Isogrammic-Fusion ORAM: Improved Statistically Secure Privacy-Preserving Cloud Data Access for Thin Clients

Michael T. Goodrich
University of California, Irvine
goodrich@uci.edu

ABSTRACT

We study oblivious random access machine (ORAM) simulation, in cloud computing environments where a ***thin client*** outsources her data to a server using $O(1)$-sized messages.

KEYWORDS

ORAM, cloud storage, oblivious storage, fusion trees

ACM Reference Format:
Michael T. Goodrich. 2018. Isogrammic-Fusion ORAM: Improved Statistically Secure Privacy-Preserving Cloud Data Access for Thin Clients. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3196494.3196500

1 INTRODUCTION

In the ***cloud storage*** paradigm, a client, Alice, outsources her data to a server, Bob, who stores Alice's data and provides her with an interface to access it from anywhere in the network. We assume that Bob is "honest-but-curious," in that Bob is trusted to keep Alice's data safe and available, but he wants to learn as much as possible about Alice's data. The challenge, then, is for Alice to obfuscate not just the values of her data through encryption but also obfuscate her data access pattern. Fortunately, in support of this obfuscation goal, there is a large and growing literature on methods for simulating a RAM algorithm to achieve obliviousness. Such oblivious algorithm simulation methods provide ways for Alice to privately outsource her data to Bob by replacing each single access in an algorithm, $\mathcal{A}$, that Alice is executing into a set of accesses. Formally, we assume Alice outsources a storage of size n, with indices in $[0, n-1]$, and that $\mathcal{A}$ uses the following operations:

- write(i, v): Write v into memory cell i.
- read(i): Read v from memory cell i.

A related concept is ***oblivious storage*** (OS), e.g., see [13, 16–18], where Alice wishes to store a dictionary at the server, Bob, of size at most n, and her algorithm, $\mathcal{A}$, accesses this dictionary using the following operations:

- put(k, v): Add the pair (item), (k, v). This causes an error if there is already an item with key k.
- get(k): Return and remove the item, (k, v), associated with the key, k. If there is no such item, then this operation returns NOT-FOUND.

We assume Alice's algorithm, $\mathcal{A}$, will issue an access sequence of some given length, N, where N is at most polynomial in n. The challenge is for Alice to make sure that Bob learns nothing about her access sequence beyond the values of n and N, that is, that she achieves statistical secrecy for her data access pattern.

Let σ denote a sequence of N read/write operations (or get/put operations). An ORAM (or OS) scheme transforms σ into a sequence, σ', of operations. We assume that each item is stored using a semantically-secure encryption scheme, so that independent of whether Alice wants to keep a key-value item unchanged, for each access, the sequence σ' involves always replacing anything Alice accesses with a new encrypted value so that Bob is unable to tell if the underlying plaintext value has changed.

The security for an ORAM (or OS) simulation is defined in terms of a computational game. Let σ_1 and σ_2 be two different access sequences, of length N, for a key/index set of size n, that are chosen by Bob and given to Alice. Alice chooses uniformly at random one of these sequences and transforms it into access sequence σ' according to her ORAM (OS) scheme, which she then executes according to this scheme. Alice's ORAM (OS) scheme is ***statistically secure*** at hiding her access pattern if Bob can determine which sequence, σ_1 or σ_2, Alice chose with probability at most $1/2$.

The ***I/O overhead*** for such an OS or ORAM scheme is a function, $T(n)$, such that the total number of messages sent between Alice and Bob during the simulation of all N of her accesses from σ is $O(N \cdot T(n))$ with high probability.

In this paper, we provide methods for improving the asymptotic I/O overhead for such ORAM simulations. The approach we take to achieve this goal is to first transform the original RAM access sequence, σ, into an intermediate OS sequence, $\hat{\sigma}$, which has a restricted structure that we refer to as it being ***isogrammic*** [1], and we then efficiently transform the isogrammic sequence, $\hat{\sigma}$, into the final access sequence, σ', by taking advantage of this structure. We define a sequence, $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_N)$, of put and get operations to be ***isogrammic*** if the following conditions are satisfied [11]:

(1) For every get(k) operation, there is a previous put(k, v) operation.
(2) For any put(k, v) operation, there is not already an item in the set with key k.
(3) For each put(k, v) operation, the key k includes a component that is chosen uniformly and independently at random from a sufficiently large key space (which depends, e.g., on configuration parameters).

[1] An ***isogram*** is a word, like "uncopyrightable," without a repeated letter. E.g., see wikipedia.org/wiki/Isogram.

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196500

Thus, isogrammic access sequences do not result in error conditions, never get NOT-FOUND responses, and use keys that are substantially random. The following parameters are also relevant:

- w: The word size in bits for indices and keys. As is standard (e.g., see [9]), we assume $w \geq \log n$, so an index or key can be stored in $O(1)$ memory words.
- M: The number of words in Alice's private memory.
- B: The maximum number of words in a message block sent from/to Alice in one I/O operation.

In this paper, we are interested in scenarios where M is $o(\log n)$ and B is $O(1)$, so as to design efficient statistically-secure ORAM simulation schemes for ***thin clients***, that is, clients that have asymptotically sublogarithmic-sized local private memory and use constant-sized messages to communicate with the server. Moreover, we desire schemes whose I/O overhead bounds hold with high probability.

Besides being of interest in its own right, and returning to the original thin-client models for ORAM simulation [6, 9], designing improved ORAM simulation methods for thin clients also can improve other cryptographic primitives and protocols, which use ORAM simulations as black-box or white-box constructions, such as dynamic searchable encryption and secure two-party computation.

Previous Related Results. ORAM simulation methods originate from the seminal work of Goldreich and Ostrofsky [9], who achieve an I/O overhead of $O(\log^3 n)$ for thin clients, with M and B being $O(1)$, with a scheme that fails with polynomial probability and requires the use of a random oracle (hence, is not statistically secure). Subsequently, Goodrich and Mitzenmacher [12] present an ORAM simulation with an $O(\log n)$ I/O overhead and constant-sized messages, and their method is also not statistically secure. In addition, it requires that M be $\Omega(n^\epsilon)$, for some fixed constant $\epsilon > 0$, i.e., their result is not for thin clients. Kushilevitz *et al.* [14] improve the I/O overhead for thin clients (with a constant-size client-side memory) to be $O(\log^2 n/\log\log n)$, but their method assumes the existence of random oracles.

