Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue*

Elaine Shi

November 10, 2019

Abstract

We propose Path Oblivious Heap, an extremely simple, practical, and optimal oblivious priority queue. Our construction also implies a practical and optimal oblivious sorting algorithm which we call Path Oblivious Sort. Not only are our algorithms asymptotically optimal, we show that their practical performance is only a small constant factor worse than insecure baselines. More specificially, assuming roughly logarithmic client private storage, Path Oblivious Heap consumes $2\times$ to $7\times$ more bandwidth than the ordinary insecure binary heap; and Path Oblivious Sort consumes $4.5\times$ to $6\times$ more bandwidth than the insecure Merge Sort. We show that these performance results improve existing works by 1-2 orders of magnitude. Finally, we evaluate our algorithm for a multi-party computation scenario and show $7\times$ to $8\times$ reduction in the number of symmetric encryptions relative to the state of the art.

1 Introduction

We show how to construct a statistically secure oblivious priority queue through a simple modification to the (non-recursive) Path ORAM algorithm [39]. Since our construction is compellingly simple, we begin the paper by presenting the construction informally. Let N denote the maximum number of items the priority queue can store. Imagine a 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. For readers familiar with Path ORAM [39], so far the data structure is the same as a Path ORAM binary tree. Our key insight is the following (which, in hindsight, turns out to be surprisingly simple): we additionally tag each node in the tree 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 — to do this we only need to examine this path and all sibling nodes to 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.

*Dedicated to the memory of Emil Stefanov (1987-2014), to all those fun times, the many days and nights that we worked together.

- **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. Perform eviction on the path just read, and moreover, recalculate the subtree-mins of the path just read.

We stress that due to subtree-min labels that our algorithm maintains throughout, *there is no need for a recursive position map* which was necessary in previous tree-based ORAMs [12, 39, 42] and oblivious data structures [44] — one can easily find out the path of the minimum element by examining the root's subtree-min label.

Other types of requests, such as **Delete**, **DecreaseKey**, and **IncreaseKey** can be supported in a similar manner. We can additionally hide the request type by emulating the super-set of access patterns for all requests. Since each request always operates on at most two paths in a binary tree, we call our construction Path Oblivious Heap.

Theorem 1 (Path Oblivious Heap)**.** *Assume that each memory word is at least* $\log N$ *bits and every item in the priority queue can be stored in* $O(1)$ *words. Further, suppose that the CPU has* $O(\log N + \log \frac{1}{\delta})$ *words of private cache. There exists a statistically secure oblivious priority queue algorithm that supports each operation in the set* {**Insert**, **ExtractMin**, **Delete**, **DecreaseKey**, **IncreaseKey**} *requiring only* $O(\log N)$ *words to be transmitted between the CPU and the memory per operation, where* N *denotes the maximum number of elements the priority queue can store, and* δ *denotes the failure probability per request.*

Readers familiar with tree-based ORAM constructions might also recall that Circuit ORAM [42] is a further improvement to Path ORAM [39]: in Circuit ORAM, the CPU needs only $O(1)$ words of private cache whereas in Path ORAM, the CPU needs $O(\log N + \log \frac{1}{\delta})$ words of private cache. The only difference between Circuit ORAM and Path ORAM is that the two adopt a different path-eviction algorithm. If we instantiated the above Path Oblivious Heap algorithm but now with Circuit ORAM's eviction algorithm, we obtain the following corollary which further reduces the CPU's private cache to $O(1)$:

Corollary 1 (Path Oblivious Heap: Circuit Variant)**.** *Same as Theorem 1, but now the CPU needs only* $O(1)$ *words of private cache and each request consumes* $O(\log N + \log \frac{1}{\delta})$ *bandwidth.*

Henceforth to distinguish the two variants instantiated with Path ORAM and Circuit ORAM's eviction algorithms respectively, we call them the Path-variant and the Circuit-variant respectively[1] From a theoretical perspective, the Circuit-variant is a strict improvement of the Path-variant — if the CPU had $O(\log N + \log \frac{1}{\delta})$ private cache in the Circuit-variant, the bandwidth needed per request would also be $O(\log N)$ just like the Path-variant.

Optimality. Path Oblivious Heap outperforms existing works both in asymptotic and concrete performance, and moreover achieves optimality in light of the recent lower bound by Jacob et al. [25]. We recommend the Path-variant for a *cloud outsourcing* scenario, and the Circuit-variant for RAM-model *multi-party computation* [24,31] — recall that these are the two primary application scenarios for oblivious algorithms.

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 of length $\log N$ (as well as all sibling nodes) 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.

[1]The title of our paper calls both the Path-variant and the Circuit-variant "Path Oblivious Heap" generically since in both variants, every request operates on $O(1)$ number of tree-paths.

Reference implementation. A reference implementation (created by Daniel Rong and Harjasleen Malvai) is available at `https://github.com/obliviousram/PathOHeap`.

1.1 Applications

Oblivious streaming sampler and applications in distributed differential privacy. In Section 5.2, we show how to leverage our Path Oblivious Heap algorithm to design an efficient oblivious streaming sampler, i.e., an algorithm that randomly samples k elements from an incoming stream of a-priori unknown length (possibly much larger than k), consuming at most $O(k)$ memory; moreover, the algorithm's access patterns do not reveal which elements in the stream have been sampled. We describe how this oblivious sampler can be a key building block in designing an intereseting class of distributed differential privacy mechanisms motivated by practical scenarios that companies such as Google and Facebook care about.

Practical oblivious sort. 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 $1 - \delta$. Our new Oblivious Sorting algorithm can replace bitonic sort [6] which is the *de facto* choice for implementation today despite being non-optimal and consuming $O(n \log^2 n)$ cost [31, 33, 34] — see Section 1.2 for more discussions.

Evaluation results suggest that in a cloud-outsourcing setting, our oblivious heap and oblivious sorting algorithms consume only a small constant factor more bandwidth relative to insecure baselines. Specifically, Path Oblivious Heap consumes only $2\times$ to $7\times$ more bandwidth than the ordinary insecure binary heap; and Path Oblivious Sort consumes only $4.5\times$ to $6\times$ more bandwidth than the insecure Merge Sort.

1.2 Related Work

Oblivious RAMs. Oblivious RAM (ORAM) was first proposed by Goldreich and Ostrovsky [20, 21] in a ground-breaking work. They show that any RAM algorithm can be compiled to an oblivious counterpart whose memory access patterns leak no information, and somewhat surprisingly, such oblivious compilation incurs only poly-logarithmic (multiplicative) blowup relative to the insecure baseline. Subsequent works have improved Goldreich and Ostrovsky's construction; to date the best known ORAM algorithm achieves $O(\log N)$ blowup where N is the size of memory consumed by the original RAM [4]. On the other hand it has been shown that logarithmic overhead is necessary for ORAMs [20, 21, 28].

Oblivious data structures. Since ORAM is a generic technique that compiles *any* algorithm or data structure to an oblivious form, a naïve way to obtain an oblivious priority queue is to apply an ORAM compiler to a standard insecure priority queue algorithm such as the widely-adopted binary heap. Although in theory this achieves logarithmic slowdown w.r.t. the insecure binary heap [4], the known theoretical optimal construction, OptORAMa [4], is completely impractical.

A couple recent works have shown how to construct *practical* oblivious priority queues that enjoy logarithmic slowdown w.r.t. to the insecure binary heap. Specifically, Toft [40] constructs an oblivious priority queue with $O(\log^2 N)$ cost per request (*c.f.* binary heap requires $O(\log N)$ cost per request). Wang et al. [44] show a more practical construction where each request can be completed in $O(\log N(\log N + \log \frac{1}{\delta}))$ time and bandwidth (measured in words transmitted between the CPU and memory).

