Regulatory Compliant Oblivious RAM

Bogdan Carbunar[1] and Radu Sion[2]

[1] Motorola Labs
carbunar@motorola.com
[2] Stony Brook Network Security and Applied Cryptography Lab
sion@cs.stonybrook.edu

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 full 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 existing records are overwritten and WORM semantics are preserved.

Specifically, clients can add encrypted data records to the database ("the ORAM") hosted by a service provider. Each record will be associated with a regulatory mandated expiration time. Once stored, the client can read all data obliviously, (and add new records) – only leaking that access took place and nothing else. No access patterns or data records or any other information is leaked. The server, without having plaintext access to the data or the client access patterns then ensures – in a client-server interaction – that any client access is WORM compliant: it is either a read of an existing record, or an addition of a new record (with a new index – no overwriting permitted).

To achieve the above, at an overview level, the solution outlines as follows. The server hosts two ORAMs, one storing the actual data items (the W-ORAM)

and one allowing the private retrieval of items expiring at any given time (the E-ORAM). The E-ORAM is effectively a helper data structure allowing the client to determine which items to expire at given time intervals. Client access to the E-ORAM needs to be private, but does not need to be proved correct.

The server exports an access API to the W-ORAM to the client consisting of four types of operations: write, read, expire and compliance verification. For access pattern privacy purposes as in the traditional ORAM protocols, the data set stored at the server contains both "real" and "fake" items – this is discussed later. Then, during any legitimate access to the W-ORAM the client will prove in ZK to the server that the item written is real, "well formed" and can be decrypted later in the case of an audit. Moreover she also proves that the access does not in fact overwrite any existing database item. Similarly, in the expiration operation, the client proves in ZK that the element to be removed from the W-ORAM has indeed expired. Finally, at audit time, the data has to be accessible to an authorized auditor, even in the case of a non-cooperating client (e.g., that could refuse to reveal encryption keys).

We show that our solution does not change the computational complexity of existing ORAM implementations. However, we warn that the constants involved are non-negligible and render this result of theoretical interest only for now. Future work focuses on reducing these overheads towards true practical efficiency.

2 Related Work

2.1 Oblivious RAM

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 from 0 to $\log n$ blocks.

ORAM Reads. 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.

ORAM Writes. 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. Inserts are performed identically to writes, since no old value will be discovered in the query phase. Note that semantic security properties of the re-encryption function ensure the server is unable to distinguish between reads, writes, and inserts, since the access patterns are indistinguishable.

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. Recall that since every block is encrypted with a semantically secure encryption function, the adversary cannot distinguish between fake and real blocks.

Oblivious Scramble. In [22] Williams et al.introduced an algorithm that performs an oblivious scramble on a array of size n, with $c\sqrt{n}$ local storage, in $O(n \log \log n)$ time with high probability. Informally, the algorithm is a merge sort, except a random number generator is used in place of a comparison, and multiple sub-arrays are merged simultaneously. The array is recursively divided into segments, which are then scrambled together in groups. The time complexity of the algorithm is better than merge sort since multiple segments are merged together simultaneously. Randomly selecting from the remaining arrays avoids comparisons among the leading items in each array, so it is not a comparison sort.

2.2 Oblivious Transfer with Access Control

Camenish et al. [23] 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..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. [24] 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.

Note that the above oblivious transfer flavors do not consider by definition the problem of obliviously enforcing WORM semantics as well as writing to the data. Our regulatory compliant problem is complicated by the fact that we also allow clients to add to the database while proving that operations performed on the data do not overwrite old records. One can trivially extend OT with an add call, by imposing $O(N)$ communication and computation overheads. However, by building our solution on ORAM we can perform both read and add operations with only poly-logarithmic complexity and traffic overheads.

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. Compliant record expiration of inaccessible records should be easily proved to auditors.