Addressing the more challenging goal of designing an ORAM simulation method that achieves statistical security, Damgård *et al.* [6] introduce an ORAM scheme that is statistically secure and achieves an I/O overhead of $O(\log^3 n)$. Moreover, their method works for thin clients, with M and B being $O(1)$; in fact, their method remained until the present paper the best previous statistically secure ORAM method for thin clients. Stefanov *et al.* [19] introduce the Path ORAM method, which is statistically secure but not suited for thin clients, in that it uses B-sized messages and assumes B is $\omega(\log n)$ and M is $\omega(B\log n)$, to achieve an I/O overhead of $O(\log^2 n/\log B)$. Chung *et al.* [3] provide an alternative "supermarket ORAM" implementation for the case when B is $O(1)$ and M is polylogarithmic, which has an I/O overhead of $O(\log^2 n\log\log n)$, but their method is nevertheless not a solution for thin clients. Still, applying an observation of Chung and Pass for their "Simple-ORAM" scheme [4], the supermarket ORAM method can be made to work on a thin client, but then the overhead becomes worse that the overhead for the scheme of Damgård *et al.* [6]. These schemes for non-thin clients use a simple tree-based approach (which our scheme also uses), but, in order to achieve efficient I/O overheads, they require an additional complication of carefully-implemented recursive applications of their approaches, which take away from their simplicity. Ohrimenko *et al.* [16] present an oblivious storage (OS) scheme that achieves an I/O overhead of $O(1)$, but it requires that M and B be $\Omega(n^\epsilon)$, for a fixed constant $\epsilon > 0$, and it assumes the existence of random oracles. Addressing a more restricted problem than ORAM or OS simulation, Wang *et al.* [21] introduce an interesting "oblivious data structure" framework, which applies to algorithms that use a small number of bounded-degree data structures, such as stacks, queues, or search trees, to achieve an $O(\log n)$ I/O overhead for data-structure access sequences. Their work can be seen a precursor to isogrammic access sequences. Unfortunately, their algorithms are based on the (non-recursive) Path ORAM of Stefanov *et al.* [19], however, which requires that M and B be $\omega(\log n)$; hence, their results for oblivious data structures are not for thin clients, but instead require superlogarithmic-sized client-side memory and superlogarithmic-sized messages to be exchanged between the client and the server. The Circuit ORAM [20], improves the circuit complexity for such tree-based ORAM simulations, but still requires blocks to be of size at least $\Omega(\log^2 n)$; hence, it is also not designed for thin clients.

At a high level, our work is similar to the recent BIOS ORAM scheme of Goodrich [11], which is a tree-based ORAM scheme that uses B-trees instead of binary trees to improve the I/O overhead for ORAM simulation for non-thin clients. Like our scheme, his scheme avoids recursion by a reduction to isogrammic access sequences, but his scheme is not for thin clients.

Our Results. We provide statistically secure ORAM simulation methods for thin clients that asymptotically improve the I/O overheads of previous statistically secure ORAM simulation methods for thin clients. We summarize our results in Table 1. given in an appendix. Our isogrammic-fusion ORAM scheme is the first statistically secure ORAM to achieve an I/O overheads of $O(\log^2 n)$ or $O(\log^2 n\log\log n)$ for thin clients, whereas the previous best result for this cloud-computing scenario, by Damgård *et al.* [6], has an I/O overhead of $O(\log^3 n)$.

We refer to our approach to ORAM simulation as ***isogrammic-fusion*** ORAM, due to its combination of two concepts for ORAM simulations. The first concept is the exploitation of ***isogrammic*** access sequences [11]. This framework extends the oblivious data structure framework of Wang *et al.* [21], which is only for fat clients. The second concept we use is the main technical device we utilize to achieve our results, which is a bit-level parallel data structure known as the ***fusion tree*** [2, 8]. By "***bit-level parallelism***" we are referring to storing information in words of size $w = \Theta(\log n)$ bits and accessing this information using bit-level operations, such as AND, OR, XOR, shift, etc. Note that the standard ORAM and OS client-server models are completely agnostic regarding bit-level parallelism at the client; hence, our use of this technique fits squarely in the standard, classic ORAM model [9]. We provide a review of fusion trees and a "warm up" isogrammic simulation of stacks and queues in appendices, as well as omitted proofs.

2 REDUCING ORAM TO ISOGRAMMIC OS

In this section, we describe how to reduce ORAM simulation to an isogrammic OS problem, where every key used for get and put operations includes $\Theta(\log n)$ random bits. So, suppose Alice has a

RAM algorithm, $\mathcal{A}$, with memory size n, which she would like to simulate, such that the data for $\mathcal{A}$ is stored at a server, Bob.

Given the n indices in the range $[0, n-1]$ for the cells of $\mathcal{A}$'s data storage, rather than have Alice index cells using the indices as addresses for cells in an array, we have Alice access the cells of her outsourced storage stored by Bob searching down a complete binary tree, R, having the cells of her storage associated with R's leaves. This tree-based approach is actually quite common in ORAM simulations, e.g., see [7, 19]. That is, our first reduction replaces each read(i) or write(i, v) operation with a sequence of $O(\log n)$ put(k, v) and get(v) operations.

Without loss of generality, let us assume that n is a power of 2, so that every root-to-leaf path in R has the same length, namely, $\log n$. For each access for an item with index, i, we simply do a search for i in R using the standard binary-tree searching algorithm. This requires that we access exactly $\log n$ nodes, which, admittedly does not yet give rise to an isogrammic access sequence.