An interesting question is whether we can outperform generic ORAM simulation for constructing an oblivious priority queue. Note that the ORAM lower bound applies only to a *generic* oblivious compiler, but not for any specific algorithm such as priority queue. The very recent work of Jafargholi et al. [26] was first to answer this question affirmatively: they showed how to construct an oblivious priority queue where each request completes in amortized $O(\log N)$ bandwidth, but requiring $O(\log \frac{1}{\delta})$ words of CPU cache. In another very recent work, Jacob et al. [25] prove that any oblivious priority queue must incur $\Omega(\log N)$ bandwidth per request even when the CPU can store $O(N^\epsilon)$ words in its private cache where $0 < \epsilon < 1$ is an arbitrary constant.

Comparison with Jafargholi et al. [26]. Since the very recent work by Jafargholi et al. [26] is the most closely related, we now provide more detailed comparison with them. Theoretically, on a standard word-RAM, our result is asymptotically better than theirs since we require only $O(1)$ words of CPU private registers where Jafargholi et al. [26] requires super-logarithmic in the security parameter to get negligible failure probability (also in the security parameter). In practical cloud-like scenarios, our evaluation results in Section 6 suggest that our algorithm is *at least an order of magnitude faster than theirs*. Furthermore, our overhead notion is measured in terms of the *worst-case* cost per request where they use an *amortized* notion — in their scheme, a request can in the worst case incur linear cost (although this happens infrequently).

Nonetheless Jafargholi et al. [26]'s algorithm is theoretically interesting in the following senses. First, their algorithm has asymptotically better IO performance when the memory's native block size is much larger than a single entry stored in the priority queue. Specifically, imagine that one must pay the cost of a single big block even when retrieving, say, one entry, which is much smaller than the memory block. In this case, Jafargholi et al. [26] achieves an additional factor of χ speedup relative to ours assuming each memory block can store χ entries. While this metric is theoretically interesting, it has limited relevance to the two primary applications for oblivious priority queue and oblivious sorting: 1) cloud outsourcing and 2) secure multi-party computation:

- In a practical cloud-outsourcing scenario, typically the main bottleneck is the client-server bandwidth and not the server's disk IO. Observe that the client-server transmission is not bound to any "native block" constraint For example, our algorithm retrieves a couple tree paths at a time, and one could combine these paths into one or more network packets (even though the buckets fetched are not necessarily contiguous in physical storage). For this reason, almost all prior ORAM schemes designed for the client-server setting [23, 35, 36, 38, 45] focus on measuring the bandwidth consumption.
- In multi-party computation [24, 42], similarly, there is no "native block size" constraint. In this case, a word-RAM model with $O(1)$ CPU registers is the most appropriate as prior works have compellingly shown [42].

Another theoretically interesting aspect of their construction is that it can be instantiated with a k-wise independent hash function, and thus fewer random bits need to be consumed per request. In comparison, our construction consumes logarithmically many random bits per request.

Oblivious sorting. Theoretically speaking, n items can be obliviously sorted in $O(n \log n)$ time using sorting networks such as AKS [3] and Zigzag sort [22]. These constructions are optimal: due to recent lower bounds [17, 30], we know that any oblivious sorting scheme must incur $\Omega(n \log n)$ time on a word-RAM, either assuming that the algorithm treats each element as "indivisible" [30] or assuming that the famous Li-Li network coding conjecture [29] is true [17].

Unfortunately, known optimal sorting networks [3, 22] rely on expander graphs and thus suffer from enormous constants, making them completely impractical. Almost all known implementations of oblivious sorting, either for cloud-outsourcing [13, 38, 46] or for multi-party computation [31, 33, 34], instead adopt the asymptotically worse bitonic sort [6] which costs $O(n \log^2 n)$. Given our evaluation results, Path Oblivious Sort should now become the scheme of choice in either a cloud-outsourcing or multi-party computation setting for almost all parameters we care about.

Offline ORAM. Boyle and Naor [7] show that given a RAM-model oblivious sorting algorithm that consumes $T(n)$ time to sort n elements, one can construct an offline ORAM with $T(N)/N$ blowup. In comparison with a standard (online) ORAM, an offline ORAM must see all the requests upfront. Thus, just like Jafargholi et al. [26], *our work also implies a statistically secure offline ORAM with $O(\log N)$ blowup.* Jafargholi et al. [26] achieves this result too but assuming that the CPU can cache super-logarithmic (in the security parameter) number of words. We remove this extra assumption and require only $O(1)$ CPU-registers.

We stress that although logarithmic-overhead online ORAM is also possible, the only known construction [4] for the online setting requires computational assumptions. Our scheme is *statistically* secure, and the *statistically* secure logarithmic-overhead *offline* ORAM result is of incomparable nature to known optimal online ORAM [4].

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. A RAM program's runtime is the number of CPU steps it takes to complete the computation, its bandwidth or IO cost is the number of *words* transmitted between the memory and the CPU. In cases where the CPU may store super-constant number of words and compute on them in a single time step, the bandwidth of a RAM may be larger than the runtime.

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 — below we define the "programming interface" in the same way as that of an insecure binary heap:

- $\mathsf{ref} \leftarrow \textbf{Insert}(k, v)$: 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.
- $(k, v, \mathsf{ref}) \leftarrow \textbf{FindMin}()$: return the item whose key is the smallest (henceforth called the minimum item) in the priority queue without deleting it.
- $\textbf{Delete}(\mathsf{ref})$: delete the item with the reference ref from the priority queue.
- $(k, v) \leftarrow \textbf{ExtractMin}()$: 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 — in a standard binary heap, to call **Delete** and **DecreaseKey** operations, the caller also needs to supply a reference handle to the object being deleted or modified. The recent work by Jafargholi et al. [26] in fact adopts 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 avoid the reliance on a k-wise independent hash.

2.2 Oblivious Simulation of Priority Queue

We first define a weaker notion of security (called type-revealing security), where, informally speaking, only the request types are revealed to the adversary but not the contents of the requests. For example, in our oblivious sorting application later, this weaker notion is sufficient. It is not difficult to strengthen our definition and construction to additionally hide the request types too incurring only minimal additional cost — we shall later elaborate on type-hiding security in Appendix D.

We define obliviousness through a simulation-based approach: we would like that the adversary's observations in the real-world be statistically close to an ideal-world in which access patterns are simulated by a simulator that observes only the request types but not the contents of the requests. In the ideal world, the reference handles returned to the adversary contain only the time at which the element was inserted where time is measured in the number of requests so far. In the real world, we may imagine that all additional fields of the reference handle (besides the time) are encrypted or secret-shared so the adversary cannot observe the actual contents of the handle in the same way that actual data contents are encrypted or secret shared and unobservable to the real-world adversary (see also Remark 3).

