Path Oblivious Heap

Elaine Shi

1 Introduction

We show how to construct an optimal, statistically secure oblivious priority queue through a simple modification to the (non-recursive) Circuit ORAM algorithm [12]. Let N denote the maximum number of items the priority queue can store. Imagine a Circuit ORAM binary tree with N leaves, where each non-root node in the tree can hold $O(1)$ records that are either real or dummy, and the root node can hold super-logarithmically (in the security parameter) many records. Every real element in the tree carries its own position label, which ties the element to a path in the tree. Each node in the tree is additionally tagged with the minimum element in its subtree (henceforth called a subtree-min) as well as its position label. Observe that whenever a path in the tree is modified, it takes only path-length amount of work to modify the subtree-min of all nodes along the path. We can support **Insert** and **ExtractMin** queries as follows:

- **Insert**: to insert an item, assign it a random position label that ties the element to a random path in the tree. Add the item (tagged with its position label) to the root bucket. Perform eviction on two randomly selected paths (that are non-overlapping except at the root). An eviction operation tries to move real elements on the path closer to the leaf (while making sure that every element still resides on the path it is assigned to). Recalculate the subtree-mins of the two eviction paths.

- **ExtractMin**: by examining the root node's subtree-min, find out which path the minimum element resides in. Read that path, remove the minimum element from the path and save it in the CPU's local cache. Recalculate the subtree-mins of the path just read.

Other types of requests, such as **Delete**, **DecreaseKey**, and **IncreaseKey** can be supported in a similar manner. Since each request always operates on at most two paths in a binary tree, we call our construction Path Oblivious Heap.

Theorem 1 (Practical, optimal oblivious priority queue)**.** *Assume that each memory word is at least* $\log N$ *bits long and every item in the priority queue can be stored in* $O(1)$ *words. There exists a statistically secure oblivious priority queue algorithm that supports each operation in the set* $\{\textbf{Insert}, \textbf{ExtractMin}, \textbf{Delete}, \textbf{DecreaseKey}, \textbf{IncreaseKey}\}$ *in* $O(\log N + \log \frac{1}{\delta})$ *time and IO, where* N *denotes an upper bound on how many elements the priority queue can store, and* δ *denotes the failure probability per request. Further, the above holds even when the CPU has only* $O(1)$ *words of private cache.*

Path Oblivious Heap outperforms existing works both in terms of asymptotical and concrete performance. Our construction is optimal in light of the recent lower bound by Jacob et al. [6], and is amenable for implementation either in a cloud outsourcing scenario or in RAM-model multi-party computation [5, 8]. The overhead of our scheme relative to an insecure binary heap (which is the most widely adopted priority queue implementation) is minimal: binary heap requires fetching a single tree path (whose length is up to $\log N$) where each node stores a single data item. Assuming that the CPU has enough local cache to store an entire tree path, our scheme requires fetching only 2 paths per request but each node in the tree now stores 2 items (in fact, in Section 6.2, we describe a variant that requires fetching only a single path per request but each node in the tree stores 3 items). Note that if the CPU permanently stores the root bucket, exactly $\log N$ buckets need to be transmitted per path.

Last but not the least, our work immediately implies a *practical and optimal oblivious sort* algorithm which we call Path Oblivious Sort, which can sort N elements in $N(\log N + \log \frac{1}{\delta})$ time and IO with probability $1 - \delta$.

1.1 Related Work

Toft [11] constructs an oblivious priority queue with $O(\log^2 N)$ cost per request. Wang et al. [13] show a more practical construction where each request can be completed in $O(\log N(\log N + \log \frac{1}{\delta}))$ time and IO. The very recent work of Jafargholi et al. [7] showed how to construct an oblivious priority queue where each request completes in amortized $O(\log N + \log \frac{1}{\delta})$ time and $O(\log N)$ IO, but requiring $O(\log \frac{1}{\delta})$ words of CPU cache.

Jacob et al. [6] prove that any oblivious priority queue must incur $\Omega(\log N)$ IO per request even when the CPU can store $O(N^\epsilon)$ words in its private cache where $0 < \epsilon < 1$ is an arbitrary constant.

2 Definitions

We consider algorithms in the standard RAM model, where word-level addition and bitwise boolean operations can be accomplished in unit time (note that we need not assume word-level multiplication in unit time). Like in the standard RAM model, we assume that each memory word is capable of storing its own index, i.e., if the RAM's total memory size is n, then each memory word is at least $\log n$ bits long.

Throughout the paper, unless otherwise noted, we assume that the CPU has only $O(1)$ words of private cache. We measure the efficiency of a RAM algorithm using the standard notion of runtime. Note that if a RAM algorithm runs in time T, it also implies that the algorithm requires transmitting only T words between the CPU and memory — thus our overhead metric also implies bandwidth overhead.