Additionally, when records and their associated access patterns are sensitive they need to be concealed from a curious server. The main purpose of WORM-ORAM is to enable WORM semantics while preserving data confidentiality and access pattern privacy. This inability of the server to "see" data and associated access patterns prevents the deployment of conventional file/storage system access control mechanisms or data outsourcing techniques. Thus, we have the following additional requirements:

- Data records are encrypted from the server (*confidentiality*).
- The server cannot distinguish between different read operations targeting the same or different data records (*access privacy*).
- During a read, in the ORAM protocol, to enforce *WORM semantics*, clients will need to prove to the server that any access did not remove data records. Specifically, when re-inserting one of the read elements back into the root of the ORAM pyramid, the client needs to prove to the server in ZK that the inserted element is a correct re-encryption of the previously removed "real" element (see Section 2.1 for details).
- During a re-shuffle, in the ORAM protocol, clients can prove that no "real" elements were converted into "fake" ones.

Additionally, in the WORM-ORAM scenario, we assume the following:

- The server is allowed to distinguish between record expiration, read and write operations.
- Clients 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.

Several participants are of concern. First, clients have incentives to *rewrite history* and alter or completely remove *previously written* records. We note that in the regulatory scenario, there exists an apparent imbalance – clients are assume to correctly store records at the time of their creation – only later does regretting the past becoming a concern. Thus the main focus of WORM assurances is not to prevent history but rather just its rewriting. In reality, the "regret" time interval between the creation/storage and regretting of a record is non-zero, application-specific, and often quite large. To remove any application dependency, here we consider the strongest WORM guarantees, in which records are not to be altered as soon as they are written.

Second, the storage provider (server) is *curious* and has incentives to illicitly gain information about the stored data and access patterns thereto. As the regulatory storage provider, the server is the main enforcer of WORM semantics and record expiration. Naturally, the server is assumed to not collude with clients illicitly desiring to alter their data. In summary, the server is trusted to run protocols correctly yet it may try to use information obtained from correct runs to obtain undesired information. This assumption is natural and practical as otherwise one can easily imagine a server simply deleting stored records in a

denial of service attack. Basic denial of service on the client or server side is not of interest here.

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 [25] 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.

The Goldwasser, Micali and Rackoff [26] *zero knowledge proof of quadratic non-residuosity* proceeds roughly as follows. Given two parties A and B, A claims knowledge of $QR(v, n) = 0$, for $(v|n) = 1$. A proves this in zero knowledge to B, that is, without revealing n's factorization. To achieve this, B selects m random values $r_1, .., r_m$ and flips m coins. For each coin c_i, if $c_i = 0$ B computes $x_i = r_i^2 \; mod \; n$ to A, otherwise it computes $x_i = vr_i^2 \; mod \; n$. B sends all computed x_i values to A. A needs to send back the square roots of the quadratic residues it detects in the list $x_1, ..x_m$. If $QR(v, n) = 0$, then A correctly detects the residues. If $QR(v, n) = 1$, all the values received by A will be quadratic residues. A can then cheat only with probability $1/2^m$.

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 [27]. 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. We now provide a brief overview of each operation described above and follow with a detailed description in the next section.

Gen. Operation executed initially, to generate system parameters for each participant: client, server, auditor.

Enc, Dec, RE. *Enc* and *Dec* provide the basic encryption and decryption operations for elements to be stored in the W-ORAM. *RE* is the W-ORAM element re-encryption operation. *RE* is needed to ensure that the server cannot distinguish the same W-ORAM element accessed multiple times, while allowing the server to prove in zero knowledge the element's correctness.

Write. Operation used by the client to store an element on the server. The client needs to label the element and determine its expiration time. The client stores the element indexed by the label on the W-ORAM and the label indexed by the element's expiration time in the E-ORAM.

Read. Allows the client to retrieve from the W-ORAM an element indexed by an input label. The operation is based on existing ORAM reading techniques. In addition, it obliviously ensures that the client cannot remove or alter any real element from the W-ORAM.

Shuffle. Re-shuffles a level (provided as input) in the W-ORAM. Based on existing ORAM shuffling techniques, it needs to ensure that the client cannot remove or alter existing W-ORAM elements.

