Simple and Efficient Two-Server ORAM

S. Dov Gordon[1], Jonathan Katz[2], and Xiao Wang[2]

[1] George Mason University, Fairfax, USA
gordon@gmu.edu
[2] University of Maryland, College Park, USA
{jkatz,wangxiao}@cs.umd.edu

Abstract. We show a protocol for two-server oblivious RAM (ORAM) that is simpler and more efficient than the best prior work. Our construction combines any tree-based ORAM with an extension of a two-server private information retrieval scheme by Boyle et al., and is able to avoid recursion and thus use only one round of interaction. In addition, our scheme has a very cheap initialization phase, making it well suited for RAM-based secure computation. Although our scheme requires the servers to perform a linear scan over the entire data, the cryptographic computation involved consists only of block-cipher evaluations.
A practical instantiation of our protocol has excellent concrete parameters: for storing an N-element array of arbitrary size data blocks with statistical security parameter λ, the servers each store $4N$ encrypted blocks, the client stores $\lambda + 2\log N$ blocks, and the total communication per logical access is roughly $10 \log N$ encrypted blocks.

1 Introduction

Protocols for *oblivious RAM* (ORAM) allow a client to outsource storage of an array to a server, and then read from/write to that array without revealing to the server anything about the data itself or the addresses of the data blocks being accessed (i.e., the client's *memory-access pattern*). Since the introduction of the problem by Goldreich and Ostrovsky [**?**], it has received a significant amount of attention [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. The main parameters of interest are the storage at the client and server, as well as the number of communication rounds and the total client-server bandwidth needed to read or write one logical position of the array. In classical work on ORAM, the server was only required to physically read and write elements of some (encrypted) data array; more recent work [**?**,**?**,**?**,**?**,**?**] has considered solutions in which the server performs non-trivial computation as well. In that case, solutions relying on non-cryptographic computation, or symmetric-key cryptography alone, are preferable.

Lu and Ostrovsky [**?**] proposed exploring ORAM in a model where there are *two* non-colluding servers storing data on behalf of the client; the client interacts with the servers to read and write data, but the servers do not need to interact with (or even know about) each other. The solution by Lu and Ostrovsky achieves parameters that are asymptotically better than those realized by any

single-server solution: for accessing an N-element array of B-bit data blocks, the client in their protocol has storage independent of N and B, the servers each store $O(N)$ encrypted data blocks, and reading/writing has an amortized communication complexity of $O(\log N)$ encrypted data blocks. On the other hand, like most ORAM constructions with sublinear communication (with a few exceptions discussed below), the Lu-Ostrovsky protocol requires $O(\log N)$ rounds of interaction between the client and servers per logical memory access; since it is based on a hierarchical approach [**?**] and requires periodic reshuffling, their scheme is also relatively complex and does not offer good worst-case performance guarantees. A recent two-server ORAM scheme by Abraham et al. [**?**] improves the communication overhead to $O(\log N/\log\log N)$ when $B = \Omega(\lambda \log^2 N)$, but still requires $O(\log N)$ rounds.

1.1 Summary of Our Results

We show here a construction of a two-server ORAM protocol that improves on prior work both concretely and theoretically. Our scheme is also very simple to describe and implement, which we view as an added advantage especially when applying ORAM to RAM-based secure computation.

Concretely, our scheme is extremely efficient. In one instantiation of our scheme, the client stores $\lambda + 2\log N$ data blocks (where λ is a statistical security parameter), the servers each store $4N$ encrypted data blocks, and the total communication per logical read/write is only roughly $10 \log N$ encrypted blocks. This can be compared to the Lu-Ostrovsky scheme, which is estimated by the authors to have server storage $2N + O(\log^9 N)$ and an amortized bandwidth of more than $160 \log N$ encrypted data blocks per logical memory access. (Abraham et al. do not offer concrete estimates of the performance of their scheme, but we believe our protocol will have better communication overhead for practical parameters, especially for moderate B.) A drawback of our protocol is that it requires the servers to perform a linear scan of the entire data, and perform a linear number of symmetric-key operations.

In a theoretical sense, we improve upon prior work in several respects. Most importantly, our protocol requires only one round of communication per logical access; note that achieving logarithmic communication overhead with one round of interaction is a major open question for single-server ORAM.[1] Second, our communication bound holds *in the worst case*, in contrast to the Lu-Ostrovsky scheme for which it holds only in an amortized sense. Finally, in contrast to the scheme of Abraham et al., our protocol has good communication overhead regardless of the block size.

Applications to secure computation. Classical work on generic secure computation views the function being computed as a boolean or arithmetic circuit.

[1] Known single-server ORAM schemes with sublinear worst-case communication and one round of interaction [**?**,**?**] have communication complexity at least $\kappa B \log^2 N$ and would be prohibitively inefficient to implement. Other one-round schemes [**?**,**?**] have sublinear communication only in an amortized sense.

More recently, researchers have explored secure-computation protocols that work directly in the RAM model of computation [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. A basic idea in these works is to leverage ORAM to ensure that the parties' accesses to (shared) memory are oblivious. These works all assume either that the shared memory is initially empty, or that initialization of the ORAM data structure is done during some trusted preprocessing phase, because initializing a non-empty ORAM as part of the protocol would be infeasible. For our ORAM protocol, initialization is essentially "for free" and can be done locally by the servers without any interaction with the client. (To the best of our knowledge, this is not true for any prior ORAM scheme with sublinear communication overhead.) This makes our protocol extremely well-suited for applications to RAM-based secure computation in both the two-party and multi-party settings.