2.1 Priority Queue

We assume that when the priority queue is first initiated, a constructor function is called which takes in a desired security parameter λ. Afterwards, the priority queue supports the following types of operations:

- $\mathsf{ref} \leftarrow$ **Insert**(k, v): insert a value v with the key k into the priority queue, and return a reference (i.e., handle) to the inserted element denoted ref.
- $(k, v, \mathsf{ref}) \leftarrow$ **FindMin**$()$: return the item whose key is the smallest (henceforth called the minimum item) in the priority queue without deleting it.
- **Delete**(ref): delete the item with the reference ref from the priority queue.
- $(k, v) \leftarrow$ **ExtractMin**$()$: remove and return the minimum item from the priority queue.

Given the above operations, we can support **DecreaseKey** and **IncreaseKey** by calling **Delete** followed by **Insert**. For simplicity we define only the above operations explicitly.

Remark 1. Our definition of a priority queue matches the standard interface provided by a popular binary heap. The recent work by Jafargholi et al. [7] in fact uses a slightly non-standard definition that is somewhat stronger than the standard binary heap: in particular, their **Delete** operation takes in only the item to be deleted but not its reference handle — for this reason, they need a k-wise independent hash function in their constrction to calculate an element's handle from the item itself. By adopting the standard definition, we can avoid the reliance on a random oracle or a pseudo-random function.

2.2 Obliviousness

Ideal functionality $\mathcal{F}_{\rm pq}$. An ideal-world priority queue, denoted $\mathcal{F}_{\rm pq}$, implements the above priority-queue interface correctly; further we assume that in the ideal world, the reference ref of an element is simply the *time* at which the element was inserted (where time is measured by the number of operations so far).

$(1-\epsilon)$-obliviousness. Let N denote an upper bound on the number of elements stored in the priority queue, and let $T \geq N$ denote an upper bound on the total number of priority queue operations. Let $\epsilon(\lambda, T)$ be a function in λ and T. We say that a priority queue algorithm PQ satisfies $(1-\epsilon)$-obliviousness iff there exists a stateful simulator Sim, such that for any conforming, even computationally unbounded adversary $\mathcal{A}$, its views in the following experiments $\mathbf{Ideal}^{\mathcal{A}}$ and $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}$ have statistical distance at most $\epsilon(\lambda, T)$:

- $\mathbf{Ideal}^{\mathcal{A}}(1^\lambda)$: $\mathcal{A}$ adaptively issues T priority-queue queries. For each query Q, $\mathcal{A}$ receives not only the output from $\mathcal{F}_{\rm pq}$ but also the outcome of $\mathsf{Sim}(1^\lambda, N, \mathsf{Q.type})$ where $\mathsf{Q.type} \in \{\texttt{insert}, \texttt{findmin}, \texttt{delete}, \texttt{extractmin}\}$ extracts the query type from the query Q.
- $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}(1^\lambda)$: $\mathcal{A}$ interacts with a challenger $\mathcal{C}$ which internally runs a copy of the real-world algorithm PQ internally instantiated with the security parameter λ. Note that $\mathcal{A}$ recognizes only ideal-world references that represent the time at which elements were inserted; however PQ recognizes the real-world references whose format is determined by the algorithm itself. Therefore, $\mathcal{C}$ acts as a man-in-the-middle between $\mathcal{A}$ and PQ, it passes queries and answers back-and-forth between $\mathcal{A}$ and PQ, translating the references from the ideal-world format to the real-world format and vice versa.

  Finally, at the end of each query, $\mathcal{C}$ also informs $\mathcal{A}$ the *access pattern* incurred by the algorithm PQ in answering this query (where by "access pattern" we mean the ordered sequence consisting of every memory address accessed).

We require that a *conforming* adversary $\mathcal{A}$ must always submit a valid ideal-world reference in any `delete` request, i.e., the reference must correspond to a time at which an insertion was made and moreover the corresponding element inserted must still exist in the priority queue.

3 Path Oblivious Heap: A Simple Oblivious Priority Queue

For simplicity, assume that we know a-priori an upper bound N on the total number of elements that the priority queue can hold, and an upper bound $T \geq N$ of the total number of priority-queue operations.

We will use the notation λ to denote an appropriate security parameter. We would like that for each operation, the priority queue's correctness fails with probability negligibly small in λ.

3.1 Data Structure

The primary data structure is a Circuit-ORAM binary tree with N leaves where N denotes an upper bound on the number of entries stored in the priority queue.