Expire. This operation makes use of both the E-ORAM and W-ORAM to remove all elements from the W-ORAM whose expiration time equals an input expiration time epoch. The operation needs to obliviously convince the server that only expiring elements are removed and no other W-ORAM elements are altered.

Audit. Enables an auditor to access the entire W-ORAM and search for keywords of interest.

5 Solution

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 [28]. 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). 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 $(s^2\ mod\ n, k)$.

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 $(A', B') = (Mg^{2u}, h^{2u})$. Note that u may also be used as an input parameter by $RE((A, B), u)$.

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.

We now prove the semantic security of *Enc*.

Theorem 1 *Enc is IND-CPA secure.*

Proof. Let Q be an adversary that can break the semantic security of *Enc* with advantage ϵ. We then build an adversary Q^* that can break the DDH assumption in G without knowing n's factors, with probability ϵ. Let CH be a challenger. CH interacts with Q^* by sending the triple $(A = g^a, B = g^b, C = g^c)$. Q^* needs to decide whether $c = ab$ or is randomly distributed.

Q^* sends A to Q as the public key (h in our protocol). A then sends to Q^* two messages M_0 and M_1. Q^* picks a bit $\alpha \in_R \{0,1\}$ randomly and sends back to Q the tuple $(M_\alpha B^2, C^2)$. Q sends back its guess for α. If Q guesses correctly, Q^* sends to CH the value 1 ($c = ab$) or 0 (c is random).

When $c = ab$, the tuple $(M_\alpha B^2, C^2)$ is a correct ciphertext of M_α. Then, the interaction between Q^* and Q is correct and the probability of Q^* to output 1 is $1/2 + \epsilon$. When c is distributed randomly, the tuple $(M_\alpha B^2, C^2)$ is independent of α. The probability of Q^* outputting 1 is then $1/2$. Thus, Q^* has advantage ϵ in the DDH game.

Note that we can similarly prove that *RE* is IND-CPA. That is, 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 $1/2$. In the following, we describe first the main E-ORAM operations and then the W-ORAM accessing operations.

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

Algorithm 1 E-ORAM: Write a new label under an expiration time and enumerate all labels indexed under an expiration time.

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

As mentioned in Section 2, $Read_{ORAM}$ denotes the standard ORAM read operation, taking as input the ORAM and a label and returns 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. Let *ObliviousScramble* 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. We now present the most important operations for accessing the E-ORAM, Write and Enumerate.

Write(E-ORAM,T_{exp},label). The pseudo-code of this operation is shown in Algorithm 1, lines 1-11. It allows the client to record the fact that *label* expires at time T_{exp}. It first reads the element currently stored under T_{exp} (line 2). If no such element exists (line 3), it generates an element encoding the fact that this label is the first to be stored under T_{exp} (line 4) and writes it on the E-ORAM (line 5). If however a label is already stored under T_{exp} (line 6), retrieve that label l along with the counter c that specifies how many labels are currently expiring (stored in E-ORAM) at T_{exp} (line 7). Note that the read operation performed on line 2 removes this element from the E-ORAM. Then, since now $c+1$ labels expire at T_{exp}, store label l and the incremented counter in the E-ORAM under T_{exp} (line 8). 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).

Enumerate(E-ORAM,T_{exp}). This operation enables C to retrieve all the labels in E-ORAM that expire at T_{exp}. Its pseudo-code is shown in 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).

5.2 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 $F_{lkey}(label)$.

In the following we describe the main W-ORAM accessing operations.