Our scheme has the added advantage that reads from a public address can be done very efficiently, with communication of only $2 \log N$ encrypted blocks and negligible computation. This property is also very useful in applications to secure computation.

1.2 Overview of Our Construction

Our construction can be viewed as combining any tree-based ORAM protocol [**?**,**?**,**?**,**?**] with a two-server private information retrieval (PIR) scheme [**?**]. (Combining ORAM and PIR was suggested previously by Mayberry et al. [**?**] in the single-sever setting, and Abraham et al. [**?**] in the two-server setting.) We describe each of these primitives informally, and then provide an overview of our construction. Section **??** contains formal definitions; a detailed description of our protocol is given in Section **??**.

Tree-based ORAM. At a somewhat informal level, which will be sufficient to understand the main ideas of our construction, a tree-based ORAM scheme—in the single-server setting—works in the following way.[2] Let D denote the client's data array with $D[i]$, for $0 \leq i < N$, denoting the data block stored at address i of the array. The client maintains a function $\mathsf{position}$ (called a *position map*) that maps logical memory addresses to leaves in a binary tree of depth $L = O(\log N)$ stored by the server, where each node in the tree can store some bounded number of data blocks. It will be convenient for us to assume that every node in the tree stores the same (constant) number of data blocks, with the exception of the root that can store more items. Instead of being stored on the server, the root is stored on the client and is also called a *stash*.

At any point in time, the value $D[i]$ is stored at some node on the path from the root of the tree to the leaf at $\mathsf{position}(i)$ (we call this the *path to* $\mathsf{position}(i)$). The client performs a logical read of address i by reading the entire path to $\mathsf{position}(i)$ and taking the value of $D[i]$ that is found closest to the root; a logical write to address i is done by storing the new value of $D[i]$ in the stash (replacing any old value of $D[i]$ found there).

[2] For simplicity, we ignore encryption of the data blocks in the description that follows.

Executions of an *eviction procedure* are interspersed with logical reads and writes. At a high level, during this procedure the client chooses a path $\mathcal{P}$ in the tree and then, for each data block $D[i]$ stored at some node in that path, pushes that block as far down in $\mathcal{P}$ as possible subject to the constraint that $D[i]$ must lie on the path to $\mathsf{position}(i)$. The updated values of the nodes on path $\mathcal{P}$ are then rewritten to the server. The purpose of the eviction procedure is to prevent nodes in the tree from overflowing.

Note that to ensure obliviousness, the position map must be random (so the server cannot correlate a particular path being read by the client with a logical address) and $\mathsf{position}(i)$ must be updated each time $D[i]$ is read (so the server cannot tell when the same logical address is accessed repeatedly). Since the position map itself has size $\Theta(N)$, the client must store the position map on the server in order to achieve client storage $o(N)$. The position map can be stored recursively using a tree-based ORAM; note, however, that this induces several rounds of interaction between the client and server for each logical memory access, and also increases the server-side storage.

Private information retrieval. Abstractly, a private information retrieval (PIR) scheme provides a way for a client to obliviously read a data block from an N-element array of B-bit items stored on a server using $o(BN)$ communication. For our purposes, the main distinction between PIR and ORAM is that PIR supports reads only. Historically, PIR schemes have also involved only one round of interaction.

PIR was first considered in the multi-server setting [**?**], where information-theoretic security is possible. Although PIR with computational security is possible in the single-server setting [**?**,**?**,**?**], constructions of (computationally secure) PIR in the two-server setting have much better computational efficiency. In particular, a recent construction of two-server PIR by Boyle et al. [**?**,**?**,**?**] requires only symmetric-key operations by both the client and the server, uses only one round, and has communication complexity $2B + O(\kappa \cdot \log N)$ for κ a computational security parameter. (In fact, they show that the communication can be reduced asymptotically to $2B + O(\kappa \cdot \log(N/\kappa))$ but for practical parameters this does not seem to yield a concrete improvement.)

Our construction. We show how to combine tree-based ORAM with PIR to obtain an efficient and conceptually simple protocol in the two-server setting.

In existing tree-based ORAM schemes the eviction procedure is already oblivious, as it involves either choosing a random eviction path [**?**] or choosing eviction paths according to a deterministic schedule [**?**,**?**]. Thus, only reads need to be made oblivious. As noted earlier, in prior work this is achieved using a random position map that is updated after each read. Our first conceptual insight is that we can instead have the client use (two-server) PIR to read the path associated with a particular data block. As a consequence, we can avoid ever having to update the position map (see below for why we need a position map at all) and so can use a pseudorandom position map, thereby avoiding recursion and allowing us to obtain a one-round protocol.

Obliviously reading a path in a tree of depth L can always be done using L parallel executions of a generic PIR protocol. Our second observation is that we can do better than this by adapting the specific (two-server) PIR scheme of Boyle et al. so as to natively support oblivious reading of a *path* in a tree with less than L times the communication. Details are given in Section **??**.