Buckets. Each tree node is also called a *bucket* since it stores an array of either real or dummy elements. Except for the root bucket which has $\omega(\log \lambda)$ capacity where λ is a security parameter, every other non-root bucket $\mathcal{B}$ in the tree can store $|\mathcal{B}| = 5$ elements and each element is either real or dummy[1].

Real and dummy elements. Each real element is of the form (k, v, ref), i.e., it stores not only contains a key-value pair denoted (k, v), but also a reference $\mathsf{ref} := (\mathsf{pos}, \tau)$ containing two pieces of metadata:

1. a random position label $\mathsf{pos} \in \{0, 1, \ldots, N - 1\}$ — this random position label is chosen uniformly at random when the element is inserted through an **Insert** operation; and
2. a timestamp τ remembering that this pair (k, v) was inserted during the τ-th operation — later τ will be included in inserted elements' references to make sure that the references are globally unique.

Henceforth we assume that a dummy element is of the form $(k = \infty, v = \bot, \mathsf{ref} = \bot)$. In particular, a dummy element has the maximum possible key.

Definition 1 (Path invariant [12])**.** *We maintain exactly the same path invariant as in Circuit ORAM [12]: a real element with the position label* pos *must reside somewhere along the path from the root to the leaf node identified by* pos.

Subtree minimum. Additionally, each bucket $\mathcal{B}$ in the tree is always tagged with its *subtree-min* $M := (k, v, (\mathsf{pos}, \tau))$, which denotes the minimum element contained in the subtree rooted at $\mathcal{B}$. Henceforth, if two elements have the same key k, we will break ties using the timestamp field τ.

3.2 Basic Operations

Bucket operations. Henceforth we assume that each bucket $\mathcal{B}$ supports two basic operations both of which can be implemented obliviously in $|\mathcal{B}| = O(1)$ cost:

1. $\mathcal{B}.\mathsf{Add}(k, v, \mathsf{ref})$: add the tuple (k, v, ref) to the bucket $\mathcal{B}$ and throw an $\mathsf{Overflow}$ exception if unsuccessful. This can be accomplished obliviously through a linear scan of the bucket, and writing the tuple to a dummy location. For obliviousness, whenever a real element is encountered during the scan, make a fake write, i.e., write the original element back. If no dummy location was found during the scan, throw an $\mathsf{Overflow}$ exception.
2. $\mathcal{B}.\mathsf{Del}(\mathsf{ref})$: delete an element with the reference ref from the bucket $\mathcal{B}$ if such an element exists. This can be accomplished obliviously through a linear scan of the bucket, writing the original element back if it does not have the reference ref; otherwise replacing it with dummy.

[1]Although a (non-root) bucket size of 5 is needed in the proofs [12], in practice we suggest choosing the bucket size to be 2 with 2 reverse-lexicographical-ordering evicitions per eviction, or choosing a bucket size of 3 with a single eviction on a random path upon **Insert** and on the delete path upon **Delete** or **ExtractMin** — see Section 6.2.

Path operations. We will need two types of path operations. Henceforth let $\mathcal{P}$ denote a path in the tree identified by the leaf node's index.

1. $\mathcal{P}.\mathsf{ReadNRm}(\mathsf{ref})$. Read every bucket on the path $\mathcal{P}$ and if an element of the reference ref exists, save its value in the CPU's local cache and remove it from the path. This can be accomplished by scanning through every bucket $\mathcal{B}$ on $\mathcal{P}$ from root to leaf and call $\mathcal{B}.\mathsf{Del}(\mathsf{ref})$.
2. $\mathcal{P}.\mathsf{Evict}()$. Eviction is an algorithm that works on a path, and tries to move real elements on the path closer to the leaf while respecting the path invariant. Circuit ORAM [12] describes an Evict algorithm that completes in time $C \cdot L$ where C denotes the number of words needed for storing each element (including its metadata ref) and L denotes the path length. We can employ their Evict algorithm directly and we refer the reader to the original Circuit ORAM paper [12] for details of the algorithm.
3. $\mathcal{P}.\mathsf{UpdateMin}()$. Whenever we operate on a path, the subtree-mins on the path need to be updated using a $\mathsf{UpdateMin}$ procedure. This procedure can be accomplished in time proportional to the path length as described below: for every bucket $\mathcal{B}$ on path $\mathcal{P}$ from leaf to root, recalculate its subtree-min by taking the minimum of 1) the minimum element of the current bucket $\mathcal{B}$; and 2) the subtree-mins of both children.

3.3 Heap Algorithms

We assume that the priority queue maintains a counter denoted τ that records the number of operations that have been performed so far, i.e., this counter τ increments upon every operation. At any point of time, if a bucket throws an $\mathsf{Overflow}$ exception when trying to add an element, the entire algorithm simply aborts with an $\mathsf{Overflow}$ exception.