The modification we perform, then, is that instead of doing such accesses in a non-isogrammic fashion, let us instead convert our access sequence to be isogrammic, so that we can then make our accesses oblivious using our isogrammic OS scheme. In particular, let us apply a simple scheme inspired by an observation of Wang *et al.* [21], where we reference each node in R using a random key comprising $\Theta(\log n)$ bits. That is, for each node, u, in R, we store a random nonce, r_u, comprising $\Theta(\log n)$ bits, which is replaced with a new random nonce each time we access u. Initially, each node u in R is given an initital random nonce, r_u, which is stored at u and is chosen uniformly and independently in the range from 0 to n^c, for some constant, $c \geq 3$. Furthermore, for each node w in R that is an internal node, we store at u the random nonces for u's two children, as well as the indexing information to support our doing binary searches in R to search for a given index, i, from the root to the leaf cell for index i.

Initially, we create the nodes in R according to a standard bottom-up binary tree construction algorithm, so that when we create a node, u, we assign it to have a freshly-chosen random nonce, r_u. In addition, since we are constructing R using a bottom-up algorithm, at the time we create u, we know the random nonces for u's two children, x and y; hence, we can store these values at u. In terms of our associated isogrammic sequence, then, when we create an internal node, u, we issue a put(k, v) operation, where $k = (r_u, u)$ and $v = (I, r_x, x, r_y, y)$, and I is the indexing information that allows us to do binary searches (e.g., the smallest index for u's right child, y). If u is a leaf, then we issue a put(k, v) operation, where $k = (r_u, u)$ and $v = (i, V)$, and V is the data value that is stored at the memory cell, i, for this leaf. Note that none of these put operation can cause an error due to there already being an item present with the given key, k, since each key, (r_u, u), is unique. Moreover, each such key also comprises $O(\log n)$ random bits. We also create and maintain a global variable that stores the random nonce for the root.

Suppose, then, that we are to process a read(i) or write(i, v) operation in Alice's access sequence, converting it to a sequence of (isogrammic) put(k, v) and get(k) operations. We perform a binary search in R, for the index, i, beginning by reading the global variable storing the random nonce, r_u, for the root, u, of R and issuing an operation, get$((r_u, u))$. This get operation returns the contents of the memory record for the root, which has the form $v = (I, r_x, x, r_y, y)$, where I is the index that allows us to determine, based on i, if we should continue searching at x or at y. Suppose, without loss of generality, that we should next search at x. Before we continue our search at x, we push the record, (I, u, r_x, x, r_y, y), onto a stack stored at the server, using our isogrammic stack implementation, to keep track of our search path in R. Next, we issue an operation, get$((r_x, x))$, and we repeat the above search steps until we reach a leaf node in R. Once we have the data for the leaf node at index i, Alice performs whatever steps of her algorithm, $\mathcal{A}$, required for i.

Next, we rebuild the path that we just searched in R, giving each node in this path a new random nonce, issuing a sequence of $O(\log n)$ get(k) and put(k, v) operations. Specifically, we first give the leaf node, u, in R, at index i, a new new random nonce, r_u. Also, let V be the data value for this cell as determined by this step in Alice's algorithm, $\mathcal{A}$. Then we issue an operation put(k, v), where $k = (r_u, u)$ and $v = (i, V)$. Next, we pop the top record, (I, u, r_x, x, r_y, y), from the stack stored at the server, using our isogrammic stack implementation. Suppose, without loss of generality, that we had next gone to the node x when we first encountered this node u. Then we create a new random nonce, r_u, for the node u. Also, we can know at this point the random nonce, r_x, for the new child node, x, we just created for u, and from the record we just popped off the stack, we can recall the random nonce, r_y, for the other child, y, of u. At this point, therefore, we issue a put(k, v) operation, where $k = (r_u, u)$ and $v = (I, r_x, x, r_y, y)$. Note that this put operation cannot cause an error due to there already being an item with key k. Furthermore, note that this pattern of put and get operations reveals no information about Alice's data values or memory indices, since these operations form a data-oblivious pattern that consists of a sequence of get operations (to search down the tree R) followed by a sequence of put operations (to rebuild this search path in R using new random nonces), and each sequence contains exactly $\log n$ operations. We repeat the above computation in this way until the stack is empty, at which point we store in our global variable the information for the root. Also, note that by using this global stack at the server, we can implement each step of this algorithm using $O(1)$ client-side memory, as well as $O(1)$-sized messages. Our ORAM simulation continues in this way, so that each step in Alice's algorithm, $\mathcal{A}$, involves a root-to-leaf traveral in R followed by a leaf-to-root replacement of the nodes and nonces we just "used up." This pattern of functionality reveals no information about Alice's data values or memory addresses and each key includes a random nonce chosen uniformly and independently at random from a key space of polynomial size, since the key used for each put operation includes a random nonce comprising $\Theta(\log n)$ bits. Furthermore, note that each get operation is indexed using a key that we know was issued by a previous put operation. Thus, the resulting sequence is isogrammic.

Theorem 1. *Given a RAM algorithm, $\mathcal{A}$, with memory size, n, where n is a power of 2, we can simulate the memory accesses of $\mathcal{A}$ using an isogrammic access sequence that initially creates $O(n)$ put operations and then $O(\log n)$ get and put operations for each step of $\mathcal{A}$. Moreover, each key used in a get or put operation comprises a random nonce of $\Theta(\log n)$ bits.*

3 FUSION-TREE OS FOR SMALL SETS

The main "inner-loop" component of our algorithms is an adaptation of the binary-tree ORAM method of Damgård *et al.* [6] to fusion trees and to the OS setting. Our construction applies only for small sets, however, because fusion-tree nodes need to address other fusion-tree nodes using just $O(w^{1/2})$ bits.

Initially, we start with a fusion tree, F, with an address space of potential total size $O(n)$, which stores any initial set of elements, such that the nodes of F are stored in an array of size n whose elements are randomly shuffled, using a data-oblivious shuffling algorithm (e.g., see [1, 6, 10, 12]). In addition, we store in the same array as the nodes of F a singly linked list of Dn' "dummy" nodes, which have the same size as fusion-tree nodes and are randomly shuffled with the nodes of the fusion tree, F, where $D = 2\lceil \log n / \log w \rceil$ is the depth of F and n' is the number of initial items. We store a global header pointer for F that points to the next unused dummy node in this linked list. This initialization can occur, for example, as a part of a global initialization of multiple such fusion trees.