Since a position map is no longer needed for obliviousness, it is tempting to think that we can avoid the position map altogether. Unfortunately this is not the case, as we still need a (pseudo)random mapping of addresses to leaves in order to ensure *correctness*—specifically, so that the probability of an overflow remains negligible. In our case, however, we show that it is sufficient to choose a random position map *once*, at the outset of the protocol, and then leave it fixed for the remainder of the execution. This also means that we can generate the pseudorandom position map based on a short key chosen at the beginning of the protocol. Finally, we observe that this allows for extremely efficient initialization (in settings where the data—perhaps in encrypted form—is initially held by the server), at least when the memory-access pattern is chosen non-adaptively; specifically, initialization can be done by sending the key defining the position map to the server, who then arranges the data blocks as needed.

2 Background

2.1 Oblivious RAM

We use the standard definitions of correctness and security for ORAM [**?**], repeated here for completeness. Readers familiar with these definitions can safely skip to the next section.

For fixed N, B, we define a *memory access* to be a tuple (op, i, v) where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $i \in \{0, \ldots, N-1\}$, and $v \in \{0,1\}^B$. Let D be an N-element array containing B-bit entries. The result of applying (read, i, v) to D is $D[i]$, and the array D is unchanged. The result of applying (write, i, v) is $\perp$, and D is updated to a new array D' that is identical to D except that $D'[i] = v$. Given an initial array D and a sequence of memory accesses $(\mathsf{op}_1, i_1, v_1)$, $\ldots$, $(\mathsf{op}_M, i_M, v_M)$, we define correctness for the sequence of results $o_1, \ldots, o_M$ in the natural way; namely, the sequence of results is correct iff, for all t, the result o_t is equal to the last value written to i_t (or is equal to $D[i_t]$ if there were no previous writes to i_t).

A two-server, one-round ORAM scheme is defined by a collection of four algorithms $\mathsf{ORAM.Init}$, $\mathsf{ORAM.C}$, $\mathsf{ORAM.S}$, and $\mathsf{ORAM.C'}$ with the following syntax:

- $\mathsf{ORAM.Init}$ takes as input 1^λ, 1^κ and elements $D[0], \ldots, D[N-1] \in \{0,1\}^B$. It outputs state st and data T to be stored at the servers.
- $\mathsf{ORAM.C}$ takes as input st and a memory access (op, i, v). It outputs updated state $\mathsf{st'}$ along with a pair of queries q_0, q_1.
- $\mathsf{ORAM.S}$ takes as input data T and a query q. It outputs updated data T' and a response r.

– $\mathsf{ORAM.C'}$ takes as input state st and a pair of responses r_0, r_1. It outputs updated state $\mathsf{st'}$ and a value o.

We define correctness and security via an experiment Expt. Given an array D (which defines the parameters N and B) and a sequence of memory accesses $\mathsf{seq} = ((\mathsf{op}_1, i_1, v_1), \ldots, (\mathsf{op}_M, i_M, v_M))$, experiment $\mathsf{Expt}(1^\lambda, 1^\kappa, D, \mathsf{seq})$ first runs $(\mathsf{st}_0, T_0) \leftarrow \mathsf{ORAM.Init}(1^\lambda, 1^\kappa, D)$ and sets $T_{0,0} = T_{0,1} = T_0$. Then, for $t = 1$ to M it does:

1. Run $(\mathsf{st}'_{t-1}, q_{t,0}, q_{t,1}) \leftarrow \mathsf{ORAM.C}(\mathsf{st}_{t-1}, (\mathsf{op}_t, i_t, v_t))$.
2. Run $(T_{t,b}, r_{t,b}) \leftarrow \mathsf{ORAM.S}(T_{t-1,b}, q_{t,b})$ for $b \in \{0, 1\}$.
3. Run $(\mathsf{st}_t, o_t) \leftarrow \mathsf{ORAM.C'}(\mathsf{st}'_{t-1}, r_{t,0}, r_{t,1})$.

Let $\mathsf{view}_b = (T_0, q_{1,b}, \ldots, q_{M,b})$. The output of the experiment is $(\mathsf{view}_0, \mathsf{view}_1, o_1, \ldots, o_M)$.

Correctness requires that for any polynomial M there is a negligible function negl such that for any λ, κ, D, and sequence of $M = M(\lambda)$ memory accesses $\mathsf{seq} = ((\mathsf{op}_1, i_1, v_1), \ldots, (\mathsf{op}_M, i_M, v_M))$, if we compute $(\mathsf{view}_0, \mathsf{view}_1, o_1, \ldots, o_M) \leftarrow \mathsf{Expt}(1^\lambda, 1^\kappa, D, \mathsf{seq})$ then the sequence of results $o_1, \ldots, o_M$ is correct (for D and seq) except with probability $\mathsf{negl}(\lambda)$.