FindMin(): Let $(k, v, \mathsf{ref}) :=$ the subtree-min of the root bucket $\mathcal{B}_{\text{root}}$ and return (k, v, ref).

Insert(k, v):

1. Choose a random position label $\mathsf{pos} \overset{\$}{\leftarrow} \{0, 1, \ldots, N-1\}$.
2. Call $\mathcal{B}_{\text{root}}.\mathsf{Add}(k, v, (\mathsf{pos}, \tau))$ where τ denotes the number of operations performed so far.
3. Pick two random eviction paths $\mathcal{P}$ and $\mathcal{P}'$ that are non-overlapping except at the root — the two eviction paths can be identified by the indices of the leaf nodes.
4. Call $\mathcal{P}.\mathsf{Evict}()$ and $\mathcal{P}.\mathsf{UpdateMin}()$; then call $\mathcal{P}'.\mathsf{Evict}()$ and $\mathcal{P}'.\mathsf{UpdateMin}()$.
5. Return the reference $\mathsf{ref} := (\mathsf{pos}, \tau)$.

Delete(ref) where $\mathsf{ref} := (\mathsf{pos}, \tau')$:

1. Let $\mathcal{P}$ denote the path from root to the leaf node identified by pos;
2. Call $\mathcal{P}.\mathsf{ReadNRm}(\mathsf{ref})$ and $\mathcal{P}.\mathsf{UpdateMin}()$.

ExtractMin(): Let $(k, v, \mathsf{ref}) :=$ **FindMin**() and call **Delete**(ref).

We shall prove the following theorem later in Section 4.1.

Theorem 2 (Obliviousness)**.** *The above* PQ *algorithm*[2] *satisfies* $(1-\epsilon)$*-obliviousness for* $\epsilon = T \cdot e^{-\Omega(|\mathcal{B}_{\text{root}}|)}$.

Note that for $|\mathcal{B}_{\text{root}}| = \omega(\log \lambda)$ and $T = \mathsf{poly}(\lambda)$, ϵ is a negligible function in λ.

[2]For a formal proof, assume the algorithm is augmented with a partial eviction on each delete path. See Remark 2.

4 Analysis

4.1 Obliviousness

We now prove that the above algorithm satisfies $(1-\epsilon)$-obliviousness for $\epsilon = T \cdot \exp(-\Omega(|\mathcal{B}_{\text{root}}|))$ where $|\mathcal{B}_{\text{root}}|$ denotes the capacity of the root bucket.

Since **ExtractMin** is implemented by **FindMin** and **Delete**, and our current security definition is willing to reveal the type of operations, without loss of generality in our proofs it suffices to consider only three types of requests: **FindMin**, **Insert**, and **Delete**. Observe also the following:

- **FindMin** has a deterministic access pattern;
- the access pattern of **Insert** is fully determined by the choice of the two eviction paths $\rho, \rho' \in \{0, 1, \ldots, N-1\}$; and
- the access pattern of **Delete**(ref) where ref := (pos, τ) is fully determined by the position label pos contained in the ref.

Modified notion of access pattern. For convenience, in our proof, we will use a modified notion of access pattern for our real-world algorithm:

- the access pattern of **FindMin** is $\emptyset$;
- the access pattern of **Insert** is defined by the choice of the two eviction paths $\rho, \rho' \in \{0, 1, \ldots, N-1\}$; and
- the access pattern of **Delete**(ref) where ref := (pos, τ) is defined by pos.

We now consider an algorithm PQ_∞ which is the same as the real-world algorithm PQ but with unbounded buckets. Recall that we will use the modified notion of access pattern for PQ_∞. Under this modified notion of access pattern, we first show that PQ_∞ satisfies perfect obliviousness.

Lemma 1. *Under the modified notion of access pattern,* PQ_∞ *satisfies* 1*-obliviousness.*

Proof. The simulator Sim is defind in the most obvious manner: upon receiving `findmin` output $\emptyset$; upon receiving `insert`, output two random eviction paths $\rho, \rho' \in \{0, 1, \ldots, N-1\}$ that are non-overlapping except at the root; upon receiving `delete`, output a random number from $\{0, 1, \ldots, N-1\}$.

To see why the adversary's views in $\mathbf{Ideal}^{\mathcal{A}}$ and $\mathbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$ are identically distributed, we make the following observations:

Fact 1. $\mathcal{A}$ *always receives the correct answer upon a* `findmin` *request in the experiment* $\mathbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$.