We have a hierarchy of arrays, $C_1, C_2, \ldots, C_\ell$, which serve as caches, where ℓ is $O(\log n)$, such that each cache, C_i, contains $\lceil n/2^i \rceil (D+1)$ real and dummy nodes stored in a shuffled order. The last cache, C_ℓ, contains $O(\log^2 n / \log w)$ nodes. Initially, these caches are empty, but as the simulation progresses, these caches will be constructed and shuffled.

Each cache, C_i, contains two types of nodes:

- A fusion tree node, v, which is a node belonging to our fusion tree, F, and is in the cache C_i due to v being accessed in a previous simulation step. If v is an external node, then it contains an item from our set, S. Otherwise, if v is an internal node, then it contains the $O(w^{1/2})$ compressed keys to identify its children, as well as $O(w^{1/2})$ pointers to the fusion tree nodes for these children. We maintain the invariant that each of these children nodes are stored either in F or in a cache, C_j, such that $j \leq i$. Because lower-indexed caches are always "older" than higher-indexed caches, we can maintain this invariant every time we build and shuffle a cache. There are at most $n/2^i$ fusion tree nodes in any cache, C_i.
- A dummy node, v. The other type of node in a cache, C_i, is a dummy node. Each dummy node has the same size as a fusion-tree node and these dummy nodes are linked to form a single simply linked list of dummy nodes in C_i, which are stored in random locations in the array for C_i, due to C_i having been randomly shuffled. We store a header pointer for each C_i (at a fixed location for C_i at the server) and any time we need to access a dummy node in such a C_i for the sake of obliviousness, we follow this header to the next available dummy node. Once we read this dummy node, we then update this header to point to the next unused dummy node in the linked list (using the pointer stored in the node we would have just accessed). Likewise, for the sake of obliviousness, even if we are accessing C_i to lookup a fusion node, we first access this global header for dummy nodes, then we lookup the fusion node, and then we do a write back to this header node (to write back its unchanged value in way that the server cannot tell is different to our writing back a changed value as if we just accessed a dummy node). The number of dummy nodes in C_i is set to make the total size of C_i be $\lceil n/2^i \rceil (D+1)$.

To perform a put(k, v) or get(k) operation, we traverse a path in F from the root to the leaf that is either holding k (in the case of a get(k) operation) or is the location where we would need to add a new item (in the case of a put(k, v) operation). In addition, to maintain the balance of F, we may need to access other nodes, but the number will always be $O(\log n / \log w)$, and we can pad this set with dummy nodes so that it is always the same number, D (for the sake of obliviousness). Let π denote the set of $D = O(\log n / \log w)$ nodes that are traversed. After performing the search for the set, π, of D nodes in F (possibly padded with dummy nodes), we store all the nodes in π in C_ℓ, and we obliviously shuffle C_ℓ. In a general step of the algorithm, we are interested in performing an search for some key, k, in F, following a search set, π, of nodes in F (plus dummy nodes as needed to make the number exactly D). At the time of this search, each node for the search set, π, is stored in one of the caches $C_1, \ldots, C_\ell$, or in the bottom level in F. The root of F, which forms the first node in π, is stored in a global location in C_ℓ (e.g., $C_\ell[0]$). So we begin by reading the root, r, of F from C_ℓ.

For each node, u, that we discover in π (starting with $u = r$), we do a comparison with our key k to determine which child of u we should read next (for our search). This node is either in one of the caches or the bottom level in F, and at this point we now know the exact location for this child, x, of u. Nevertheless, for the sake of obliviousness, we perform a read in each of C_ℓ to C_1 and also the bottom level for F. If such a lookup is for a cache (or F) that does not contain x, then we do a lookup for the next dummy node in this cache (or F), and if this cache (or F) contains x, we do the lookup for x. We repeat his sequence of lookups for each node in the search/update set, π, and for each one we add it a queue, Q, stored at the server. Once we have reached the leaf in F for k, Alice performs whatever internal computation for k (e.g., for her RAM algorithm), and we then repeat this entire lookup procedure for the next access that Alice makes. Thus, this approach fully obfuscates each get or put operation. Also, Alice performs $O(\log^2 n / \log w)$ I/Os between herself and Bob for each such access.

Each time Alice has performed $(n/2^i)D$ lookups in a cache C_i (that is, she has done $n/2^i$ accesses in her OS sequence since C_i was constructed), then she does a rebuild action. If the rebuilding is for C_1, then she re-initializes the entire structure. If it is for some C_i, for $i > 1$, then she obliviously shuffles C_i with C_{i-1}, leaving C_i empty and merging all its fusion nodes into C_{i-1}, obliviously adding a sufficient number of dummy nodes to bring the total size of C_{i-1} to be $n_{i-1} = (n/2^{i-1})(D+1)$. The details for this procedure are based on using oblivious-sorting, but the important thing to observe is that it runs with an overhead of $O(\log n_i)$, if M is $O(1)$ (e.g., see [1, 6, 10]), and an overhead of $O(1)$ if M is $O(n_i^\epsilon)$, for some constant $0 < \epsilon \leq 1/2$ (e.g., see [12]). Moreover, these bounds hold with probability $1 - 1/2^{O(w)}$, since we can do such shuffling steps by sorting n random numbers of $O(w)$ bits each.

Theorem 2. *Suppose we have a set, S, of $n \leq 2^{\sqrt{w}}$ items. Then we can perform an OS for S that has an I/O overhead that is $O(\log^3 n / \log w)$, with $O(1)$-sized client private-memory, or $O(\log^2 n / \log w)$, with $O(n^\epsilon)$-sized client private-memory, for a constant $0 < \epsilon \leq 1/2$. In either case, messages are of constant size. This simulation is statistically secure, even for non-isogrammic access sequences, and the I/O overhead bounds hold with probability $1 - 1/2^{O(w)}$.*

4 OUR ISOGRAMMIC OS ALGORITHM

In this section, we describe our isogrammic OS algorithm, which has an I/O overhead of $O(\log n \log\log n)$ or $O(\log n)$, depending on whether $M = O(1)$ or $M = O(\log^{\epsilon} n)$, for some constant $0 < \epsilon \leq 1/2$. So suppose we wish to support a data set of size $O(n)$ subject to put(k, v) and get(k) operations that come from an isogrammic access sequence. Recall that the keys used in such an access sequence have $O(\log n)$ random bits, that is, $O(w)$ random bits, since $w = \Theta(\log n)$. We use the random part of each key as the primary index for each key.