An ORAM protocol is secure if for any λ and PPT adversary A the following is negligible in κ:

$$\left| \Pr \left[\begin{array}{c} (D_0, \mathsf{seq}_0, D_1, \mathsf{seq}_1) \leftarrow A(1^\lambda, 1^\kappa); b \leftarrow \{0,1\}; \\ (\mathsf{view}_0, \mathsf{view}_1, o_1, \ldots, o_M) \leftarrow \mathsf{Expt}(1^\lambda, 1^\kappa, D_b, \mathsf{seq}_b) \end{array} : A(\mathsf{view}_0) = b \right] - \frac{1}{2} \right|$$

(and analogously for view_1), where D_0, D_1 have identical parameters N, B, and where $\mathsf{seq}_0, \mathsf{seq}_1$ have the same length. As usual, this notion of security assumes the servers are honest-but-curious.

We remark that, as is typical in this setting, both correctness and security are defined with respect to a non-adaptive selection of inputs (in terms of both the original data and the sequence of memory accesses). Our scheme remains secure even for adaptively chosen inputs, though in that case we cannot use the optimized initialization procedure discussed at the end of Section **??**.

2.2 Private Path Retrieval

We review the notion of *private information retrieval (PIR)*, and propose an extension that we call *private path retrieval (PPR)*. We then describe an efficient construction of a two-server PPR scheme based on a two-server PIR scheme of Boyle et al.

Abstractly, a PIR scheme allows a client to obliviously learn one value out of an array of N values stored by a pair of servers. Specialized to XOR-based, one-round protocols in the two-server setting, we define a PIR scheme as a pair of algorithms $(\mathsf{PIR.C}, \mathsf{PIR.S})$ with the following syntax:

– $\mathsf{PIR.C}$ is a randomized algorithm that takes as input parameters $1^\kappa, B, N$, and an index $i \in \{0, \ldots, N-1\}$. It outputs a pair of queries q_0, q_1.

– $\mathsf{PIR.S}$ is an algorithm that takes as input $D[0], \ldots, D[N-1] \in \{0,1\}^B$, and a query q. It outputs a response r.

Correctness requires that for all κ, B, N, i, and D as above, we have

$$\Pr\left[\begin{array}{c}(q_0, q_1) \leftarrow \mathsf{PIR.C}(1^\kappa, B, N, i); \\ \{r_b := \mathsf{PIR.S}(D, q_b)\}_{b\in\{0,1\}} \ : \ r_0 \oplus r_1 = D[i]\end{array}\right] = 1.$$

A PIR scheme can be used by a client C and a pair of servers S_0, S_1 in the natural way. S_0 and S_1 each begin holding identical copies of an N-element array D of B-bit data blocks. When C wants to learn the element located at address i, it computes $(q_0, q_1) \leftarrow \mathsf{PIR.C}(1^\kappa, B, N, i)$ and sends q_b to S_b. The servers compute their corresponding responses r_0, r_1, and send them to the client. The client can then recover $D[i]$ by computing $D[i] = r_0 \oplus r_1$.

Security requires that neither server learns anything about the client's desired address i. In other words, it is required that for all B, N, i, i', and $b \in \{0,1\}$ the following distributions are computationally indistinguishable (with security parameter κ):

$$\{(q_0, q_1) \leftarrow \mathsf{PIR.C}(1^\kappa, B, N, i) : q_b\} \quad \text{and} \quad \{(q_0, q_1) \leftarrow \mathsf{PIR.C}(1^\kappa, B, N, i') : q_b\}.$$

Private path retrieval. For our application, we extend PIR to a new primitive that we call *private path retrieval* (PPR). Here, we view the data stored by the servers as being organized in a depth-L binary tree with $N = 2^L$ leaves; the client wishes to obliviously obtain all the values stored on some *path* in that tree from the root to a leaf. (In fact, it will be convenient to omit the root itself.) Formally, and again specializing to XOR-based, one-round protocols in the two-server setting, we define a PPR scheme as a pair of algorithms $(\mathsf{PPR.C}, \mathsf{PPR.S})$ with the following syntax:

– $\mathsf{PPR.C}$ is a randomized algorithm that takes as input parameters $1^\kappa, B, N$, and an index $i \in \{0, \ldots, N-1\}$ corresponding to a leaf node. It outputs a pair of queries q_0, q_1.
– $\mathsf{PPR.S}$ is an algorithm that takes as input a tree T of elements $T[x] \in \{0,1\}^B$, for $x \in \{0,1\}^{\leq \log N}$, and a query q. It outputs a response vector $r^1, \ldots, r^L$.

Representing $i \in \{0, \ldots, N-1\}$ as an L-bit integer in the obvious way, we let $\langle i \rangle_t$ denote the t-bit prefix of i for $1 \leq t \leq L$. Correctness for a PPR scheme requires that for all κ, B, N, i, and T as above, and all $t \in \{1, \ldots, L\}$, we have

$$\Pr\left[\begin{array}{c}(q_0, q_1) \leftarrow \mathsf{PPR.C}(1^\kappa, B, N, i); \\ \{(r_b^1, \ldots, r_b^L) := \mathsf{PPR.S}(1^\kappa, T, q_b)\}_{b\in\{0,1\}} \ : \ r_0^t \oplus r_1^t = T[\langle i \rangle_t]\end{array}\right] = 1.$$

Security requires that neither server learns anything about the client's desired path. That is, we require that for all B, N, i, i', and $b \in \{0,1\}$ the following distributions are computationally indistinguishable (with security parameter κ):

$$\{(q_0, q_1) \leftarrow \mathsf{PPR.C}(1^\kappa, B, N, i) : q_b\} \quad \text{and} \quad \{(q_0, q_1) \leftarrow \mathsf{PPR.C}(1^\kappa, B, N, i') : q_b\}.$$

