(e));
19.    for (j := 1; j ≤ s; j + +) do
20.        w[i] := genRandom();
21.        re := RE(e, w[i]),
               "add", s[i]g^w[i], u[i] − w[i]);
22.        append(P_j[i], E_k(re));
#Shuffle T^new[l] and proofs
23.T^new[l] := OS(T^new[l]);
24.for (j := 1; j ≤ s; j + +) do
25.    P_j := OS(P_j);
#decrypt shuffled elements
26.for (i := 1; i ≤ |T^new[l]|; i + +) do
27.    e := T^new[l][i];
28.    T^new[l][i] := D_k(e);
29.    for (j := 1; j ≤ s; j + +) do
30.        e := P_j[i];
31.        (A, B, str, C, D) := D_k(e);
32.        P_j[i] := (A, B, E_k(str, C, D));
#proof verification step
34.for (j := 1; i ≤ s; i + +) do
35.    if (!verify(T[l], T^new[l], P_j)) then
36.        undo(W − ORAM, l − 1, l);
37.        return error;
#commit new level
38.T[l] := T^new[l];
```

the values $A_i g^{2w_i}$, $B_i h^{w_i}$ and $\Gamma_2[i]$. The server verifies that $RE((A_i g^{2w_i}, B_i h^{2w_i}), \Gamma_2[i])$ occurs in $T^{new}[l]$ once.

Analysis: We now present the following results, whose proofs can be found in [32].

Theorem 4: A correct execution of ZK-PRS has $O(\log N \log\log N)$ amortized complexity.

Theorem 5: ZK-PRS is a zero knowledge proof system of $Shuffle \in WORM$.

10 Element Expiration

Expire(T): Use as input a time epoch T and remove all the elements from the W-ORAM that expire in that epoch (see Algorithm 5). Use the E-ORAM to enumerate all the labels that expire at T (line 4). For each such $label$ (line 5) read (and remove) from the W-ORAM the corresponding element (line 6). Note that the $Read_{ORAM}$ operation also returns the entire list E of elements removed from the W-ORAM – containing $\log N$ elements. Then, build a zero knowledge proof of correctness, ZK-PEE (line 7). ZK-PEE proves that E contains one real element that expires at T and the rest ($\log N - 1$ elements) are fakes. If the proof verifies, the server accepts the expiration, otherwise restores the W-ORAM to the state before the Read of line 6 (line 9) and returns error (line 10). We now describe ZK-PEE.

10.0.1 Zero Knowledge Proof of Element Expiration.

ZK-PEE takes as input the element to be expired, R and the list E of all elements that were removed from W-ORAM when R was read (line 6). Note that $R \in E$. Let m be the number of elements in E and let $E = \{(s_1^2, k_1), .., R, .., (s_m^2, k_m)\}$. Let r be R's index in E. ZK-PEE consists of s rounds. During each round the following steps are executed by the client and the server.

Step 1: Proof Generation: The client generates a random permutation $\pi \in_R \mathcal{P}_m$ and a random vector $\bar{w} = \{w_1, .., w_m\}$, where $w_i \in_R \{0,1\}^{\mathbf{k}}$ are odd. The client computes the list $P = \pi(E\bar{w})$ and sends it to the server.

Step 2: Proof Verification: The server flips a bit b. If $b = 0$, the client reveals $\bar{w}$. The server verifies that all $w_i \in \bar{w}$ are odd and that $P = \pi(E\bar{w})$. If $b = 1$, the client reveals $Dec(R, d, k) = M(x) = (E_k(x), T_{exp}, "real", rnd)$ to the server along with the encryption factor uw_r and the square roots of the remaining m-1 (fake) elements in P, $s_1 g^{w_1}, .., s_m g^{w_m}$. The server verifies the revealed element: (i) its format, that is, $T_{exp} = T$ and the third field is "real" and (ii) its correctness, $(M(x)g^{2uw_r}, h^{2uw_r}) \in P$. The server also verifies that the remaining m-1 elements in P are fakes, by checking that $(s_i g^{w_i})^2$ occurs in the first field of exactly one element in P.

Analysis: Let e be the number of elements that expire simultaneously. Then, the following result holds (see proofs in [32]).

Theorem 6: A correct execution of Expire has $O(e \log N)$ complexity.