Proof. In the experiment $\mathbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$, the challenger $\mathcal{C}$ always correctly translates the ideal-world and real-world references. If so, it is not too hard to see that PQ_∞ always returns the correct minimum element upon **FindMin** — this is because our algorithm guarantees that all nodes' subtree-min are correctly maintained at the end of each request. □

Therefore, to prove Lemma 1, it suffices to show that the simulated access patterns output by Sim are identically distributed by the access patterns of PQ_∞. Notice that upon an `insert` query, no matter in $\mathbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$ or $\mathbf{Ideal}^{\mathcal{A}}$, the adversary always sees two fresh random numbers from $\{0, 1, \ldots, N-1\}$ even when conditioned its view so far in the experiment. Now consider a `delete` query and suppose that $\mathcal{A}$ wants to delete an element inserted at time τ:

- In the experiment $\mathbf{Real}^{\mathsf{PQ}_\infty,\mathcal{A}}$, the adversary $\mathcal{A}$ sees the random path chosen for the element inserted at time τ, and this random choice was made earlier at time τ;

- In the experiment $\mathbf{Ideal}^{\mathcal{A}}$, the adversary $\mathcal{A}$ sees a random path chosen right now by Sim.

It is not hard to see that even in the experiment $\mathbf{Real}^{\mathsf{PQ}_\infty,\mathcal{A}}$, upon a `delete` query and conditioned on the adversary's view so far, the random path revealed is uniform at random from the range $\{0, 1, \ldots, N-1\}$ — specifically, notice that the adversary's view so far does not depend on the random choice made earlier at time τ. □

Lemma 2 (Probability of $\mathsf{Overflow}$)**.** *For any conforming adversary $\mathcal{A}$, in the real-world experiment $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}$, an* $\mathsf{Overflow}$ *exception is encountered with probability at most $T \cdot \exp(-\Omega(|\mathcal{B}_{\text{root}}|))$.*

Proof. If there is a conforming adversary $\mathcal{A}$ that can cause $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}$ to encounter $\mathsf{Overflow}$ with probability ν, we can easily construct an adversary $\mathcal{B}$ that cause the (non-recursive) $\mathsf{CircuitORAM}$ to encounter $\mathsf{Overflow}$ with probability ν too. Specifically,

- $\mathcal{B}$ invokes a non-recursive $\mathsf{CircuitORAM}$ parametrized also with N and λ — note that in the binary tree of $\mathsf{CircuitORAM}$ every bucket has the same capacity as the corresponding node in our PQ algorithm.

- Furthermore, $\mathcal{B}$ internally maintains a correct priority queue, and upon any `findmin` query from $\mathcal{A}$ it always returns the correct answer to $\mathcal{A}$.

- whenever $\mathcal{A}$ submits `findmin`, $\mathcal{B}$ returns the $\emptyset$ access patterns to $\mathcal{A}$;

- whenever $\mathcal{A}$ submits an `insert` request, $\mathcal{B}$ may arbitrarily choose this element's logical address addr to be any fresh address that is different from the address of any inserted elements so far; now $\mathcal{B}$ submits a $(\mathsf{Insert}, \mathsf{addr}, *)$ request to its own challenger where $*$ denotes an arbitrary payload string which we do not care about. As a result, $\mathcal{B}$ obtains the $\mathsf{CircuitORAM}$'s access patterns that are fully determined by two eviction paths $\mathcal{P}$ and $\mathcal{P}'$, $\mathcal{B}$ returns these paths' identifiers to $\mathcal{A}$.

- whenever $\mathcal{A}$ issues a `delete` request on an element inserted at time τ, $\mathcal{B}$ finds out the correct logical address addr of this element and submits a $(\mathsf{ReadNRm}, \mathsf{addr})$ request to its challenger. As a result, $\mathcal{B}$ obtains the $\mathsf{CircuitORAM}$'s access patterns that are fully determined by a read path, $\mathcal{B}$ returns this read path's identifier to $\mathcal{A}$.

Now, the experiment $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}$ is fully determined by the random coins $\vec{\psi}$ consumed by $\mathcal{A}$ and the random coins $\vec{\mu}$ consumed by PQ. If the execution of $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}$ determined by $(\vec{\psi}, \vec{\mu})$ encounters overflow, the execution of the above experiment determined also by $(\vec{\psi}, \vec{\mu})$ — where $\vec{\psi}$ denotes $\mathcal{A}$'s random coins and $\vec{\mu}$ denotes $\mathsf{CircuitORAM}$'s random coins — will also encounter overflow.

Thus, the lemma follows directly from Theorem 5 of Section A. □

Remark 2 (A note about partial eviction)**.** Due to Remark 4 of Section A, we in fact need to augment our PQ algorithm slightly for the above lemma to hold: for every **Delete** operation, we need to perform a "partial eviction" operation on the path from which we deleted an element. We refer the reader to the Circuit ORAM [12] work for full details of the partial eviction. In a practical implementation, just like the authors of Circuit ORAM recommended, omitting this partial eviction empirically improves the practical performance by a small constant factor (and also simplifies the implementation).