Ideal functionality $\mathcal{F}_{\text{pq}}$. An ideal-world priority queue, henceforth denoted $\mathcal{F}_{\text{pq}}$, implements the above priority-queue interface correctly. As mentioned, 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)[2]

Oblivious simulation. Let N denote an upper bound on the number of elements stored in the priority queue. 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. Let $\mathsf{PQ}(1^\lambda, N)$ denote a priority queue algorithm whose execution is parametrized with a security parameter λ and the maximum capacity N. Henceforth we often write PQ for short omitting the default parameters. We say that PQ is a $(1-\epsilon)$-oblivious simulation of $\mathcal{F}_{\text{pq}}$ iff there exists a stateful simulator Sim, such that for any conforming, even computationally unbounded adversary $\mathcal{A}$, its views in the following experiments $\mathbf{Ideal}^{\mathcal{A}}(1^\lambda, N, T)$ and $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}(1^\lambda, N, T)$ have statistical distance at most $\epsilon(\lambda, T)$ for any choice of λ, N, and T:

- $\mathbf{Ideal}^{\mathcal{A}}(1^\lambda, N)$: $\mathcal{A}(N, T)$ adaptively issues T priority-queue queries. For each query Q, $\mathcal{A}$ receives not only the output from $\mathcal{F}_{\text{pq}}$ 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 Q.
- $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}(1^\lambda, N, T)$: $\mathcal{A}(N, T)$ interacts with a challenger $\mathcal{C}$ which internally runs a copy of the real-world algorithm $\mathsf{PQ}(1^\lambda, N)$ instantiated with the security parameter λ and the maximum

[2]Note that the adversary knows the index of the query; thus using time as the ideal-world reference does not reveal additional information to the adversary. Basically, in the ideal world, the adversary is able to later on ask to delete an element by specifying the time at which it was inserted.

capacity N. 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.

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 $\mathcal{A}(N, T)$ must satisfy the following:

1. it always submits 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; and
2. the total number of elements inserted into the priority queue that have not been extracted must never exceed N.

Note also that the notion of oblivious simulation captures correcteness and security requirements in a single simulation-based definition — this approach is inspired by the standard literature on multi-party computation.

Typically, we want the scheme's failure probability $\epsilon(\lambda, T)$ to be *negligibly small in λ as long as the total number of requests $T(\lambda)$ is polynomially bounded in λ*. If this is guaranteed, we say that the scheme "*realizes an oblivious priority queue*" as formally defined below.

Definition 1 (Oblivious priority queue)**.** *We say that PQ realizes an oblivious priority queue iff PQ $(1-\epsilon)$-obliviously simulates $\mathcal{F}_{\text{pq}}$, and moreover, the failure probability term $\epsilon(\lambda, T)$ is negligibly small in λ for any T that is polynomially bounded in λ.*

Remark 2 (On the failure probability's dependence on T)**.** Notice that in our definition of oblivious simulation above, we allow the scheme's failure probability $\epsilon(\lambda, T)$ to depend on the number of requests T. In our scheme later, the failure probability suffers from a union bound over T. This definitional approach is standard in the cryptography literature: it is customary to let the scheme's failure probability be dependent on the adversary's running time (which is lower bounded by T in our case). In cryptography, a typical guarantee we aim for is that as long as the adversary's running time is polynomially bounded in the security parameter λ — in our case, this implies that T is polynomially bounded in λ — we would like the scheme's failure probability to be negligibly small in λ.

Remark 3. In the security definition above, we assume that the adversary can only observe the access patterns but not the contents of the data being transmitted. In practice, standard techniques such as encryption or secret-sharing can be employed to hide data contents. Encryption is often used in a single-server setting whereas secret sharing may be employed in a multi-server setting [9, 32] or in multi-party computation [24, 31, 42].

3 Path Oblivious Heap

In this section, we provide a formal description of our oblivious priority queue algorithm. We first present a version instantiated from the non-recursive Path ORAM [39], we then discuss how to

modify the construction to use Circuit ORAM's eviction algorithm [42] to get tighter bounds on the CPU's private cache. This organization is not only to aid understanding, but also because of the fact that the two variants are each recommended for the cloud outsourcing and secure multi-party computation settings respectively.

For simplicity, for the time being we assume that the priority queue is preconfigured with an a-priori upper bound N on the number of elements that it can hold, and an upper bound $T \geq N$ on the total number of priority-queue requests. *We will later get rid of this known-T assumption* in Section 3.5 to support an unbounded number of queries.

We will use the notation λ to denote an appropriate security parameter. We would like that over $\mathsf{poly}(\lambda)$ number of requests, the priority queue's security failure probability be negligibly small in λ.

3.1 Data Structure

The primary data structure is a (non-recursive) Path-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. Every *non-root* bucket $\mathcal{B}$ in the tree can store a suitable constant number of elements and each element is either real or dummy. The root bucket's size will be related to the security parameter — in particular, later we will show that for the failure probability to be negligly small for any polynomially bounded request sequence, the root bucket's capacity $|\mathcal{B}_{\text{root}}|$ must be set to be super-logarithmic in the security parameter λ, We refer the reader to Remark 4 for discussions on what is a suitable constant to adopt for the non-root bucket size.

Real and dummy elements. Each real element is of the form (k, v, ref), i.e., it 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 $(k = \infty, v = \perp, \mathsf{ref} = \perp)$. In particular, a dummy element has the maximum possible key.

Definition 2 (Path invariant [42])**.** *We maintain exactly the same path invariant as in Path ORAM [39]: 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.

Remark 4 (Non-root bucket capacity: provable vs. practical choices)**.** For our mathematical proofs to go through, we need to set non-root bucket's capacity to be 5 or larger. However, for practical implementation, we recommend a practical choice of 2 for the non-root bucket capacity. Note that this is customary in the tree-based ORAM line of work — in all earlier works such as Path

ORAM [39] and Circuit ORAM [42], the provable stochastic bounds are not tight in the constant; and through simulation, one can observe that even smaller bucket sizes lead to exponentially sharp tail bounds. For example, in Path ORAM [39] and Circuit ORAM [42], a bucket size of 4 or 5 is in the formal theorem statements, but these works use a bucket size of 2 or 3 in implementation. How to further tighten these constants in the stochastic proofs seems rather challenging and has been left open since the beginning of this line of work.

3.2 Basic Operations

Throughout our construction, we assume that data contents are either freshly re-encrypted or re-secret-shared when being written to external memory (see also Remark 3).

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 for non-root buckets and in $|\mathcal{B}_{\text{root}}|$ cost cost for the root bucket — we will show later that $|\mathcal{B}_{\text{root}}|$ must be super-logarithmic in the security parameter for the failure probability to be negligibly small for any polynomially bounded request sequence:

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. If such an element exists, return the element; else return dummy. 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.

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 calling $\mathcal{B}.\mathsf{Del}(\mathsf{ref})$.
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. We shall review Path ORAM's eviction algorithm shortly below and this is what we will adopt in our scheme too.
3. $\mathcal{P}.\mathsf{UpdateMin}()$. Whenever we operate on a path, the subtree-mins on the path need to be updated using an $\mathsf{UpdateMin}$ 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.

Background: review of Path ORAM's eviction. As mentioned, $\mathcal{P}.\mathsf{Evict}()$ is an eviction algorithm that operates on a path. In Path ORAM [39], the eviction algorithm takes the entire path and tries to pack the elements on the path as close to the leaf as possible while respecting the path invariant of every element.

To achieve this, the CPU can locally perform the following algorithm after reading back the path, where the root is assumed to be at the smallest level of the path $\mathcal{P}$, a slot $\mathcal{L}$ in a bucket is said to be empty if the slot currently contains a dummy element, and recall that each element's position label pos is encoded in the reference ref:

Path ORAM's eviction algorithm [39]:

For bucket $\mathcal{B}$ ranging from the leaf to the root on $\mathcal{P}$:

For every empty slot in $\mathcal{L} \in \mathcal{B}$:

If $\exists$ an element in $\mathcal{P}$ in a level smaller than $\mathcal{B}$ and moreover this element can legitimately reside in $\mathcal{B}$ based on its position label pos:

Move this element to this the slot $\mathcal{L}$.

3.3 Heap Operations

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.