Constructing a PPR scheme. It is immediate that any PIR scheme can be used generically to construct a PPR scheme. Briefly: the servers view the the tree they store as a collection of L arrays, with the ith level of the tree corresponding to an array D_i containing 2^i elements. The client can then obliviously retrieve a path in the tree by running any underlying PIR protocol L times, once for each array $D_1, \ldots, D_L$. This increases both the client-to-server and the server-to-client communication by roughly a factor of L. This construction is "overkill," though, in the sense that it allows the client to retrieve an *arbitrary* data block at each level of the tree, whereas a PPR scheme only needs to support retrieval of data blocks along a *path*. This suggests that it may be possible to further optimize the construction.

Indeed, we show that by adapting the specific PIR scheme of Boyle et al. a better solution is possible. The communication complexity of their basic PIR scheme is $2B + O(\kappa \log N)$; thus, the generic construction sketched above would give a PPR scheme with communication complexity $2B \log N + O(\kappa \log^2 N)$. We show how to improve this to $2B \log N + O(\kappa \log N)$.

Rather than give the details of the PIR scheme of Boyle et al., we describe their scheme abstractly. To retrieve the ith element of an array D of length N, the client in their scheme sends each server S_b a query of length $\kappa+1+(\kappa+2)\cdot\log N = O(\kappa \log N)$ bits; the query enables that server to compute a sequence of bits $\lambda_b[0], \ldots, \lambda_b[N-1]$ with the property that $\lambda_0[j] \oplus \lambda_1[j] = 1$ iff $j = i$. Server S_b then responds with $r_b = \bigoplus_{j=0}^{N-1} \lambda_b[j] \cdot D[j]$. It is easily verified that $r_0 \oplus r_1 = D[i]$.

To construct a PPR scheme, we leave the client algorithm unchanged. Let i denote the leaf corresponding to the path the client wishes to retrieve. As before, server S_b then computes a sequence of bits $\lambda_b[0], \ldots, \lambda_b[N-1]$ where $\lambda_0[j] \oplus \lambda_1[j] = 1$ iff $j = i$. Each server then constructs a logical binary tree of depth $L = \log N$ with the λ-values at the leaves, and recursively defines the values at each internal node of this logical tree to be the XOR of the values of its children. In this way, each server S_b obtains[3] a collection of bits $\{\lambda_b[x]\}_{x \in \{0,1\}^{\leq L}}$ with the property that $\lambda_0[x] \oplus \lambda_1[x] = 1$ iff x is a prefix of i (or, in other words, iff the node corresponding to x is on the path from the root to the ith leaf). Server S_b then computes the sequence of responses $r_b^t = \bigoplus_{x\,:\,|x|=t} \lambda_b[x] \cdot T[x]$ for $1 \leq t \leq L$. One can verify that $r_0^t \oplus r_1^t = T[\langle i \rangle_t]$ for all t. Note also that security of the PPR scheme is implied immediately by security of the original PIR scheme, which in turn is based on the existence of pseudorandom functions.

Summarizing, we have:

Theorem 1. *Assuming the existence of pseudorandom functions, there is a two-server PPR scheme in which the client sends each server a query of length $O(\kappa \log N)$, and each server sends back a response of length $B \cdot \log N$.*

[3] Readers familiar with the construction of Boyle et al. may observe that these values are already implicitly defined as part of their scheme; we explicitly describe the computation of these values for self-containment.

3 A Two-Server ORAM Scheme

We now present our two-server ORAM scheme, which can be viewed as being constructed by adapting the ring ORAM protocol [**?**] to the two-server setting and then combining it with the PPR scheme from Section **??**. We build on ring ORAM for concreteness, but our general idea can also be applied to several other tree-based ORAM schemes from the literature (e.g., [**?**,**?**,**?**]).

3.1 Description of our Scheme

Preliminaries. The client's data is viewed as a sequence of $N = 2^L$ data blocks $D[0], \ldots, D[N-1] \in \{0,1\}^B$. Each server stores identical copies of a depth-L, full binary tree T with N leaves numbered from 0 to $N-1$; we number the levels of the tree from the root at level 0 to the leaves at level L, and refer to each node of the tree (except the root) as a *bucket*. (The root will be treated differently from the other nodes; see further below.)

As in other tree-based ORAM schemes, the client maintains a position map that maps logical memory addresses to leaves in T. In our case, the position map will be static and we implement it by a pseudorandom function $F_K : [N] \to [N]$, with K chosen by the client. For $\mathsf{pos} \in \{0, \ldots, 2^L - 1\}$ denoting a leaf in T, we let $\mathcal{P}(\mathsf{pos})$ denote the path consisting of all buckets in the tree from the root to that leaf.