Theorem 7: ZK-PEE is a zero knowledge proof system of Expire $\in$ WORM.

11 Audit

Audit(d,k).: Take as input the decryption keys d and k to search for desired elements in W-ORAM. Call $Dec((A, B), d, k)$, for all elements (A, B) in the W-ORAM. Once all the elements are recovered, they can be searched for desired keywords.

Algorithm 5 Operation that removes all W-ORAM elements that expire at time T.

```
1.Expire(E − ORAM, W − ORAM : ORAM, T : int
2.  L : id[]; #expiring labels
3.  E : string[]; #removed from W − ORAM
4.  L := Enumerate(E − ORAM, T);
5.  for each label in L do
6.      (R, E) := Read_ORAM(W − ORAM, label);
7.      Proof := ZK − PEE(R, E);
8.      if (verify(Proof, E) = 0) then
9.          undo(W − ORAM);
10.         return error;
11.     fi od
12.end
```

12 Key Management

For the sake of presentation clarity, we have presented a simplified element encoding operation (Enc). Specifically, an element x is stored as the pair $(M(x)g^{2b}, h^{2b})$, consisting of an encrypted part and a recovery key. However, during element expiration (see Section 10) the client needs to prove to the server (in zero knowledge) the fact that one element in the list of accessed elements expires. For this, the client needs to provide the server not only with the decrypted element but also with the obfuscating exponent (b in the above example). Since an element may have been accessed and re-encrypted many times during read and reshuffle operations, the client keeps track of the changes in the obfuscation exponent.

We address this by storing a third field for any element: the encrypted exponent, e.g., $E(b)$ in the above case, where E is any semantically secure symmetric key encryption method, whose key is private to the client. Whenever the element is re-encrypted (during read and re-shuffle operations), the new exponent is stored encrypted, replacing the existing one. The use of a semantically secure encryption method prevents the server from using this third field to correlate reshuffled elements. For fake elements the third field is random and changes whenever a fake is being "re-encrypted".

13 Experimental Evaluation

We have implemented our solution using OpenSSL and we have tested it on the configuration depicted in Table 1. We used the same PC configuration (single core 2.4GHz with 256MB of RAM and 120MB/s sustained read/write rates) for both server and client platforms. As such, the server and client can perform 250 modular exponentiations per second, leading to 125 record re-encryptions per second and 272K AES encryptions on 1024 blocks. The link between client and server was a duplex 10MB/s. The outsourced dataset consists of 1024 bit records. In the following, we look at the overheads of read and shuffle as they are the most expensive operations. The element expiration operation follows the same steps as a read and thus its cost is similar.

Resource	Spec
Processor	2.4GHz
RAM	256KB
Disk bandwidth	120MB/s
Link bandwidth	10MB/s
Block size	1024b
T_{RE}	125 ops/s
T_{sym} on 1024b	272355 ops/s

TABLE 1
Client and server configurations.

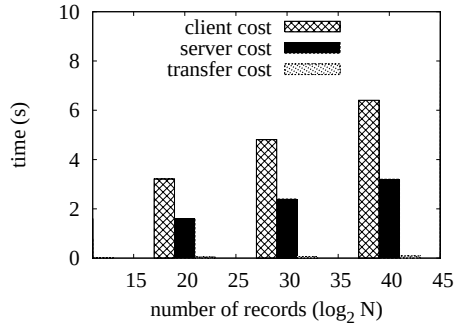

Fig. 1. ZK-POR client and server overheads. Read overhead (shown in seconds) as a function of the dataset size, on the dataset size.

Read Overheads: Figure 1 shows the overhead of the ZK-POR process as a function of the number of records, N. The number of proof sets employed is 40, for a client cheating probability of 2^{-40}. The x-axis shows the number of records in logarithmic scale. We have experimented with datasets ranging from 1Mb to 1Tb. For a 1Tb dataset (2^{30} records of size 1024), the client cost is under 7s and the server cost is under 4s. The transfer cost of the 41 sets of records including also the disk read/write times is only a fraction of a second. Thus, the total overhead of a read is around 10s. The server overhead is roughly half the overhead of the client, since the server has to verify re-encryptions only on half the sets generated by the client.