Path Oblivious Heap:

- **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 $\mathsf{pos} \xleftarrow{\$} \{0, 1, \ldots, N-1\}$.
  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 $\mathcal{P}.\mathsf{ReadNRm}(\mathsf{ref})$, $\mathcal{P}.\mathsf{Evict}()$, and $\mathcal{P}.\mathsf{UpdateMin}()$.
- **ExtractMin**(): Let $(k, v, \mathsf{ref}) :=$ **FindMin**() and call **Delete**(ref).

We shall prove the following theorem later in Appendix B.

Theorem 2 (Oblivious simulation of $\mathcal{F}_{\text{pq}}$)**.** *Suppose that every non-root bucket has capacity at least 5 and that the root bucket's capacity is denoted by $|\mathcal{B}_{\text{root}}|$. The above* PQ *algorithm is a $(1-\epsilon)$-oblivious simulation[3] of $\mathcal{F}_{\text{pq}}$ for $\epsilon = T \cdot e^{-\Omega(|\mathcal{B}_{\text{root}}|)}$.*

As a special case, suppose that all non-root buckets have capacity 5 and the root bucket's capacity $|\mathcal{B}_{\text{root}}| = \omega(\log \lambda)$, then the resulting scheme realizes an oblivious priority queue by Definition 1.

[3] Although our earlier formal definition of oblivious simulation assumes that the priority queue algorithm PQ does not know T upfront, it is easy to extend the definition for a priority queue algorithm PQ that knows an upper bound on T a-priori — basically, in $\mathbf{Real}^{\mathsf{PQ},\mathcal{A}}(1^\lambda, N, T)$, pass not only N to PQ as input, but also T.

Proof. Deferred to Appendix B. □

3.4 Asymptotical Efficiency

Clearly **FindMin** looks at only the root's subtree-min label, and moreover each **Delete**, **Insert**, and **ExtractMin** request consumes path-length amount of bandwidth. Suppose that the CPU locally stores the root bucket, then it is not hard to see that to support each **Delete**, **Insert**, and **ExtractMin** request, only $O(\log N)$ entries of the priority queue need to be transmitted. Additionally relying on Theorem 2 to parametrize the root bucket's size, we can now analyze the scheme's efficiency and security tradeoff as formally stated in the Theorem 3 below. Theorem 3 follows in a somewhat straightforward fashion from Theorem 2 through variable renaming and augmented with a simple performance analysis — we present its proof nonetheless for completeness.

Theorem 3. *Let $N(\lambda)$ and $T(\lambda)$ denote the a-priori upper bound on the number of elements in the priority queue and the maximum number of requests respectively. 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 $C := \lceil (D + \log T)/w \rceil$.*

There is a suitable constant c_0 such that if we instantiate the algorithm in Section 3 with the root bucket capacity being $|\mathcal{B}_{\text{root}}| = c_0 \log \frac{1}{\delta(\lambda)}$ where $0 < \delta(\lambda) < 1/T(\lambda)$, and a non-root bucket capacity of 5, then the resulting scheme $(1 - T\delta)$-obliviously simulates $\mathcal{F}_{\text{pq}}$ consuming $O(C \cdot (\log N + \log \frac{1}{\delta}))$ words of CPU private cache; and moreover it completes each **FindMin** *request in $O(C)$ bandwidth, and supports each* **Delete**, **Insert**, *and* **ExtractMin** *request consuming at most $O(C \log N)$ bandwidth, measured in the number of words transmitted between CPU and memory.*

Proof. Consider the algorithm presented in Section 3 that is parametrized with N and T, a non-root bucket capacity of 5, and the root's capacity being $|\mathcal{B}_{\text{root}}| := c_0 \cdot \log(1/\delta)$ where c_0 is a suitable constant such that the $\exp(-\Omega(|\mathcal{B}_{\text{root}}|))$ failure probability in Theorem 2 is at most δ. Now, the $(1 - T\delta)$-oblivious simulation claim follows directly from Theorem 2. For the asymptotic performance claims, observe that each **FindMin** request looks at only the root's subtree-min label and thus takes only $O(C)$ time; and each **Delete**, **Insert**, and **ExtractMin** request visits $O(1)$ number of tree-paths where the root may be permanently cached on the CPU side; thus $O(C \log N)$ bandwidth is required. □

We want the failure probability to be negligibly small in λ as long as T is polynomially bounded in λ. To achieve this we can set the root bucket's capacity to be super-logarithmic in λ ensuring that the per-request failure probability is negligibly small. We thus derive the following corollary.

Corollary 2 (Oblivious priority queue)**.** *Let C be defined in the same way as Theorem 3. For any arbitrarily small super-constant function $\alpha(\lambda)$, for every T there exists an algorithm that realizes an oblivious priority queue by Definition 1 supporting at most T requests, consuming $O(C \cdot (\log N + \alpha(\lambda) \log \lambda))$ words of CPU private cache, and moreover the algorithm completes each* **FindMin** *request in $O(C)$ bandwidth, and each* **Delete**, **Insert**, *and* **ExtractMin** *request in $O(C \cdot \log N)$ bandwidth measured in number of words transmitted between the CPU and memory.*

Proof. Follows in a straightforward fashion from Theorem 3 by letting the root bucket capacity $|\mathcal{B}| = \alpha(\lambda) \cdot \log \lambda$. Now, for $T = \mathsf{poly}(\lambda)$, the failure probability is bounded by $\mathsf{poly}(\lambda) \cdot \exp(-\Omega(\alpha(\lambda) \log \lambda))$ which is negligibly small in λ as long as $\alpha(\cdot)$ is super-constant. □

3.5 The Case of Unknown T

So far we have assumed that we know the maximum number of requests (denoted T) a-priori since the scheme needs to allocate enough space in each entry to store a timestamp of $\log T$ bits. It is easy to remove this assumption and construct an oblivious priority queue that only needs to know N a-priori but not T. Recall that we needed the $\log_2 T$-bit timestamp in the references only to serve as a unique identifier of the element. Based on this idea, we will leverage an oblivious priority queue and an oblivious stack [44].

1. *Primary* PQ: the primary PQ works just as before except that the originally $\log_2 T$-bit τ is now replaced with a $\log_2 N$-bit unique identifier. We will leverage an unconsumed-identifier stack to ensure that every element existing in PQ receives an unconsumed identifer τ.

2. *Unconsumed-identifier stack* S: stores a list of unconsumed identifiers[4] from the domain $\{0, 1, \ldots, N-1\}$. Initially, S is initialized to contain all of $\{0, 1, \ldots, N-1\}$.

Now, for each PQ.**Insert** query, we call $\tau \leftarrow \mathsf{S}$.**Pop** to obtain an unconsumed identifier and τ will become part of the reference for the element being inserted. Whenever PQ.**ExtractMin** or PQ.**Delete** is called, let τ be part of the reference for the element being extracted or deleted. We now call S.**Push**(τ) to return τ to the unconsumed pool.

In our unknown-T construction, the metadata stored in each entry is now of $O(\log N)$ bits rather than $O(\log T)$ bits for the earlier known-T case (note that in general, $T \geq N$). We thus obtain the following corollary where C is now redefined as $C := \lceil (D + \log N)/w \rceil$.