A *record* $(\mathsf{flag}, i, \mathsf{pos}, \mathsf{data}) \in \{0,1\} \times \{0,1\}^{\log N} \times \{0,1\}^{\log N} \times \{0,1\}^B$ contains four fixed-length fields, encrypted using a key held by the client. (For simplicity in what follows, we omit explicit mention of encrypting/decrypting these blocks.) If $\mathsf{flag} = 1$ then the record is *real* and we have $\mathsf{pos} = F_K(i)$ and $\mathsf{data} = D[i]$; if $\mathsf{flag} = 0$ then the record is a *dummy record* and $i, \mathsf{pos}, \mathsf{data}$ can be arbitrary (so long as they are the correct length). Each bucket in the binary tree stored by the servers contains Z records, where Z is a parameter we fix later.

As an optimization, we have the client store the root of the tree and refer to the root as the *stash*. (We stress, however, that when we refer to a path $\mathcal{P} = \mathcal{P}(\mathsf{pos})$ in the tree, that path always includes the root/stash.) All records in the stash are real, and we allow the stash to store more than Z records. Of course, the records in the stash do not need to be encrypted.

Invariant. In our scheme, the servers store identical copies of the tree T at all times. As in other tree-based ORAM schemes, we maintain the invariant that, for all i, there is always a (real) record $(1, i, \mathsf{pos}, D[i])$ located in some bucket on $\mathcal{P}(\mathsf{pos})$. It is possible that multiple real records with the same index appear in the tree at the same time; in this case, the one closest to the root is always the most up-to-date copy.

Accessing memory. To read logical address i of its array, the client simply needs to read the path $\mathcal{P}(F_K(i))$ and then find the corresponding record closest to the root. For obliviousness, reading this path is done using our PPR scheme. A logical write of the value v to address i of the array is done by storing the

record $(1, i, F_K(i), v)$ in the stash (removing from the stash any outdated record with the same logical address, if necessary).

Eviction. As described, writing to the array will cause the number of records stored in the stash to grow without bound. We prevent this by performing an *eviction procedure* after every A memory accesses, where A is a configurable parameter. This eviction procedure reads a path $\mathcal{P}$ in the tree, updates the buckets in that path, and then writes the updated path $\mathcal{P}'$ back to the servers. To fully specify this process, we need to determine two things: (1) how the paths to be evicted are chosen and (2) how the chosen paths are updated.

- Following Gentry et al. [**?**], we choose paths to be evicted according to a deterministic schedule, namely, in reverse lexicographic order. This is also the schedule used in ring ORAM. Note that using a deterministic schedule ensures obliviousness.
- Our update procedure is similar (but not exactly identical) to the one used in path ORAM [**?**] and ring ORAM [**?**]. As in those schemes, we update a path $\mathcal{P}$ by pushing every real record $(1, i, \mathsf{pos}, v)$ in that path as far down the tree as possible, subject to the constraint that it must be located on $\mathcal{P}(\mathsf{pos})$ (and the constraint that each bucket holds at most Z records). In addition, prior to doing this, we also clear out any stale records in $\mathcal{P}$. That is, if for any i there are multiple records of the form $(1, i, \mathsf{pos}, \star)$ in $\mathcal{P}$, then only the one closest to the root is kept; the rest are replaced with dummy records.

We give a formal description of our scheme, assuming initialization of the tree has already been done, in Figure **??**. See below for a discussion of initialization.

Parameters. Each record has length exactly $1 + 2\log N + B$ bits before being encrypted.[4] Encryption adds at most κ additional bits; this can be reduced by using a global counter keeping track of how many records have been encrypted thus far. We let R denote the size, in bits, of a record (after encryption). If $\kappa = O(B)$ and $B \geq \log N$ (which is typical in practice), we have $R = O(B)$.

As described, the client's stash can grow arbitrarily large. We show in the next section that when $A = 1$ (i.e., eviction is done after every access) and $Z = 3$ the client's stash contains at most λ records except with probability negligible in λ. The servers each hold fewer than $2N$ buckets, with each bucket containing Z records; thus, for the parameter settings discussed above, each server's storage is at most $2ZNR = O(BN)$ bits.

The total communication for a logical memory access can be computed as follows:

1. As part of the PPR scheme, the client sends $O(\kappa \log N)$ bits to each server, and each server responds with $RZ \log N$ bits.
2. For eviction, one server sends $RZ \log N$ bits to the client, and then the client sends $RZ \log N$ bits to each server.

[4] As a small optimization, $F_K(i)$ need not be stored in a record, as the client can recompute it when needed.

The state of the client includes the stash, a key K, and a counter ctr initialized to 0 that indicates the next eviction path. The servers store ctr and identical copies of a tree T. The parameter A determines how often eviction is done.

On input (op, i, v) do:

1. Let $\mathsf{pos} := F_K(i)$.
2. The client uses PPR to read $\mathcal{P}(\mathsf{pos})$ from T.
3. If $\mathsf{op} = \mathsf{read}$ then scan through $\mathcal{P}(\mathsf{pos})$ to find the real record $(1, i, \mathsf{pos}, v_i)$ closest to the root, and output v_i.
4. If $\mathsf{op} = \mathsf{write}$ then (1) remove any records of the form $(1, i, \mathsf{pos}, \star)$ from the stash, and (2) add the record $(1, i, \mathsf{pos}, v)$ to the stash.
5. Set $\mathsf{ctr} = \mathsf{ctr} + 1 \bmod A \cdot N$. If $\mathsf{ctr} = 0 \bmod A$ then run procedure $\mathsf{Evict}(\mathsf{ctr}/A)$.