Proof of Theorem 2. We now prove Theorem 2. From Lemma 1 and Lemma 2 we have that PQ satisfies $(1-\epsilon)$-obliviousness under the modified notion of access patterns. It is easy to see that this implies that PQ satisfies $(1-\epsilon)$-obliviousness under the original notion of access patterns too.

4.2 Asymptotical Efficiency

Suppose that each item of the form (k, v) in the priority queue can be represented by D bits. Each element in our binary-tree data structure has $\Theta(\log N + \log T + D)$ bits. Assume that the RAM's word size is large enough to hold such an element (if it takes C memory words to store this information, we can simply multiply the overhead by C). We can now analyze the scheme's efficiency.

The total space is upper bounded by $O(N)$. Even when the CPU can locally store only $O(1)$ words, each priority-queue request can be completed in $O(\log N + |\mathcal{B}_{\text{root}}|)$ time. Specifically, assuming that T and N are polynomially bounded by λ, and the root bucket size $|\mathcal{B}_{\text{root}}|$ is an arbitrarily small super-logarithmic function in λ, then our PQ scheme will satisfy $(1 - \mathsf{negl}(\lambda))$-obliviousness due to Theorem 2 later in Section 4.1.

Theorem 3 (Oblivious priority queue). *Let $D := |k| + |v|$ be the number of bits needed to represent an item of the priority queue; let w be the bit-width of each memory word, and let $C := (D + \log T)/w$. Then, for any $0 < \delta < 1/T$, there exists a priority queue algorithm that satisfies $(1-T\delta)$-obliviousness; and moreover it completes each* **FindMin** *request in $O(C)$ time, and each* **Delete**, **Insert**, *and* **ExtractMin** *request in $O(C \cdot (\log N + \log \frac{1}{\delta}))$ time.*

So far we have assumed that the the CPU has only $O(1)$ words of private cache. If, however, the CPU's local cache is just a little larger — specifically, as large as the size of a path — then every priority-queue request can be served in a *single round-trip* between the CPU and memory.

The case of unknown T. So far we have assumed that we know the maximum number of requests (denoted T) a-priori. It is easy to remove this assumption and construct an oblivious priority queue that only needs to know N a-priori but not T. Basically, the caller constructs PQ by declaring N upfront and letting $T = 3N$. After making N queries, the caller can read out all items stored in PQ by repeatedly calling **ExtractMin**[3] and inserting these items into a new instance of PQ, also parametrized with N and $T = 3N$. Note that even if the existing PQ may in fact be storing fewer than N items at the time of destruction, the caller always needs to make N **ExtractMin** queries to hide the number of items. This goes on, and each PQ instance needs to support at most $3N$ queries: N **Insert** queries at initialization, followed by N arbitrary queries during its life cycle, followed by N **ExtractMin** queries at destruction. The cost of this periodic rebuild can be amortized to each request so far. Thus we have the following corollary:

Corollary 1 (The case of unknown T). *Let N be an a-priori known upper bound on the number of items the priority queue can store. Let T be the actual number of requests which may not be known upfront. Let $D := |k| + |v|$ be the number of bits needed to represent an item of the priority queue; let w be the bit-width of each memory word, and let $C := (D + \log N)/w$.*

Then, for any $0 < \delta < 1/T$, there exists a priority queue algorithm that satisfies $(1 - T\delta)$-obliviousness; and moreover it completes each **FindMin** *request in $O(C)$ time, and each* **Delete**, **Insert**, *and* **ExtractMin** *request in $O(C \cdot (\log N + \log \frac{1}{\delta}))$ time.*

[3] A practical optimization is to read every slot in the binary-tree data structure to extract all elements stored.

Finally, we can also support the case of unknown N at the price of a slight leakage of information: we can initialize a PQ instance with N capacity (e.g., $N = 16$); whenever we detect that the total number of items exceeds N, we read out every item in the current instance and insert them into a new instance initialized with $2N$ capacity, and so on. This approach leaks when the total number of items exceeds the next power of 2. In practice, we also recommend employing a differentially private all-prefix-sum algorithm [1–3] to obfuscate exactly when the total number of items exceed the next power of 2.

5 Path Oblivious Sort: Practical and Optimal Oblivious Sort

Given m elements each of the form (k, v), we can obliviously sort them as follows: 1) initiate an oblivious PQ parametrized with a security parameter λ and the space-time parameters $N = T = m$; 2) insert each element sequentially into an oblivious priority queue by calling the **Insert** algorithm; and 3) call **ExtractMin**() a total of m times and write down the outputs one by one. If the oblivious PQ satisfies ϵ-obliviousness, then the resulting oblivious sorting algorithm correctly sorts m numbers with $1 - \epsilon$ probability.