Corollary 3 (The case of unknown T)**.** *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 $C := \lceil (D+\log N)/w \rceil$.*

For any arbitrarily small super-constant function $\alpha(\lambda)$, there exists an algorithm that realizes an oblivious priority queue by Definition 1 supporting an unbounded number of requests, consuming $O(C \cdot (\log N + \alpha(\lambda) \log \lambda))$ words of CPU private cache, and moreover the algorithm completes each **FindMin** *request in $O(C)$ bandwidth, and each* **Delete**, **Insert**, *and* **ExtractMin** *request in $O(C \cdot \log N)$ bandwidth measured in number of words transmitted between the CPU and memory.*

Proof. We have described the construction in the paragraph before this corollary, where every instance of known-T oblivious priority queue adopted satisfies Corollary 2. The performance analysis follows in a straightforward fashion by observing that we can construct an oblivious stack (where each element fits in a single memory word) with $O(\log N)$ bandwidth assuming $O(\log N + \alpha(\lambda) \log \lambda)$ words of CPU private cache due to Wang et al. [44]. □

4 The Circuit Variant: Tighter Bounds and Applications in Multi-Party Computation

So far, we presented an oblivious heap construction instantiated from the (non-recursive) Path ORAM [39]. The resulting scheme (i.e., the Path-variant) requires that the CPU be able to locally store and process path-length amount of data, i.e., a total of $O(\log N + \log \frac{1}{\delta})$ of priority queue elements where δ is the per-request failure probability (see Theorem 3). We recommend the Path-variant for cloud outsourcing scenarios since it is reasonable to assume that the client can store $O(1)$ tree-paths; and by making this assumption, every priority queue request can be served in a single roundtrip.

[4] In fact, for type-revealing security, even a *non-oblivious* stack would work. For type-hiding security (see Appendix D), however, we would need an oblivious stack that hides the type of operations.

The Path-variant, however, is not the best candidate for a secure multi-party computation scenario. Suppose that each priority queue entry fits in a single memory word of w bits, with the earlier Path-variant, each priority queue request is supported with a circuit of size $O(wL \log L)$ where $L = O(\log N + \log \frac{1}{\delta})$ denotes the path length [42,43]. Specifically, the circuit size is dominated by the eviction circuit; and implementing Path ORAM's eviction algorithm as circuit requires oblivious sorting on the path as previous works have shown [42, 43], resulting in a size-$O(wL \log L)$ circuit.

In this section, we describe a Circuit-variant where the eviction algorithm follows that of non-recursive Circuit ORAM [42]. This variant overcomes the above two drawbacks in exactly the same way how Circuit ORAM [42] overcomes the drawbacks of Path ORAM [39]:

1. it reduces the CPU's private cache to $O(1)$ words; and
2. in a multi-party computation scenario, the Circuit-variant results in optimal circuit size. Specifically, assuming that each entry fits in one word of size w, then the circuit size for each priority queue request is bounded by $O(w \cdot L)$ where $L = O(\log N + \log \frac{1}{\delta})$ denotes the path length.

As earlier works have pointed out [24, 42], having small CPU cache is in fact important for having small circuit size in multi-party computation. In a multi-party computation scenario, the CPU's private state is secret shared among multiple parties and the CPU's computation must be done through an expensive cryptographic protocol that models the CPU's private computation as a boolean or algebraic circuit; and moreover, this circuit must take all of the CPU's private cache as input. For this reason, if we adopt an algorithm with L amount of CPU cache, the circuit representing CPU computation must have size at least L (but possibly more than L, e.g., the Path-variant requires $\Theta(wL \log L)$ circuit size).

4.1 Algorithm

Background on Circuit ORAM. Recall that Circuit ORAM is an improvement of Path ORAM: Path ORAM's eviction (see Section 3.2) requires that the CPU store and compute on an entire path; Circuit ORAM devises a new eviction algorithm that avoids this drawback: specifically, the eviction algorithm now can be evaluated by a CPU with $O(1)$ words of private cache. At a very high level, without going into algorithmic details, recall that Path ORAM's eviction algorithm is the most aggressive (i.e., greediest) one can perform on a path. By contract, Circuit ORAM's eviction algorithm is the greediest but subject to making only a single data-scan over the path, from root to leaf (and additionally, $O(1)$ number of metadata scans to prepare for the actual data scan).

More specifically, Circuit ORAM [42]'s eviction algorithm in fact differentiates between two types of paths[5]:

- If $\mathcal{P}$ is *not* a path where a ReadNRm operation has just taken place, then a full eviction is performed where the CPU makes a single data scan over the entire path $\mathcal{P}$ from root to leaf, and attempts to evict as greedily as possible subject to a single data scan (as mentioned, $O(1)$ metadata scans are performed first to prepare for the data scan).

[5]In this paper, we consider the variant of Circuit ORAM [42] with provable stochastic bounds on the overflow probability. The Circuit ORAM paper in fact suggests a simplified version recommended for implementation, where the partial eviction on the ReadNRm path is not performed. Although so far we do not know a formal stochastic proof for this simplified variant, empirical evaluations show that this simplified variant has the same exponentially sharp tail bound on the overflow probability as the provable variant.

- If $\mathcal{P}$ is a path where a ReadNRm operation has just taken place, the eviction algorithm now performs a partial eviction instead: in this case, eviction is performed on a segment of $\mathcal{P}$ from the root to the bucket where an element has just been removed; for the remainder of the path $\mathcal{P}$, dummy eviction is performed.

The details of Circuit ORAM's eviction algorithm are somewhat involved and we refer the readers to the original Circuit ORAM paper [42] for a full-fledged description and various open-source projects [1,2] for reference implementations. For the purpose of this paper, the details of the eviction algorithm are not important — the reader may treat this part as a blackbox, and in fact even our proofs need not open this blackbox since we reduce the probability of overflow to Circuit ORAM's probability of overflow.

The following fact is shown in the Circuit ORAM paper [42]:

Fact 1 (Efficiency of Circuit ORAM's eviction algorithm)**.** *Given a path $\mathcal{P}$ containing L elements, where each element can be encoded in C memory words, Circuit ORAM's eviction algorithm can be accomplished on $\mathcal{P}$ in $O(C \cdot L)$ bandwidth consuming only $O(1)$ words of CPU private cache.*

Furthermore, the eviction algorithm operating on $\mathcal{P}$ can be encoded as a boolean circuit of size $O(C \cdot w \cdot L)$ where w denotes the bit-length of each word.

The Circuit-variant. We are now ready to describe the Circuit-variant of our Path Oblivious Heap algorithm. Essentially the algorithm is identical to the one described in Section 3, the only difference being that whenever eviction is needed on a path, we employ Circuit ORAM's eviction algorithm instead of Path ORAM's eviction algorithm.

Based on Circuit-variant we can derive the following theorem — note that here we simply state the version with unknown T.

Theorem 4 (Circuit-variant with typical parameters)**.** *Let C, w be defined as in Corollary 3: C denotes the number of words for storing each priority queue entry and w denotes the bit-length for each word. For any arbitrarily small super-constant function $\alpha(\lambda)$, there exists an algorithm that realizes an oblivious priority queue by Definition 1 supporting an unbounded number of requests, consuming only $O(1)$ words of CPU private cache, and moreover,*

1. *it completes each* **FindMin** *request in $O(C)$ runtime and bandwidth, and each* **Delete**, **Insert**, *and* **ExtractMin** *request in $O(C \cdot (\log N + \alpha(\lambda) \log \lambda))$ amortized runtime and bandwidth;*

2. *for every request, the algorithm computes on and updates $O(1)$ tree-paths, and the computation performed can be represented as an $O(Cw \cdot (\log N + \alpha(\lambda) \log \lambda))$-sized boolean circuit.*

Proof. Using exactly the same proof of Theorem 2 in Appendix B. Specifically, the proof of Theorem 2 reduces Path Oblivious Heap's overflow probability to that of Path ORAM, relying on Theorem 7 that was proven in the Path ORAM paper [39]. As stated in Theorem 7, in fact the same theorem holds for Circuit ORAM [42] too. Thus, we can identically prove Theorem 2 for the Circuit-variant. Now, letting every non-root bucket be of capacity 5, letting the root bucket be of capacity $|\mathcal{B}| = \alpha(\lambda) \log \lambda$, we can conclude that the failure probability is negligibly small in λ as long as T is polynomially bounded in λ.