$\mathsf{Evict}(\mathsf{ctr})$:

1. Let $\mathcal{P}$ be the path corresponding to ctr under reverse lexicographic order. Request $\mathcal{P}$ from one of the servers.
2. If, for any i, there are multiple (real) records $(1, i, \mathsf{pos}, \star)$ in $\mathcal{P}$, then only the one closest to the root is kept; the rest are replaced with dummy records.
3. Process the remaining real records one-by-one, starting from the root. For each such record $\mathsf{record} = (1, i, \mathsf{pos}, v)$, find the bucket in $\mathcal{P}$ furthest from the root that (1) is on $\mathcal{P}(\mathsf{pos})$ and (2) contains fewer than Z real records. Put record in that bucket in place of a dummy block. (If no such bucket is found then keep record where it is.) Finally, (re-)encrypt all records in the updated buckets.
4. The updated path $\mathcal{P}'$ is then written back to both servers.

Fig. 1. Our two-server ORAM scheme.

Thus, for the parameter settings discussed above, the total communication complexity is $O(B \log N)$ even when $A = 1$. Importantly, the constants are small; the worst-case communication (for general parameters) is at most

$$\kappa + 1 + (\kappa + 2) \cdot \log N + 5Z \cdot (\kappa + 2 \log N + B) \cdot \log N$$

bits, and the amortized communication (in bits) is

$$\kappa + 1 + (\kappa + 2) \cdot \log N + \left(2Z + \frac{3Z}{A}\right) \cdot (\kappa + 2 \log N + B) \cdot \log N.$$

Thus, as in path ORAM, we can trade off Z and A to reduce communication.

As described (and taking $A = 1$), the protocol uses three messages if we piggyback the server's eviction message with its response in the PPR scheme. However, if we delay the client's eviction message until the next time the client initiates the PPR protocol (for the next memory access), then we obtain a one-round protocol. Since the client must now store the updated path $\mathcal{P}'$ between memory accesses, this increases the storage of the client by $ZR \log N$ bits.

Initialization. Initialization can be done locally at the client by starting with a tree consisting only of dummy records and then simulating the process of writing each data block of the original array; the resulting tree is then uploaded to each server. We additionally observe that in settings where the servers initially hold the array (in encrypted form), initialization can be done in essentially the same way—but locally at each server—by having the client simply send K to the servers.[5]

3.2 Analysis

Correctness of our protocol follows by inspection, and obliviousness follows from obliviousness of the PPR scheme and the fact that a deterministic eviction procedure is used. Thus, in the remainder of this section we focus on analyzing the efficiency of the scheme, specifically, the size of the stash stored by the client. Compared to the similar analysis done for ring ORAM and other tree-based schemes, there are two differences: first, in our scheme the tree may contain stale records (i.e., real records $(1, i, \mathsf{pos}, v)$ that have been superseded by a more up-to-date record stored closer to the root on the same path $\mathcal{P}(\mathsf{pos})$); second, in our scheme the position map is fixed once-and-for-all rather than being updated each time a memory access is done. Careful examination of the proofs for prior tree-based ORAM schemes, however, shows that both of these changes have no effect on the final bound. Nevertheless, we include details of the analysis (following [**?**]) for completeness.

Recall that we assume eviction is done after every A accesses. We define the size of the stash after the Mth memory access to be the size of the stash following the last invocation of the eviction procedure. (In our one-round scheme the eviction procedure following the Mth memory access is not completed until the $(M + 1)$st memory access takes place; this difference can only increase the size of the stash by a single record.)

During the execution of our ORAM scheme, the resulting tree stored by the servers can contain two types of real records. We call a real record $(1, i, \mathsf{pos}, v)$ *stale* if there is another real record $(1, i, \mathsf{pos}, \star)$ stored closer to the root (including at the root itself); otherwise, we call the record *fresh*. Note that there is exactly one fresh record stored in the tree at any point in time for each logical memory address i. An important observation is that *stale records have no impact on the stash*. More formally:

Lemma 1. *Consider modifying the ORAM protocol (Figure* **??***), so that in step 4 of processing a write operation the client also marks any stale records*

[5] Revealing the key to the servers does not affect the security of our scheme since we do not rely on secrecy of K for obliviousness. Rather, we rely on pseudorandomness of F_K only for bounding the size of the stash. We remark, however, that our analysis of the stash size assumes that the client's sequence of memory accesses is chosen independently of K. Thus, the optimized initialization (in which the client sends K to the servers) is only applicable when the client's sequence of memory accesses is not under adaptive control of an adversarial server.

corresponding to logical address i as dummy records (without regard for obliviousness). This modification does not affect the size of the stash, regardless of the position map or the sequence of memory accesses.

Proof. The only time a stale record can possibly have any effect on the stash in an execution of the real protocol is if there is a stale record (corresponding to some logical address i) in a path $\mathcal{P}$ being processed in step 3 of the eviction subroutine. But then the fresh record corresponding to address i is also in $\mathcal{P}$ at that moment, and so the stale record would have been replaced with a dummy record in step 2 of the eviction subroutine. □

A consequence of the above is that we may treat stale records as dummy records in our analysis, and it suffices for us to keep track of the placement of fresh records.