5.3 Writing on the Server

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, B) := Enc(label, v, T_exp, params);
4.  ZKP := getQNRProof(A, n);
5.  if (verify(ZKP, A) = 1) then
6.      T_0 := getLevel(W − ORAM, 1);
7.      insert(T_0, (A, B));
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
```

Write((W-ORAM,E-ORAM,v,l,T_{exp},params). Insert a value v under a label l, with expiration time T_{exp} on the server (on the W-ORAM and E-ORAM). It takes as input also C's view of Gen's output, $params = k, g, h, G$. Algorithm 2 shows the pseudo-code of this operation. It first generates a new *label* (line 2) and calls *Enc* to produce a W-ORAM tuple (A, B) (line 3). It then generates a non-interactive zero knowledge proof of $QR(A, n) = 0$ (A's quadratic non-residuosity). If the proof verifies (line 5) the server inserts the tuple (A, B) 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 5.1).

5.4 Reading from the Server

Read(W-ORAM,label). Read takes as input the W-ORAM and a *label* and returns an element of format $(label, x, T_{exp})$. Algorithm 3 shows the pseudo-code for this operation. Read first performs on W-ORAM a standard ORAM read on the desired *label* (line 2). This returns both the W-ORAM element R of interest and the list L of elements (containing R) removed from the W-ORAM. C 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 5.4. S 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), S inserts U in the first level of the W-ORAM (lines 6-7). C decrypts the desired element R and returns the result (line 8). If any proof fails (line 9) S restores the W-ORAM to the state before the start of Read and returns error (lines 10-11).

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. ZK-POR is executed by the client C and the server S. Let m denote the number of levels in the W-ORAM, $m = \log N$.

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. C 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 by C and S.

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

Step 2: Proof Validation. S flips a coin b. If b is 0, C reveals $w_1, .., w_m$. S 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, C sends to S 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, C can easily recover s_i. S 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. S then verifies that $(A_r g^{2w_r}, B_r h^{2w_r}) = RE((A_u, B_u), \Gamma)$. If any verification fails, S outputs "error" and stops.

Theorem 2 *A correct execution of Read from W-ORAM has* $O(\log N)$ *complexity.*

Theorem 3 *ZK-POR is a zero knowledge proof system of Read ∈ WORM. That is, Read is WORM compliant.*

Due to lack of space, the proofs will be included in the journal version of the paper.

Note that the soundness property of ZK-POR ensures that a cheating client can remove an element from the ORAM during the Read operation without being detected with probability at most $1/2^s$.

5.5 Shuffling the W-ORAM

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] := ObliviousScramble(T^new[l]);
24. for (j := 1; j ≤ s; j + +) do
25.     P_j := ObliviousScramble(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];
```

When the $l-1$th level of W-ORAM stores more than 4^{l-1}, 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 [27] 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). This operation takes as input the W-ORAM and the index of its level that needs to be reshuffled. Algorithm 4 shows the pseudo-code of this operation. It first spills the content of level $l-1$ into level l (lines 3-6).

Then, it needs to compute the oblivious permutation and build its ZK proof of correctness. We call this procedure ZK-PRS and describe it in detail in the following.

Zero Knowledge Proof of Re-Shuffle. The pseudo-code of ZK-PRS is in Algorithm 4, lines 7-38. Similar to ZK-POR (see Section 5.4), ZK-PRS consists of s rounds executed by the client C and the server S. During each round, a proof list P_j is built by C (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 S. The client C generates a symmetric key k for the (G, E, D) cryptosystem.

Step 1: Element Re-Encryption. First, C 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 S), C 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. C adds f fake elements (lines 14-22). For each of the f fakes to be inserted in $T^{new}[l]$, C generates two random values $s_i, k_i \in_R \{0,1\}^{\mathbf{k}}$ (line 16), $i = 1..f$, where w_i is odd. C 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, C (and S) have a one-to-one correspondence between each element in $T^{new}[l]$, each element in each P_j and each element in $T[l]$. C then calls the *ObliviousScramble* procedure using $T^{new}[l]$ and each P_j as inputs (lines 23-25). During the ObliviousScramble 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 ObliviousScramble, S can no longer map elements from $T[l]$ to elements in the reshuffled $T^{new}[l]$ and P_j sets.

Step 4 - Decryption. C reads each element from the reshuffled $T^{new}[l]$ list, decrypts the element and writes it back in-place (lines 26-28). C 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" – moved or added fake (lines 29-32).

Step 5 - Proof Verification. S 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.