It remains to show the performance statements: observe that **FindMin** examines only the root's subtree-min label, and every other request performs ReadNRm, Evict, and UpdateMin on $O(1)$ tree-paths. Clearly ReadNRm and UpdateMin takes time at most $O(CL)$ on a word-RAM with $O(1)$ words of CPU private cache where $L = O(\log N + \alpha(\lambda) \log \lambda)$ is the path length; further,

the ReadNRm and UpdateMin operations can be represented $O(C \cdot w \cdot L)$-sized boolean circuits. The same also holds for Evict according to Fact 1. Therefore the stated performance analysis holds. □

5 Applications

5.1 Path Oblivious Sort: Practical and Optimal Oblivious Sort

We say that an algorithm $\mathsf{Sort}^m(1^\lambda, \cdot)$ is a $(1 - \epsilon(\lambda))$-oblivious sort algorithm, iff there exists a simulator Sim such that for any input array I containing m elements,

$$\mathcal{F}_{\text{sort}}(I), \mathsf{Sim}(m) \stackrel{\epsilon}{\equiv} (O, \mathsf{addr}) \text{ where } (O, \mathsf{addr}) \leftarrow \mathsf{Sort}^m(1^\lambda, I)$$

where $\mathcal{F}_{\text{sort}}$ is an ideal functionality that sorts the input I and outputs the result, and $\stackrel{\epsilon}{\equiv}$ means that the two distributions have statistical distance at most ϵ.

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.

Theorem 5 (Optimal oblivious sorting)**.** *Let $D := |k| + |v|$ be the number of bits needed to represent an item to be sorted; let w be the bit-width of each memory word, and let $C := \lceil (D + \log N)/w \rceil$. Then, for any $0 < \delta < 1$ and any m, there exists a $(1 - \delta)$ oblivious sort algorithm which consumes $O(1)$ words of CPU cache and $O(Cm(\log m + \log \frac{1}{\delta}))$ bandwidth to sort m elements.*

Proof. Deferred to Appendix C.1. □

5.2 Oblivious Streaming Sampler with Applications to Distributed Differential Privacy

A streaming sampler samples and maintains a random subset of k entries from an incoming stream, without knowing how long the stream is a-priori. The machine that implements the sampler has small space, e.g., it only has $O(k)$ space whereas the entire stream may contain $n \gg k$ entries.

Oblivious streaming sampler. In this paper we specifically care about an *oblivious* streaming sampler which is an important building block in large-scale, federated learning applications as we explain shortly afterwards. Below we first describe what an oblivious sampler aims to achieve, we then motivate why it is an important building block in large-scale, privacy-preserving federated learning. We shall first introduce the notion of an oblivious streaming sampler assuming a server with secure processor such as Intel SGX. Then, we describe an alternative instantiation where the secure processor is replaced with cryptographic multi-party computation.

Imagine that the incoming data stream is encrypted to the secure processor's secret key, and the sampled entries will reside in encrypted format in the server's memory. In this way, the adversary (e.g., the operating system on the server or a rogue system administrator) cannot observe the contents of the stream nor the sampled entries. The adversary, however, can observe the memory access patterns of the sampler. We would like to make sure that from the memory access patterns, the adversary learns no information which entries have been sampled and stored — this property is called *obliviousness*.

Of course, the secure processor can also be replaced by a cryptographic multi-party computation protocol. For example, there are m servers, possibly run by different organizations, and the users

secret share their data items among the m servers, forming a secret-shared stream. The m servers will now perform a long-running multi-party computation protocol to implement the streaming sampler, and all sampled entries will be secret-shared across the parties too. In this scenario, similarly, we also would like to make sure that the access patterns to memory do not leak any information about which entries have been sampled.

Application in distributed differential privacy. Companies like Google and Facebook have been pushing for privacy-preserving federated learning. Specifically, they collect a stream of user data, e.g., Google Chrome's opt-in diagnostics feature collects information about whether users have visited certain websites. The collected data stream will allow the service providers to perform useful machine learning and statistical analysis to optimize their systems and provide better user experience; however, the challenge is to achieve this while protecting each individual user's privacy. Specifically we shall consider differential privacy [14] which has become a *de facto* notion of privacy in such applications.

Until recently, two primary approaches were suggested to achieve distributed differential privacy in such applications:

- *Local differential privacy.* One naïve approach is a mechanism commonly called randomized response (an improved variant of this was employed by Google Chrome in an effort called RAPPOR [16]): basically, each user independently randomizes its own data: by adding sufficient noise to its data, each user is responsible for ensuring differential privacy for itself. This approach enables accurate analytics of simple function (e.g., linear functions and first-order statistics) provided that enough randomized samples can be collected. However, for learning more general functions, this approach may not provide sufficient accuracy.

- *Central differential privacy.* This approach relies on a trusted curator who is entrusted with the cleartext data collected from users, the curator performs the statistical analysis adding appropriate noise as necessary, and publishes the final (noisy) output that is guaranteed to be differentially private. The central approach allows us to compute a much broader class of functions with good accuracy, but having a trusted curator is an undesirable and strong assumption. Although the trusted curator can in principle be emulated by a cryptographic multi-party computation protocol, the cryptographic overhead is large, making it unfit for large-scale, high-bandwidth scenarios such as those faced by Google and Facebook.

Recently, there has been growing interest in a new approach that aims to achieve differential privacy in the so-called *shuffle-* or *sample-model*. Several recent papers [5,10,11,15,18,27] show that if the incoming user data is either randomly shuffled or sampled, for a class of suitable functions, users often can get away by adding much smaller noise to their data than a pure local mechanism such as randomized response (for achieving a similar degree of privacy), and in some cases we can even get a privacy-accuracy tradeoff similar to the central model, but without relying on a trusted curator!

Because of the benefits of the shuffle- and sample- models, it has been raised as an interesting practical challenge how to implement such a shuffler and sampler in practice for large-scale time-series data — this open problem is currently being actively worked on by researchers in the space. Note that no matter whether we adopt a sampler or shuffler, it is typically important that the sampler or shuffler be done by a trusted manner: specifically, *which entries have been sampled* or what permutation has been applied *must be kept secret* for the differential privacy analyses to follow.

In this paper, we show how to rely on an oblivious priority queue to implement such a privacy-preserving sampler suitable for an emerging class of differential privacy mechanisms in the so-called sample model. Note that although the solutions suggested require that the sampler be implemented by either a secure processor or through cryptographic multi-party computation (MPC), implementing a small and specialized task securely (either with secure processor or MPC) is typically much more efficient than having to implement generic data analytics tasks with secure processor or MPC.

In a concurrent and independent work [37], Sasy and Ohrimenko also consider how to design an oblivious sampler motivated by the same distributed differential privacy application — however, their problem formulation is *not* in the streaming setting, and they adopt a security notion where the adversary observes only a serialized list of memory accesses but not the CPU step in which each access is made.

5.2.1 Formal Definition

More formally, consider an ideal functionality $\mathcal{F}^k_{\text{sample}}$, which, upon receiving a stream S, outputs a set of k randomly sampled items from S in a random order, where the sampling is performed without replacement.

Assume that k is bounded by a fixed polynomial in the security parameter λ. An oblivious streaming k-sampler henceforth denoted $(O, \mathsf{addr}) \leftarrow \mathsf{OReservoir}^k(1^\lambda, S)$ is a streaming RAM algorithm which consumes only $O(k)$ memory and makes a single pass over a stream S: the output O contains k sampled items, and the notation addr denotes the ordered sequence of access patterns of the algorithm over all $|S|$ time steps.