Fix a memory-access sequence seq of length M. We assume the binary tree T stored by the servers is initially filled entirely with dummy records; we thus let seq include the memory accesses done as part of initialization. For the purposes of proving a bound on the size of the stash, we may assume that all operations in seq are writes; moreover, the data values being written are irrelevant, and so we can simply focus on the sequence of logical memory addresses being accessed. If τ is a subtree of T, then we let $n(\tau)$ denote the number of nodes in τ. A subtree is *rooted* if it contains the root, and root denotes the root node (which is itself a rooted subtree).

We treat the position map as a random function $f : [N] \to [N]$ chosen independently of the memory-access sequence. For a subtree τ we let τ_Z be a random variable denoting the number of fresh records stored in each node of τ after our ORAM scheme (with bucket size Z) is used to carry out the sequence of memory accesses in seq. As in prior work [**?**,**?**], we let τ_∞ refer to the same random variable when buckets can hold an unbounded number of records. We let $X(\tau_Z)$ be a random variable denoting the total number of fresh records stored in τ_Z. (Using this notation, we are interested in bounding $X(\mathsf{root}_Z)$.) We let $X_i(\tau_Z)$ be a random variable denoting the number of fresh records corresponding to logical address i that are in τ_Z; note that $X_i(\tau_Z) \in \{0, 1\}$.

We rely on the following result proved in prior work [**?**,**?**] for the same eviction procedure we use (when focusing on fresh blocks):

Lemma 2. *For any Z, S, it holds that*

$$\Pr[X(\mathsf{root}_Z) > Z + S] \leq \sum_{n \geq 1} 4^n \cdot \max_{\tau\,:\,n(\tau)=n} \Pr[X(\tau_\infty) > Z \cdot n(\tau) + S],$$

where the maximum is over rooted subtrees τ of T.

The following result depends on the specifics of the eviction procedure and the position map. Nevertheless, the end result we obtain for our scheme is the same as what is shown in prior work.

Lemma 3. *Set $A = 1$ in our scheme. If b is a leaf node, $\mathbf{Exp}[X(b_\infty)] \leq 1$. If b is an internal node, $\mathbf{Exp}[X(b_\infty)] \leq 1/2$.*

Proof. If b is a leaf node, then a fresh record corresponding to logical address i can only possibly be stored in that node if i is mapped to b by the position map. Since there are N logical addresses, and each is mapped to b with probability $1/N$, the claimed bound follows.

Say b is a non-leaf node at level ℓ. If b is not on any of the first M eviction paths (note that this is independent of seq or the position map f), then b will contain no fresh records. Otherwise, let $1 \leq \mathsf{ctr}_1 \leq M$ denote the last time b was on an eviction path, and let $\mathsf{ctr}_0 < M$ denote the penultimate time b was on an eviction path (set $\mathsf{ctr}_0 = 0$ if there was no such time). By the properties of reverse lexicographic ordering, we have $\mathsf{ctr}_1 - \mathsf{ctr}_0 \leq 2^\ell$. The only possible fresh records that can be in b after all M instructions are executed are those corresponding to logical write addresses used in time steps $\mathsf{ctr}_0 + 1, \ldots, \mathsf{ctr}_1$. Moreover, each such address causes a fresh record to be placed in bucket b with probability exactly $2^{-(\ell+1)}$. Thus, the expected number of fresh records in b is at most $2^\ell \cdot 2^{-(\ell+1)} = 1/2$. □

A corollary is that if τ is a rooted subtree then $\mathbf{Exp}[X(\tau_\infty)] \leq 0.8 \cdot n(\tau)$ for all $N \geq 4$ (since in that case at most $N/(2N-1) \leq 4/7$ of the nodes in τ can be leaves). Following the analysis of Ren et al. [**?**, Section 4.3] (taking $a = 0.8$), we may then conclude that when $Z \geq 3$, the probability of overflow decreases exponentially in S. This implies that the stash will not exceed λ records except with probability negligible in λ.

In Table **??** we report concrete bounds on the number of blocks in the client's stash for different values of the bucket size Z and eviction parameter A. All values in the table are obtained from our theoretical analysis assuming N is sufficiently large. Simulations indicate that the stash size is even smaller than what the theoretical bounds indicate.

	$Z = 3$	$Z = 4$	$Z = 5$	$Z = 6$	$Z = 7$
$A = 1$	16	14	13	12	11
$A = 2$	-	21	18	16	15
$A = 3$	-	32	24	21	19
$A = 4$	-	-	33	26	23
$A = 5$	-	-	-	34	28

Table 1. Bounds on the number of blocks in the client's stash. These bounds hold except with probability 2^{-40} (per operation).

3.3 Optimizations

We briefly mention a few optimizations.

Heuristic parameters. As in the ring ORAM scheme, we experimentally observe that it suffices to set $A = 1$ and $Z = 2$ (giving the parameters mentioned

in the abstract/introduction), or to set $A = 3$ and $Z = 3$ (giving slightly better communication at the expense of increased server storage).

A two-round variant. If we are willing to use one more round, the communication complexity can be further reduced by first having the client use PPR to read the indices in the records on the desired path, and then using an execution of PIR to read the single record of interest.

Acknowledgments

This material is based on work supported by NSF awards #1111599, #1563722, and #1564088.