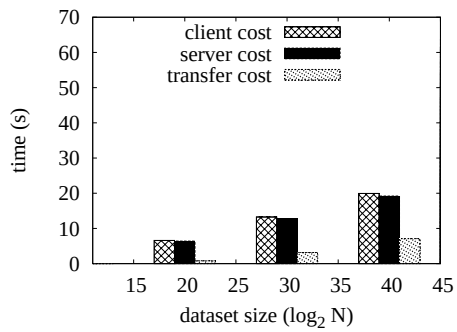

Fig. 2. Client, server and communication components of the amortized cost of ZK-PRS. Shuffle overhead (shown in seconds) as a function of the dataset size, $\log_2 N$.

Shuffle Overheads: We have measured the amortized impact of shuffles on the operation of the WORM ORAM structure and Figure 2 shows our results. The amortized cost includes the cost of re-shuffle incurred at *all* the levels in the ORAM, over all the ORAM accesses. Figure 2 shows the dependence of the cost of ZK-PRS on the number of records stored at the server, shown on the x-axis in logarithmic scale. The number of proof sets is 40. Similar to the read overheads, the client and server computation components of ZK-PRS show a logarithmic dependence on the size of the dataset. The client and server overheads are similar, up to 20 seconds for 2^{40} datasets, due to the fact that the server has to verify set re-encryptions irrespective of the outcome of the coin flip process. The communication overhead during the shuffle operation, including the time to read/write and transfer proof sets takes around 7 seconds for 1Tb datasets. This is because the client needs to shuffle not one but 41 proof sets. Thus, the total, amortized overhead of a shuffle operation is around 47s.

In ORAM, the network transfer time alone for reshuffling level i consists of about 10 sorts of $4^i logn$ data, each sort requiring $4^i log(n) log^2(4i^i logn)$ block transfers, for a total of $104^i log(n) log^2(4^i logn) 2^{10}/10MB/se$. Summing over the $\log_4 n$ levels, and amortizing each level over 4^{i1} queries, ORAM has an amortized network traffic cost per query of 3.680Gb. Over the sample 10MB/s link this is a 48 sec/query amortized transfer time. Thus, by using an improved oblivious scramble protocol, we are able to support regulatory compliance *and* maintain the cost imposed by the original ORAM.

14 Conclusions

In this paper we introduce WORM-ORAM, a solution that provides WORM compliant Oblivious RAMs. Our solution is based on a set of zero knowledge proofs that ensure that all ORAM operations are WORM compliant. The protocol features the same asymptotic computational complexity as ORAM.

15 Acknowledgments

We would like to thank Dan Boneh and Peter Williams for early comments and suggestions. We thank the reviewers for their excellent feedback. Sion is supported by the U.S. NSF and by grants from CA Technologies, Xerox/Parc, IBM and Microsoft Research.

References

[1] National Association of Insurance Commissioners. Graham-Leach-Bliley Act, 1999. www.naic.org/GLBA.
[2] U.S. Dept. of Health & Human Services. The Health Insurance Portability and Accountability Act (HIPAA), 1996. www.cms.gov/hipaa.
[3] U.S. Public Law 107-347. The E-Government Act, 2002.
[4] U.S. Public Law No. 107-204, 116 Stat. 745. Public Company Accounting Reform and Investor Protection Act, 2002.
[5] The U.S. Securities and Exchange Commission. Rule 17a-3&4, 17 CFR Part 240: Electronic Storage of Broker-Dealer Records. Online at http://edocket.access.gpo.gov/, 2003.
[6] The U.S. Department of Defense. Directive 5015.2: DOD Records Management Program. Online at http://www.dtic.mil/whs/directives/corres/pdf/50152std_061902/p50152s.pdf, 2002.
[7] The U.S. Department of Health and Human Services Food and Drug Administration. 21 CFR Part 11: Electronic Records and Signature Regulations. Online at http://www.fda.gov/ora/compliance_ref/part11/FRs/background/pt11finr.pdf, 1997.
[8] The U.S. Department of Education. 20 U.S.C. 1232g; 34 CFR Part 99:Family Educational Rights and Privacy Act (FERPA). Online at http://www.ed.gov/policy/gen/guid/fpco/ferpa, 1974.
[9] The Enterprise Storage Group. Compliance: The effect on information management and the storage industry. Online at http://www.enterprisestoragegroup.com/, 2003.