Our primary data structure for implementing an oblivious storage for isogrammic access sequences is a static complete fusion tree, H, for S, stored at the server, Bob, which has $O(n/\log^c n)$ leaves, numbered from 0 to $O(n/\log^c n)$, where $c \geq 3$ is a chosen constant. Each node, u, of H has an associated "bucket," b_u, of capacity $4L$, where $L = \Theta(\log^c n)$, which is maintained using the fusion-tree OS scheme described above in Section 3. Note that we can apply this method for each such bucket, because the number of items ever stored in any such bucket is $O(\log^c n)$. Let $W = \lceil w^{1/2} \rceil$ be the arity of our fusion tree, H, so each internal node of H has W children.

Let us further assume we have constructed H so that each root-to-leaf path has the same length, which is $O(\log n/\log w)$. Note that each leaf can be viewed as having an address of $O(\log(n/\log^c n))$ bits in a standard numbering of the leaves, so that each root-to-leaf path can be determined by the bits from this address, where going from any node to the appropriate child is performed by "reading off" the next $O(\log w)$ bits from this address, which determine the appropriate next node in the fusion tree, and then reading that next node. This allows us to perform a root-to-leaf search in H by visiting the $O(\log n/\log w)$ nodes in such a root-to-leaf path, each of which is represented using $O(1)$ words. That is, we can perform such a search using a thin client. Moreover, the tree, H, is static, so we don't need to add or remove nodes from H.

We use H as our primary "outer loop" data structure, then, for our simulation of an isogrammic access sequence. For any item, (k, v), belonging to our current set of items, the key, k, is mapped to a specific root-to-leaf path in H, which is determined by the first $O(\log(n/\log^c n))$ bits in the random part of k. During our OS simulation, we maintain the invariant that each key-value pair, (k, v), is stored in the bucket, b_u, for exactly one node, u, on the root-to-leaf path in H for the random part of k. With this in mind, let us describe our algorithms for processing get and put operations, then, using H. We describe the functioning of these operations from the perspective of the client, Alice. From the perspective of the server, Bob, the functioning of these operations will look the same. Thus, Bob cannot even distinguish whether an operation is a put or a get.

Each put(k, v) operation begins by inserting the item, (k, v), in the bucket, b_r, for the root, r, of H, using the fusion-tree OS method described in Section 3. Note that this satisfies our invariant for storing items in H; we will describe later what we do when the root bucket becomes full so as to continue satisfying our invariant. Then, for the sake of obliviousness (so Bob cannot tell whether this operation is a get or put), we uniformly and independently choose a random key, k', and traverse the root-to-leaf path in H for k', performing a search for k' in the bucket, b_u, for each node u on this path, using the fusion-tree OS method described in Section 3. Alice just "throws away" the results of these searches, but, of course, Bob doesn't know this.

For any given get(k) operation, we begin, for the sake of obliviousness, by inserting a dummy item, (k', e), in the bucket, b_r, for the root, r, of H, where e is a special "empty" value and k' is a random key, using the fusion-tree OS method described in Section 3. So as to distinguish this type of dummy item from others, we refer to each such dummy item as an ***original*** dummy item. We then traverse the root-to-leaf path, π, for (the random part of) k in H, and, for each node, u, in π, we search in the bucket, b_u, for u to see if the key-vaue pair for k is in this bucket, using the fusion-tree OS scheme described above in Section 3. By our invariant, the item, (k, v), must be stored in the bucket for one of the nodes in the path π. Note that we search in the bucket for every node in π, even after we have found and removed the key-value pair, (k, v). Because we are simulating an isogrammic access sequence, there will be one bucket with this item, but we search all the buckets for the sake of obliviousness.

An important consequence of the above methods and the fact that we are simulating an isogrammic access sequence is that each traversal of a path in H is determined by a random index that is chosen uniformly at random and is independent of every other index used to do a search in H. Thus, the server, Bob, learns nothing about Alice's access pattern from these searches. In addition, as we will see shortly, the server cannot determine where any item, (k, v), is actually stored, because the random part of the key k is only revealed when we do a get(k) operation and put operations never reveal the locations of their keys. Moreover, we maintain the fact that the server doesn't know the actual location of any item, along with our invariant, even as bucket for a node, u, becomes full and needs to have its items distributed to its children.

Periodically, so as to avoid overflowing buckets, we move items from a bucket, b_u, stored at a node u in H to u's children, in a process we call a ***flush*** operation. In particular, we flush the root node, r, every L put or get operations. We flush each internal node, u, after u has received W flushes from its parent, which each involve inserting exactly $4L/W$ real and dummy items (including new dummy items) into the bucket for u. Because of this functionality, and the fact that we are moving items based on random keys, the number of real and original dummy items in the bucket, b_u, at a time when we are flusing a node u at depth i is expected to be L, and it is at most $4L$ with high probability. Also, note that we will periodically perform flush operations across all the nodes on a given level of H at any given time when flush operations occur, which is the main reason why our I/O overhead bounds are amortized. We don't flush the leaf nodes in H, however. Instead, after every leaf, u, in H has received W flushes, we perform an oblivious compression to compress out a sufficient number of dummy items so that the number of real and dummy items in u's bucket is $4L$. Thus, the bucket for a leaf never grows to have more than $8L$ real and dummy items. If, at the time we are compressing the contents of a leaf bucket, we determine that there are more than $4L$ real items being stored in such a bucket, which, as we show, is an event that occurs with low probability, then we restart the entire OS simulation. Such an event doesn't compromise privacy, since it depends only on random keys, not Alice's data or access sequence. Thus, doing a restart just

impacts performance, but because restarts are so improbable, our I/O bounds still hold with high probability. Our method for doing a flush operation at a node, u, in H is as follows:

(1) We obliviously shuffle the real and original dummy items of b_u into an array, A, of size $4L$, stored at the server. This step will never overflow A (because of how we perform the rest of the steps in a flush operation). This step can be done using known oblivious shuffling methods (e.g., see [1, 6, 10, 12]).
(2) For each child, x_i, $i = 1, 2, \ldots, W$, of u, we create an array, A_i, of size $4L/W$.
(3) We obliviously sort the real and original dummy items from A into the arrays, $A_1, \ldots, A_\ell$, according the keys for these items, so that the item, (k, v), goes to the array A_i if the next $O(\log w)$ bits of the key k would direct a search for k to the child x_i. We perform this oblivious sorting step so that if there are fewer than $4L/W$ items destined for any array, A_i, we pad the array with (new) dummy items to bring the number of items destined to each array, A_i, to be exactly $4L/W$. However, if we determine from this oblivious sorting step that there are more than $4L/W$ real and original dummy items destined for any array, A_i, which (as we show) is an event that occurs with low probability, then we restart the entire OS simulation. Because this step is done obliviously and keys are random (hence, they never depend on Alice's data values or access pattern), even if we restart, Bob learns nothing about Alice's access sequence during this step. So, let us assume that we don't restart. This step can be done using known oblivious sorting, padding, and partitioning methods (e.g., see [1, 6, 10, 12]).
(4) For each real and dummy item (including both original and new dummy items), (k, v), in each A_i, we insert (k, v) into the bucket b_{x_i} using the fusion-tree OS method of Section 3.

The first important thing to note about a flush operation is that it is guaranteed to preserve our invariant that each item, (k, v), is stored in the bucket of a node in H on the root-to-leaf path determined by the random part of k. Moreover, because we move real and original dummy items to children nodes obliviously, in spite of our invariant, the server never knows where an item, (k, v), is stored; hence, the server can never differentiate two access sequences more than random.

Since we flush the root every L steps, and we flush every other node, u, at depth i, after it has received W flushes, and both real and original dummy items are mapped to u only if the first $i \log W$ bits of each of their random keys matches u's address, the expected number of real and original dummy items stored in the bucket for u is at most L at the time we flush u. In fact, this is a rather conservative estimate, since it assumes that none of these items were removed as a result of get operations.

Lemma 3. *The number, f, of real and original dummy items flushed from a node, u, to one of its children, x_i, is never more than $4L/W$, with high probability. Likewise, a leaf in H will never receive more than $4L$ real items, with high probability.*

Theorem 4. *We can obliviously simulate an isogrammic sequence of a polynomial number of put(k, v) and get(k) operations, for a data set of size n, with an I/O overhead of $O(\log n \log\log n)$, for constant-sized client-sized memory, or $O(\log n)$ with client-side memory of size $O(\log^\epsilon n)$, for a fixed constant $0 < \epsilon \leq 1/2$. This simulation achieves statistical security and has the claimed I/O overhead bounds with high probability.*

Putting the above pieces together, then, gives us the following:

Theorem 5. *Given a RAM algorithm, $\mathcal{A}$, with memory size, n, where n is a power of 2, we can simulate the memory accesses of $\mathcal{A}$ in an oblivious fashion that achieves statistical security, such that, with high probability, the I/O overhead is $O(\log^2 n \log\log n)$ for a constant-size client-side private memory and is $O(\log^2 n)$ for a client-side private memory of size $O(\log^\epsilon n)$, for a constant $0 < \epsilon \leq 1/2$. In either case, messages are of size $O(1)$.*

ACKNOWLEDGMENTS

This research was supported by the NSF under grant 1228639, and DARPA under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. We thank Eli Upfal and Marina Blanton for helpful discussions.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. 1983. Sorting in $c \log n$ Parallel Steps. ***Combinatorica*** 3, 1 (1983), 1–19.
[2] Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. 1999. Fusion trees can be implemented with AC^0 instructions only. ***Theoretical Computer Science*** 215, 1-2 (1999), 337–344.
[3] Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2014. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. In ***20th ASIACRYPT***. 62–81.
[4] Kai-Min Chung and Rafael Pass. 2013. A Simple ORAM. Cryptology ePrint Archive, Report 2013/243. (2013). https://eprint.iacr.org/2013/243.
[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. ***Introduction to Algorithms*** (3rd ed.). The MIT Press.
[6] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. 2011. Perfectly Secure Oblivious RAM without Random Oracles. In ***8th Theory of Cryptography Conference (TCC)***. LNCS, Vol. 6597. 144–163.
[7] Christopher W. Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. 2015. Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM. ***IACR Cryptology ePrint Archive*** 2015 (2015), 1065.
[8] Michael L. Fredman and Dan E. Willard. 1993. Surpassing the information theoretic bound with fusion trees. ***J. Comput. System Sci.*** 47, 3 (1993), 424–436.
[9] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. ***J. ACM*** 43, 3 (May 1996), 431–473.
[10] Michael T. Goodrich. 2014. Zig-zag Sort: A Simple Deterministic Data-oblivious Sorting Algorithm Running in $O(n \log n)$ Time. In ***STOC***. 684–693.
[11] Michael T. Goodrich. 2017. BIOS ORAM: Improved Privacy-Preserving Data Access for Parameterized Outsourced Storage. In ***WPES***. 41–50.
[12] Michael T. Goodrich and Michael Mitzenmacher. 2011. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In ***28th ICALP (LNCS)***, Vol. 6756. 576–587.
[13] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Practical Oblivious Storage. In ***CODASPY***. 13–24.
[14] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (in)Security of Hash-based Oblivious RAM and a New Balancing Scheme. In ***SODA***. 143–156.
[15] Michael Mitzenmacher and Eli Upfal. 2005. ***Probability and computing: Randomized algorithms and probabilistic analysis***. Cambridge Univ.
[16] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne Shuffle: Improving Oblivious Storage in the Cloud. In ***ICALP (LNCS)***, Vol. 8573. 556–567.
[17] Emil Stefanov and Elaine Shi. 2013. Multi-cloud Oblivious Storage. In ***CCS***. 247–258.
[18] E. Stefanov and E. Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In ***IEEE Symp. on Security and Privacy (SP)***. 253–267.
[19] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In ***CCS***. 299–310.
[20] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In ***CCS***. 850–861.
[21] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In ***CCS***. 215–226.