Theorem 4 (Optimal oblivious sorting)**.** *Let $D := |k| + |v|$ be the number of bits needed to represent an item of the priority queue; let w be the bit-width of each memory word, and let $C := (D + \log T)/w$. Then, for any $0 < \delta < 1$, there exists an oblivious sort algorithm that can correctly sort m integers in $O(Cm(\log m + \log \frac{1}{\delta}))$ time with probability $1 - \delta$.*

6 Practical Considerations

6.1 Hiding the Request Type

So far our algorithm hides the relative ordering of the items inserted into the priority queue but does not hide the type of the requests.

It is easy to additionally hide the type of requests too. Basically, for every request, say **ExtractMin**, we can make dummy accesses that emulate the access patterns of all other requests, including **Insert**, **FindMin**, and **Delete**, and run the real algorithm **ExtractMin**. We do the same for every type of request — note that we need to run the algorithms, fake or real, in a fixed order.

With this modification, the cost of **Insert**, **Delete**, and **ExtractMin** blow up by only a constant factor; and **FindMin** now must incur logarithmic cost too (instead of constant).

Remark 3. As a practical optimization, one can take the *smallest superset* of the access patterns of all requests, and incur only this superset access patterns for every request; doing either useful or dummy work with each physical access depending on what the request is.

6.2 Practical Performance Optimizations

So far, we have built our Path Oblivious Heap from Circuit ORAM [12] which allows us to achieve asymptotic optimality.

In a cloud outsourcing scenario, one may consider employing Path ORAM [9,10] which is almost identical to Circuit ORAM except that it uses a slightly different Evict algorithm. Although Path ORAM is asymptotically slightly worse than Circuit ORAM, in some practical scenarios such as cloud computing, it achieves better practical performance.

If we instantiate our Path Oblivious Heap using Path ORAM's eviction algorithm, we get the following asymptotical performance. Henceforth we assume that each memory word can store an item in the priority queue as well as its own index, and moreover let the path length $L := \log N + \log \frac{1}{\delta}$ where δ represents the failure probability per request:

- If the CPU can locally store L elements, then each priority queue request can be completed in $O(L \log L)$ runtime and $O(L)$ number of IOs (i.e., number of words transferred between CPU and memory);
- If the CPU can locally store $O(1)$ words, then each priority queue request can be completed in $O(L \log L)$ runtime and IOs.

When instantiated with Path ORAM, our asymptotical performance in fact matches the recent work by Jafargholi et al. [7]. However, our scheme is much more amenable to practical implementation than Jafargholi et al. [7].

In practice, we also recommend the following strategy for selecting eviction paths and parametrizing non-root nodes' bucket sizes:

1. Instead of picking two eviction paths at random, pick two eviction paths deterministically using the *reverse-lexicographical ordering* strategy first suggested by Gentry et al. [4]. Empirial evaluations in the past have shown that the bucket size can be set to as small as 2 if such a strategy is adopted [12] (using either Circuit ORAM or Path ORAM's eviction algorithm).
2. Alternatively, perform eviction on a random path upon **Insert** and perform eviction on the delete path upon **Delete** or **ExtractMin** — in this case, if we use Path ORAM's eviction algorithm, the bucket size can be as small as 3 [10].

Finally, if the reader wishes to implement Path Oblivious Heap on top of a multi-party computation backend, we recommend instantiating it with Circuit ORAM for best concrete performance [12].

Acknowledgments

The author would like to thank Kai-Min Chung for insightful technical discussions. This work is supported in part by NSF CNS-1453634, an ONR YIP award, and a Packard Fellowship.

References

[1] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming: Part II*, ICALP'10, pages 405–417, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. *ACM Transactions of Information and System Security (TISSEC)*, 14(3):26, 2011.

[3] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 715–724, New York, NY, USA, 2010. ACM.

[4] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[5] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[6] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2439–2447, 2019.

[7] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues and offline oblivious RAM. Cryptology ePrint Archive, Report 2019/237, 2019. `https://eprint.iacr.org/2019/237`.

[8] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*, 2015.

[9] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.

[10] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[11] Tomas Toft. Secure data structures based on multi-party computation. In *PODC*, pages 291–292, 2011.

[12] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.

[13] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.