We say that $\mathsf{OReservoir}^k(1^\lambda, \cdot)$ obliviously simulates $\mathcal{F}^k_{\text{sample}}$ iff the following holds: there is a simulator Sim, such that for any stream S whose length is a fixed polynomial in the security parameter λ, there is a negligible function $\nu(\cdot)$ such that the following holds:

$$\Big(\mathcal{F}^k_{\text{sample}}(S), \mathsf{Sim}(|S|)\Big) \overset{\nu(\lambda)}{\equiv}$$
$$(O, \mathsf{addr}) \text{ where } (O, \mathsf{addr}) \leftarrow \mathsf{OReservoir}^k(1^\lambda, S)$$

In the above, $\overset{\nu(\lambda)}{\equiv}$ means that the two distributions have statistical distance at most $\nu(\lambda)$.

5.2.2 Warmup: the Famous Reservoir Sampling Algorithm

A beautiful and well-known Reservoir sampling algorithm by Vitter [41], commonly referred to as Algorithm R, allows us to realize a streaming sampler with only $O(k)$ memory — but in the prior streaming algorithms line of work was not concerned about preserving privacy. Algorithm R basically works as follows: for the first k incoming items, store all of them; when the m-th item arrives where $m > k$, with probability $1/m$, overwrite an already stored item, selected at random, with the new item.

Now, the most straightforward implementation of Algorithm R would select an already stored item at random and access that position directly; but this leaks information of the type: item i and item j cannot both be stored, if the algorithm accessed the same memory location during step i and step j. A naïve way to make the algorithm oblivious is to make a linear scan over the entire stored array of size k, and whenever the index to replace is encountered, write the new element; otherwise simply pretend to make a write (but write the old contents back). To ensure full obliviousness, even if the newly arrived item is to be thrown away, we still need to make a linear scan through the k-sized array. In this way, processing every item in the stream requires $O(k)$ time, and this can be rather expensive, if say, $k = \sqrt{n}$ for some large choice of n.

5.2.3 Our Construction

We can rely on an efficient oblivious priority queue PQ, preconfigured with the maximum capacity k, to make Algorithm R oblivious.

- Whenever a new item arrives, choose a random label with $\omega(\log \lambda)$ bits and mark the item with this label. This label will be used as the comparison key for the oblivious priority queue PQ.
- For each $m \leq k$, when the m-th item arrives, after choosing a random label for the item, simply call PQ.**Insert** and insert it into the priority queue.
- For any $m > k$, when the m-th item arrives, first choose a random label for the item; next, flip a random coin ρ that is 1 with probability $1/m$ and 0 otherwise; and finally, do the following:
  - if $\rho = 1$, call PQ.**ExtractMin** to remove the item with the smallest label, and then call PQ.**Insert** to add the new item;
  - else, call PQ.**ExtractMin** to return and remove the item with the smallest label, and then call PQ.**Insert** to add back the same minimum element just extracted.
- Finally, when the stream S ends, call the PQ.**ExtractMin** algorithm k number of times to extract all k elements and write down each extracted element one by one as the output.

If we adopt our Circuit-variant in the above algorithm with the root bucket size set to $|\mathcal{B}_{\text{root}}| = \Theta(1/\delta)$, then on a word-RAM with $O(1)$ words of CPU cache, every item in the stream can be processed in $O(\log k + \log \frac{1}{\delta})$ runtime.

Theorem 6 (Oblivious streaming sampler)**.** *Suppose that* PQ *realizes an oblivious priority queue by Definition 1. Then, the above algorithm obliviously simulates* $\mathcal{F}_{\text{sample}}$*.*

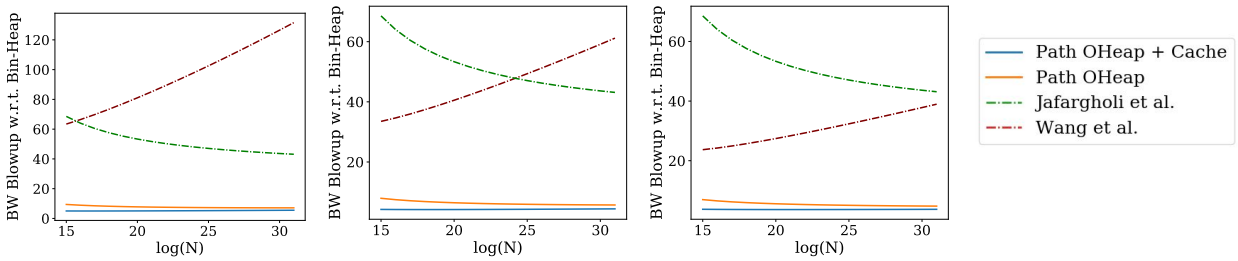
Proof. Deferred to Appendix C.2. □

6 Concrete Performance in Outsourcing

6.1 Experimental Setup

Consider a cloud-outsourcing setting where a client with small local cache stores a dataset, organized according to a priority queue, on an untrusted server. In this section, we refer to the RAM machine's CPU as the *client* and the external memory as the *server*. We built a Java simulator of our algorithm which runs on a single machine which simulates both the client and the untrusted storage.

Metric. We measure the number of bits transferred between the client and the untrusted storage server. Earlier in Section 1.2, we have explained why this is the most suitable metric for a cloud setting and adopted by almost all prior works on ORAM and oblivious algorithms for cloud outsourcing [23, 35, 36, 38, 45]. We compare the bandwidth blowup of our oblivious algorithms relative to insecure baselines, that is, the ordinary Binary Heap and Merge Sort.



(a) Payload = 64 bits **(b)** Payload = 128 bits **(c)** Payload = 256 bits

Figure 1: Concrete Performance for Cloud Outsourcing. The orange curve represents Path Oblivious Heap where the client performs path-caching. The blue curves is when our Path Oblivious Heap performs path+treetop caching to match the client storage of Jafargholi et al. [26]. The green and the maroon curves represent prior works by Jafargholi et al. [26] and Wang et al. [44] respectively. In all figures, the comparison key is 32 bits and the per-request failure probability is at most 2^{-80}.

Concrete instantiation and parameters. We use the Path-variant that is more suitable for cloud outsourcing. As mentioned in Remark 4, instead of choosing a bucket size of 5 needed in the theoretical proof, we choose a bucket size of 2 for our evaluation. We also adopt standard optimizations that have been suggested and adopted in earlier works [19,39,42]. On each insertion, we perform 1 read-path eviction [39] and 1 deterministic reverse-lexicographical order eviction which was initially suggested by Gentry et al. [19].

We apply the standard methodology for determining concrete security parameters for tree-based ORAMs [39, 42] and oblivious data structures [44] — this approach is commonly adopted in this entire line of work since all known theoretical tail bounds are not tight in the constants. Due to the observation that time average is equal to ensemble average for regenerative stochastic processes, we simulate a long run containing more than 3 billion accesses, and plot the stash size against log of the failure probability. Since by definition, we cannot simulate "secure enough" values of λ in any reasonable amount of time, we simulate for smaller ranges of λ and extrapolate to our desired security parameter, i.e., a failure probability of 2^{-80} per request. Our simulation results show that the root bucket is smaller than 20 for achieving a per-request failure probability of 2^{-80}.

In our evaluation, $N = 2^{32}$, and each element has a 32-bit key and varying payload sizes. We consider two natural choices of client-cache sizes for our algorithm:

- *Path caching.* Here we assume that the client can cache a single tree-path plus one additional subtree-min label (of a sibling node). In total, the client caches at most $20 + 2 \log N$ entries and at most $\log N + 2$ subtree-min labels.