Method	Thin Client?	Statistically Secure?	B	M	I/O Overhead
Goldreich-Ostrofky [9]	Yes	No	$\Theta(1)$	$\Theta(1)$	$O(\log^3 n)$
Damgård *et al.* [6]	Yes	Yes	$\Theta(1)$	$\Theta(1)$	$O(\log^3 n)$
Goodrich-Mitzenmacher [12]	No	No	$\Theta(1)$	$\Theta(n^\epsilon)$	$O(\log n)$
Kushilevitz *et al.* [14]	Yes	No	$\Theta(1)$	$\Theta(1)$	$O(\log^2 n/\log\log n)$
Melbourne shuffle [16]	No	No	$\Theta(n^\epsilon)$	$\Theta(n^\epsilon)$	$O(1)$
Path ORAM [19]	No	Yes	$\omega(\log n)$	$\omega(B\log n)$	$O(\log^2 n/\log B)$
Supermarket ORAM [3]	No	Yes	$O(1)$	$\Theta(\text{polylog } n)$	$O(\log^2 n \log\log n)$
BIOS ORAM [11]	No	Yes	$\Omega(\log n)$	$\Omega(\log n)$	$O(\log^2 n/\log^2 B)$
Isogrammic-fusion ORAM 1	**Yes**	**Yes**	$\Theta(1)$	$\Theta(1)$	$O(\log^2 n \log\log n)$
Isogrammic-fusion ORAM 2	**Yes**	**Yes**	$\Theta(1)$	$\Theta(\log^\epsilon n)$	$O(\log^2 n)$

Table 1: Our isogrammic-fusion ORAM bounds (in boldface), compared to some of the asymptotically best previous ORAM methods. The parameter $0 < \epsilon \leq 1/2$ is a fixed constant.

A WARM UP: STACKS AND QUEUES

Wang *et al.* [21] show how to implement simple data structures, like stacks and queues in an oblivious way. We show in this section how to implement such data structures in our isogrammic framework.

Stacks. Recall that a stack maintains a set of objects organized according to a last-in, first-out protocol, where a push(x) operation adds the element x to the set and a pop() operation returns and removes the most recently pushed element. Suppose we are given a sequence of push and pop operations. We can convert this into an isogrammic access sequence as follows.

We assume that in her private storage, Alice keeps track of the size, n, and a random nonce, r, associated with the top element of the stack. Thus, we can assume that Alice will never issue a pop() operation on an empty stack. Let us also assume that our random nonce generator provides sufficiently random nonces so that it never repeats any nonces (e.g., we can enforce this by adding a counter to nonces). We initialize such a stack, Z, by issuing a put((Z, r), null) operation, where r is an initial random nonce, and Z is the name of our stack, and we have Alice store r in her private memory as well. This put() serves to create out empty stack, Z.

To process a push(x) operation, we generate a new random nonce, r', and issue a put($k, (x, r)$) operation, where $k = (Z, r')$ and r is the current top-level random nonce that Alice is storing in her private memory. That is, we use the old top-level random nonce, r, along with x, as the "value" for our put() operation and we use the new random nonce as a part of the key, k. Alice then stores r' as the new top-level nonce in her private memory.

To process a pop operation, we issue a get(k) operation, where $k = (Z, r)$ and r is the top-level nonce that Alice stores in her private memory. This get(k) returns as its value a pair, (x, r'), where x was the most recently added element and r' was the top-level nonce when x was pushed. Thus, we return x to Alice as the actual result of her pop() operation and we have Alice store the nonce, r', as the updated top-level nonce for the stack, Z.

The access sequence of put(k, v) and get(k) operations this transformation creates is isogrammic, because (1) each get(k) has a previous put(k, v) operation, (2) we never have two put(k, v) operations with the same key, k, and (3) each key, k, includes a unique random nonce. Moreover, it adds $O(1)$ put(k, v) and/or get(k) operations for each push(x) or pop operation.

Queues. Recall that a queue is a data structure that maintains a set accounding a first-in, first-out protocol, where enqueue(x) adds an element x to the set and dequeue() removes the oldest element in the set.

We assume that in her private storage, Alice keeps track of the size, n, of the queue, and two random nonces, h and t, that are associated respectively with the head and tail of her queue, Q. Initially, we choose t at random and set $h =$ null. Also, because Alice stores n, we can assume that Alice will never issue a dequeue() operation on an empty queue.

To process an enqueue(x) operation, we generate a new random tail nonce, t', and issue a put($k, (x, t')$) operation, where $k = (Q, t)$ and t is the current random tail nonce that Alice is storing in her private memory. Alice then stores t' as the new tail nonce in her private memory. If the queue, Q, was previously empty, Alice stores the old tail nonce, t, as the new head nonce, h.

To process a dequeue operation, we issue a get(k) operation, where $k = (Q, h)$ and h is the current head nonce that Alice stores in her private memory. This get(k) returns as its value a pair, (x, t'), where x is the oldest element and t' is the nonce used in the key for the next element in the queue (unless there is none). We return x to Alice as the actual result of her dequeue() operation. If the queue, Q, becomes empty, then we generate a new tail nonce, t, and set $h =$ null. Otherwise, if the queue Q is not empty, we have Alice store the nonce, t', as the updated head nonce, h, for the queue, Q.

The access sequence of put(k, v) and get(k) operations this transformation creates is isogrammic, because (1) each get(k) has a previous put(k, v) operation, (2) we never have two put(k, v) operations with the same key, k, and (3) each key, k, includes a unique random nonce. Moreover, it adds $O(1)$ put(k, v) and/or get(k) operations for each enqueue(x) or dequeue operation.