S flips a coin b. If $b = 0$, S asks C to prove that P is a valid reshuffle of $T[l]$ and all the remaining elements in P are fakes. For this, C 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 S, $A_i g^{2w_i}$, $B_i h^{2w_i}$, str and $\Gamma_1[i]$. If $str =$ "mv", S 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", S verifies that $\Gamma_1[i]^2$ is the first field of exactly one tuple in $T^{new}[l]$. If $b = 1$, C needs to prove that P is a valid reshuffle of $T^{new}[l]$. For this, C reads each element from P, recovers $\Gamma_2[i]$ and sends to S the values $A_i g^{2w_i}$, $B_i h^{w_i}$ and $\Gamma_2[i]$. S verifies that $RE((A_i g^{2w_i}, B_i h^{2w_i}), \Gamma_2[i])$ occurs in $T^{new}[l]$ exactly once.

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
```

The following result holds. Due to lack of space we omit the proof, which will be included in the journal version of the paper.

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

Due to lack of space we omit the proof, which will be included in the journal version of the paper. The journal version also includes the proof that ZK-PRS is a zero knowledge proof system of Shuffle $\in$ WORM.

5.6 Element Expiration

Expire(T). The operation takes as input a time epoch T and removes all the elements from the W-ORAM that expire in that epoch. The pseudo-code of the operation is shown in Algorithm 5. It first uses the E-ORAM to enumerate all the labels that expire at T (line 4). Then, for each such *label* (line 5) it reads (and removes) 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. It then builds a zero knowledge proof of the fact that E contains one real element that expires at T and the rest ($\log N - 1$ elements) are fakes (line 7). 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).

Zero Knowledge Proof of Element Expiration. We present a sketch of ZK-PEE, which is called in Algorithm 5, line 7. ZK-PEE takes as input the element to be expired, R and the list of all elements that were removed from W-ORAM when R was read (line 6). Let $m = \log N$ be the number of elements in E. ZK-PEE consists of s rounds run between C and S. During each round C generates $\bar{w} = \{w_1, .., w_m\}$, where $w_i \in_R \{0,1\}^{\mathbf{k}}$ are odd. C computes a proof

list $P = \pi(E\bar{w})$, where $\pi \in_R \mathcal{P}_m$ is a random permutation. S then flips a bit b. If $b = 0$, C needs to provide $\bar{w}$. S verifies that all $w_i \in \bar{w}$ are odd and that $P = \pi(E\bar{w})$.

If $b = 1$, C reveals $Dec(R, d, k) = M(x) = (E_k(x), T_{exp}, "real", r)$ to S along with the encryption factor u and the square roots of all the other $\log N - 1$ elements in P. S verifies the revealed element: (i) its correctness, $(M(x)g^{2u}, h^{2u}) = R$ and (ii) its format, that is, $T_{exp} = T$ and the third field is "real". It also verifies that the remaining $(\log N - 1)$ elements of P are fakes.

5.7 Audit

Audit(d,k). The basic auditing operation takes as input the decryption keys d and k. It calls $Dec((A, B), d, k)$, for all elements (A, B) in the ORAM. Once all the elements are recovered, they can be searched for desired keywords.

6 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.

7 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. National Science Foundation through awards CCF 0937833, CNS 0845192, CNS 0708025, IIS 0803197, and CNS 0716608, as well as by grants from CA, Xerox, 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.p%df`, 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.pd%f`, 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/achievingregcomp%liance.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/compl%iance_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.asp%x 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. Jan Camenisch, Gregory Neven, and Abhi Shelat. Simulatable adaptive oblivious transfer. In *EUROCRYPT '07: Proceedings of the 26th annual international conference on Advances in Cryptology*, 2007.
24. S. Coull, M. Green, and S. Hohenberger. Controlling access to an oblivious database using stateful anonymous credentials. In *International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2009.
25. O. Goldreich. *Foundations of Cryptography I*. Cambridge University Press, 2001.
26. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1), 1989.
27. Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, 2008.
28. Dan Boneh. The decision diffie-hellman problem. In *ANTS-III: Proceedings of the Third International Symposium on Algorithmic Number Theory*, 1998.