A Background on Non-Recursive Circuit ORAM

Data structure. As described by Wang, Chan, and Shi [12], a non-recursive Circuit ORAM, parametrized by a security parameter λ and a capacity parameter N, is formed as a binary tree with N leafs, where each node in the tree is called a bucket. The root bucket $\mathcal{B}_{\text{root}}$ can $\omega(\log \lambda)$ store blocks whereas every internal bucket $\mathcal{B}$ can store 5 blocks. All blocks stored in the tree are either *real* or *dummy*:

- A real block is of the form $(\mathsf{addr}, \mathsf{data}, \mathsf{pos})$ where addr dentoes $\mathsf{addr} \in \{0, 1, \ldots N-1\}$ denotes the logical address of the block; data denotes an arbitrary payload string — we assume that the pair $(\mathsf{addr}, \mathsf{data})$ can fit in a single block; and $\mathsf{pos} \in \{0, 1, \ldots, N-1\}$ denotes the position label for the block.
- A dummy block is of the form $(\perp, \perp, \perp)$.

Algorithms. Henceforth whenever we say Circuit ORAM, we mean specifically the non-recursive version. Circuit ORAM supports the following operations:

$\mathsf{pos} \leftarrow$ **Insert**$(\mathsf{addr}, \mathsf{data})$:

1. Pick a random position label $\mathsf{pos} \in \{0, 1, \ldots, N-1\}$ and call $\mathcal{B}_{\text{root}}.\mathsf{Add}(\mathsf{addr}, \mathsf{data}, \mathsf{pos})$; where $\mathcal{B}_{\text{root}}.\mathsf{Add}$ can be implemented identically as described in Section 3. Note that if the bucket $\mathcal{B}$ is already fully occupied and there is no room to successfully perform the addition, an Overflow exception is thrown.
2. Pick two random eviction paths $\mathcal{P}$ and $\mathcal{P}'$ that are non-overlapping except at the root bucket — note that the paths may be identified by the indices of the leaf nodes. Now, call $\mathcal{P}.\mathsf{Evict}()$ and $\mathcal{P}'.\mathsf{Evict}()$. The details of the Evict algorithm is not too important for this paper — we only need to know that the Evict algorithm has deterministic access patterns and completes in $O(|\mathcal{B}_{\text{root}}| + \log N)$ time.
3. Return pos;

$\mathsf{data} \leftarrow$ **ReadNRm**$(\mathsf{addr}, \mathsf{pos})$:

Assume: pos must be the label returned by **Insert** when the block at address addr was added; moreover this block must not have been removed since it was added.

1. For each bucket $\mathcal{B}$ from the root to the leaf identified by pos: sequentially scan through the bucket $\mathcal{B}$:
   - when addr is encountered, remember the data field in the client's local cache and and replace the block with dummy block;
   - else write the original block back for obliviousness.
2. Return data.

Stochastic bounds for Circuit ORAM. Consider an adversary $\mathcal{A}$ that interacts with a challenger denoted $\mathcal{C}$ and adaptively submits a sequence of requests either of the form (Insert, addr, data) or of the form (ReadNRm, addr). It is guaranteed that for a ReadNRm request, a conforming adversary $\mathcal{A}$ always supplies an addr that has been added (and has not been removed since its addition).

- Whenever $\mathcal{C}$ receives a request of the form (Insert, addr, data) from $\mathcal{A}$, it simply calls Circuit ORAM's $\mathsf{pos} \leftarrow$ **Insert**$(\mathsf{addr}, \mathsf{pos}, \mathsf{data})$ algorithm and records the pos that is returned.

- Whenever $\mathcal{C}$ receives a request of the form (ReadNRm, addr) from $\mathcal{A}$, it finds out the correct position label pos for addr and calls Circuit ORAM's $\mathsf{data} \leftarrow$ **ReadNRm**$(\mathsf{addr}, \mathsf{pos})$ algorithm and returns data to $\mathcal{A}$.

- No matter which query $\mathcal{C}$, at the end of the query $\mathcal{C}$ returns to $\mathcal{A}$ the access patterns made by the Circuit ORAM algorithm.

Theorem 5 (Circuit ORAM's overflow probability [12]). *For any conforming adversary $\mathcal{A}$, the probability that the above experiment encounters* Overflow *is upper bounded by $T \cdot \exp(-\Omega(|\mathcal{B}_{\text{root}}|))$ where $|\mathcal{B}_{\text{root}}|$ denotes the capacity of the root bucket.*

Remark 4. Note that the Circuit ORAM work [12] in fact suggested two variants of the algorithm: a provable version and another slightly simplified variant recommended for practical implementation. The algorithm described so far is in fact the practical variant. To obtain the above Theorem 5, we need to add a slight tweak to the algorithm, that is, during the **ReadNRm** algorithm, we need to perform a "partial eviction" on the read path. We refer the readers to Circuit ORAM [12] for a full explanation.