B A BRIEF REVIEW OF FUSION TREES

A ***fusion tree*** data structure [2, 8] is a B-tree (e.g., see [5]) that has a branching factor of $O(w^{1/2})$ and utilizes compressed internal nodes that can each be represented using $O(w)$ bits. That is, each internal node in a fusion tree can be represented with a single memory word of size w while nevertheless achieving a branching factor that is $O(w^{1/2})$; hence, the depth of a fusion tree is $O(\log n/\log w)$. Furthermore, we can use fusion trees in our client-server model, for thin clients, because a message block stores $O(1)$ words in this model when $B = O(1)$, which is a part of our definition of a thin client. Moreover, all the standard search and insert/delete operations can be performed on a fusion trees in $O(\log_w n) = O(\log n/\log w)$ I/Os using bit-level parallel instructions on the word-size nodes that make up the internal and external nodes in the tree. Since such bit-level parallel operations would be performed by the client, Alice, in her private memory during any ORAM simulation or oblivious storage scenario, let us not concern ourselves here with their details other than to observe that from the perspective of the server, Bob, the I/Os for searching a fusion tree would look like Alice requesting $O(\log n/\log w)$ memory cells (and we can define the tree so that each leaf node has the same depth, so the number of nodes accessed is always the same for each access or update operation).

C OMITTED PROOFS

Proof. (**Theorem 1.**) We already established the claims for performance and the result being isogrammic. For the security claim, consider a simulation of the security game mentioned in the introduction, assuming the statistical security for isogrammic OS. That is, assume Bob creates two access sequences, σ_1 and σ_2, and gives them to Alice, who then chooses one at random and simulates it. For each access to a memory index, i, in the RAM simulation for her chosen σ_j, the memory cell for i is read and written to by doing a search in R. The important observation is that this access consists of $O(\log n)$ accesses a root-to-leaf sequence of nodes of R, indexed by newly-generated independent random numbers each time. Thus, nothing is revealed to Bob about the index, i. That is, the number of accesses in Alice's simulation is the same for σ_1 and σ_2, and the sequence of keys used is completely independent of the choice of σ_1 or σ_2. Thus, Bob is not able to determine which of these sequences she chose with probability better than 1/2. □

Proof. (**Theorem 2.**) For the I/O overhead bounds, note that each search or update in F requires $O(\log^2 n/\log w)$ I/Os plus the amortized I/O overhead for rebuilding steps. Moreover, each access causes us to add $D = O(\log n/\log w)$ nodes to the top-level cache, C_ℓ, for the nodes in the search set, π. To account for rebuilding steps, note that the rebuilding of C_i occurs each $n/2^i$ steps. Thus, in the case of constant-size client-side private memory, the total I/O overhead for n searches or updates (which then cause a reinitialization), is proportional to $O(n\log^2 n/\log w)$ plus at most

$$\begin{aligned}\sum_{i=0}^{\ell} 2^i \frac{n}{2^i}(D+1)\log n &\leq n(D+1)\log^2 n\\ &= O(n\log^3 n/\log w).\end{aligned}$$

In the case of client-side private memory that is of size $O(n^\epsilon)$, the total I/O overhead for n accesses (which then cause a reinitialization), is proportional to $O(n\log^2 n/\log w)$ plus at most

$$\begin{aligned}\sum_{i=0}^{\ell} 2^i \frac{n}{2^i}(D+1) &\leq n(D+1)\log n\\ &= O(n\log^2 n/\log w).\end{aligned}$$

The security claim follows from the fact that we always access the same number of nodes with each access, for a given capacity, n, for the fusion tree, and that we never access any node a second time without caching it. Thus, in the security game, Bob is not able to distinguish between two access sequences chosen by Alice. The probability claim follows from the fact that oblivious shuffling is based on obliviously sorting items with random keys of $O(w)$ bits. □

Proof. (**Lemma 3.**) The expected value of f, which can be expressed as a sum of independent indicator random variables, is at most $L/W = d\log^{c-1/2} n$, for constants $c, d \geq 3$. Thus, by a Chernoff bound (e.g., see [15]),

$$\Pr(f \geq 4L/W) \leq e^{-L/W} \leq e^{-d\log^{c-1/2} n} \leq n^{-3\log^{3/2} n}.$$

The probability bound argument for a leaf in H is similar. The lemma follows, then, by a union bound across all nodes of H and the polynomial length of access sequences. □

Proof. (**Theorem 4.**) The height of the tree, H, is $O(\log n/\log w)$. Thus, by Theorem 2, the I/O overhead in the case when $M = O(1)$ and $B = O(1)$ is proportional to

$$\frac{\log n}{\log w}\cdot\frac{\log^3 L}{\log w},$$

which, since $L = O(\log^c n)$ and $w = \Theta(\log n)$, is $O(\log n\log\log n)$. In the case when $M = O(1)$ and $B = O(\log^\epsilon n)$, by Theorem 2, the I/O overhead is proportional to $(\log n/\log w)\cdot(\log^2 L/\log w)$, which is $O(\log n)$.

For the security claim, consider an instance of the simulation game, where Bob chooses two isogrammic access sequences, σ_1 and σ_2, of length N for a key set of size n, and gives them to Alice, who then chooses one uniformly at random and simulates it according to the isogrammic OS scheme. Each access that she does involves accessing a sequence of nodes of H determined by random keys and for each node doing a lookup in an OS scheme that is itself statistically secure, by Theorem 2. In addition, put operations add items at the top bucket and are obfuscated with data-oblivious flush operations. Therefore, Bob is not able to distinguish between σ_1 and σ_2 any better than at random. □

Proof. (**Theorem 5.**) By Theorem 1, each access in $\mathcal{A}$ gets expanded into $O(\log n)$ operations in an isogrammic access sequence, and, with high probability, each such operation has overhead $O(\log n\log\log n)$, if $M = O(1)$ and $B = O(1)$, or $O(\log n)$, if $M = O(\log^\epsilon n)$ and $B = O(1)$, by Theorem 4. The security claim follows from the security claims of Theorems 1 and 4. □