[10] Enron email dataset. http://www.cs.cmu.edu/enron/.
[11] IBM Corp. IBM TotalStorage Enterprise. Online at http://www-03.ibm.com/servers/storage/, 2007.
[12] HP. WORM Data Protection Solutions. Online at http://h18006.www1.hp.com/products/storageworks/wormdps/index.html, 2007.
[13] EMC. Centera Compliance Edition Plus. Online at http://www.emc.com/centera/ and http://www.mosaictech.com/pdf_docs/emc/centera.pdf, 2007.
[14] Hitachi Data Systems. The Message Archive for Compliance Solution, Data Retention Software Utility. Online at http://www.hds.com/solutions/data_life_cycle_archiving/achievingregcompliance.html, 2007.
[15] Zantaz Inc. The ZANTAZ Digital Safe Product Family. Online at http://www.zantaz.com/, 2007.
[16] StorageTek Inc. VolSafe secure tape-based write once read many (WORM) storage solution. Online at http://www.storagetek.com/, 2007.
[17] Sun Microsystems. Sun StorageTek Compliance Archiving system and the Vignette Enterprise Content Management Suite (White Paper). Online at http://www.sun.com/storagetek/white-papers/Healthcare_Sun_NAS_Vignette_EHR_080806_Final.pdf, 2007.
[18] Sun Microsystems. Sun StorageTek Compliance Archiving Software. Online at http://www.sun.com/storagetek/management_software/data_protection/compliance_archiving/, 2007.
[19] Network Appliance Inc. SnapLock Compliance and SnapLock Enterprise Software. Online at http://www.netapp.com/products/software/snaplock.html, 2007.
[20] Quantum Inc. DLTSage Write Once Read Many Solution. Online at http://www.quantum.com/Products/TapeDrives/DLT/SDLT600/DLTIce/Index.aspx and http://www.quantum.com/pdf/DS00232.pdf, 2007.
[21] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAM. *Journal of the ACM*, 45:431–473, May 1996.
[22] Peter Williams, Radu Sion, and Bogdan Carbunar. Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In *ACM Conference on Computer and Communication Security CCS*, 2008.
[23] Bogdan Carbunar and Radu Sion. Regulatory Compliant Oblivious RAM. In *ACNS*, pages 456–474, 2010.
[24] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In *Advances in Cryptology - CRYPTO 2010*, pages 502–519, 2010.
[25] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *J. Algorithms*, 51:122–144, May 2004.
[26] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 41–50, 1995.
[27] W. Gasarch. A WebPage on Private Information Retrieval. Online at http://www.cs.umd.edu/~gasarch/pir/pir.html.
[28] W. Gasarch. A Survey on Private Information Retrieval. Online at http://citeseer.ifi.unizh.ch/gasarch04survey.html.
[29] Radu Sion and Bogdan Carbunar. 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.
[30] O. Goldreich. *Foundations of Cryptography I*. Cambridge University Press, 2001.
[31] Dan Boneh. The Decision Diffie-Hellman Problem. In *ANTS-III: Proceedings of the Third International Symposium on Algorithmic Number Theory*, 1998.

[32] Bogdan Carbunar and Radu Sion. Write Once Read Many Oblivious RAM. http://www.cs.stonybrook.edu/~sion/research/, 2011.

Bogdan Carbunar is a principal staff researcher in the pervasive platforms and architectures lab of the Applied Research Center at Motorola. His research interests include distributed systems, security and applied cryptography. He has been on the program committee of conferences such as the ISOC Network and Distributed Systems Security Symposium (NDSS), the IEEE International Conference on Multimedia and Expo (ICME) and Financial Cryptography (FC). He holds a Ph.D. in Computer Science from Purdue University.

Radu Sion heads the Stony Brook Network Security and Applied Cryptography (NSAC) Lab. His research interests include Information Assurance and Efficient Computing. He builds systems mainly, but enjoys elegance and foundations, especially if of the very rare practical variety. Sponsors and collaborators include IBM, IBM Research, NOKIA, Xerox, as well as the National Science Foundation which awarded also the CAREER Award. Radu is on the steering board and organizing committees of conferences such as NDSS, Oakland S&P, CCS, USENIX Security, SIGMOD, ICDE, FC and others.