- *Path + treetop caching.* In this case, we assume that in addition to caching roughly one path (as noted above), the client additionally caches between 6 to 7 smallest levels. This setting is for comparison with Jafargholi et al. [26]. As we explain below, since their algorithm requires larger client-side storage to be practically efficient, when comparing with them we tune our algorithm to have a matching client-side storage.

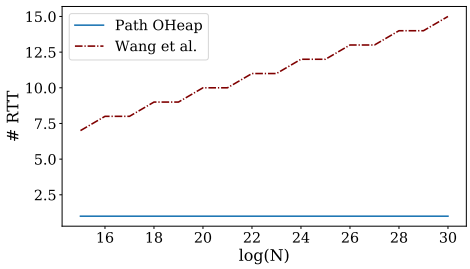
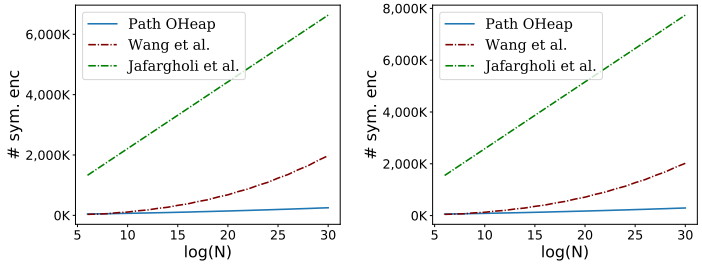


Figure 2: Number of round trips. Wang et al. [44] incurs logarithmic roundtrips due to the recursion in the algorithm; in comparison, our algorithm incurs a single roundtrip per priority-queue request, assuming that the client can store $O(1)$ tree-paths. Assuming small client storage, the worst-case number of round trips for Jafargholi et al. [26] is not in the visible region of the chart.



(a) Payload = 32 bits **(b)** Payload = 64 bits

Figure 3: Concrete Performance for MPC. The blue curve represents Path Oblivious Heap (the circuit-variant) The green and the maroon curves represent prior works by Jafargholi et al. [26] and Wang et al. [44] respectively. In all figures, the comparison key is 32 bits and the per-request failure probability is at most 2^{-80}.

Table 1: Asymptotic circuit size for various schemes in the MPC setting. $|k|$ denotes the key size, $|v|$ denotes the payload size, and N denotes the total capacity of the priority queue. The last column is the circuit size for typical parameters for ease of understanding.

Scheme	Circuit size	Circuit size for $\|k\| = \|v\| = O(\log N)$, $\log N \leq \log \frac{1}{\delta}$
Wang et al. [44]	$O((\log N \cdot (\|k\| + \log N) + \|v\|)(\log N + \log \frac{1}{\delta}))$	$O(\log^2 N \log \frac{1}{\delta})$
Jafargholi et al. [26]	$O((\|k\| + \|v\|) \cdot (\log N + \log \frac{1}{\delta}) \cdot \log \log \frac{1}{\delta})$	$O(\log N \log \frac{1}{\delta} \log \log \frac{1}{\delta})$
Path oblivious heap (circuit-variant)	$O((\|k\| + \|v\| + \log N) \cdot (\log N + \log \frac{1}{\delta}))$	$O(\log N \log \frac{1}{\delta})$

6.2 Evaluation Results

In the simulation, we assume that N elements have already been inserted into the priority queue. We then consider a sequence of requests that contains a sequence of requests containing a repetition of **Insert**, **ExtractMin**, **DecreaseKey** operations alternating in this fashion.

In our simulation, we consider the variant of Path Oblivious Heap with stronger, *type-hiding* security (see Appendix D) i.e., hiding not only the contents of the items inserted into the priority queue, but also the request type as well. We compare our algorithm with two prior works, the oblivious priority queue construction by Wang et al. [44], and the more recent construction by Jafargholi et al. [26]. We measure the number of bits transferred between the client and server, and we consider the bandwidth blowup relative to an insecure baseline, that is, the popular binary heap.

The work by Jafargholi et al. [26] has good practical performance only when the client can store super-logarithmically many entries. More concretely, their algorithm needs to perform maintainance operations called **PushUp** and **PushDown**, which operates on 3 buckets at a time. To have good practical performance, the client needs to be able to cache at least 3 buckets since otherwise oblivious sorting will be necessarily within the triplet of buckets. The bucket size is related to the

security parameter. From the analysis in their paper, and through a more precise simulation of their binomial tail bound (which is tighter than the Chernoff approximation stated in their paper), we determine that their bucket size needs to be 536 to attain a per-request failure probability of 2^{-80}. To be fair, we configure all algorithms, including the insecure binary heap, Wang et al. [44], as well as our algorithm to employ roughly the same amount of client-side cache.

Bandwidth cost. Figure 1 shows the bandwidth results. In this figure, the orange curve is when the client performs path-caching. The blue curve corresponds to the case when Path Oblivious Heap is configured with the same amount of client storage as the prior work Jafargholi et al. [26], fixing the per-request failure probability to be 2^{-80} — through our calculation, this means that our algorithm additionally caches the smallest 6 to 7 levels of the binary tree.

For the stronger notion of type-hiding security, the results show that our algorithm is only $3\times$ to $7\times$ more expensive than the insecure binary heap depending on the ratio of the payload size w.r.t. metadata size. Although not shown in the figure, we note that if hiding the request type is not needed, our algorithm will enjoy further speedup: specifically, **ExtractMin** and **Delete** requests will be $2\times$ faster since only one path is visited during each **ExtractMin** whereas **DecreaseKey** and **Insert** requests will visit two paths. For all data points we considered, our algorithm outperforms prior works by 1-2 orders of magnitude.

Among the oblivious algorithms evaluated, Wang et al. [44] is a logarithmic factor slower than our algorithm and that of Jafargholi et al. [26], although for most of the scenarios we considered in the evaluation, Wang et al. [44] has better concrete performance than Jafargholi et al. [26], especially when either N is small or the payload is large.

Number of round trips. We also evaluate the number of roundtrips incurred by our scheme in comparison with existing schemes, and the result is depicted in Figure 2. Wang et al. [44] incurs logarithmically many roundtrips due to the recursion in the algorithm; in comparison, our algorithm incurs a single roundtrip per priority-queue request, assuming that the client can store $O(1)$ tree-paths. Basically in our algorithm, the client can fetch the at most 1 ReadNRm and at most 2 Evict paths altogether, along with the relevant metadata from sibling nodes necessary for the UpdateMin operation; now, the client locally updates these paths and writes the result back to the server in one round-trip.

We did not plot the number of roundtrips for Jafargholi et al. [26] since every now and then, their scheme incur a worst-case bandwidth cost of $\Theta(N)$, and thus their number of roundtrips can be almost linear in N in the worst case assuming that the client can store only poly-logarithmically many entries (the curve will not be in the visible area of our plot if we had plotted it).

7 Concrete Performance in MPC

We next evaluate our scheme's concrete performance in secure multi-party computation (MPC) scenarios.

Schemes to compare with. We compare the cost incurred for MPC for Wang et al. [44], Jafargholi et al. [26], and our scheme. The asymptotic circuit sizes incurred by these schemes are depicted in Table 1. Assuming that the key size is at least $\log N$ bits, then roughly speaking, our scheme asymptotically outperforms Wang et al. [44] by a logarithmic factor, and we outperform Jafargholi et al. [26] by a $\log \log$ factor. Although Wang et al. [44] is asymptotically worse than Jafargholi et al. [26], as our evaluation results show, Wang et al. [44] outperforms Jafargholi et

al. [26] in practice and in this sense, Wang et al. [44] represents the state-of-the-art in terms of concrete performance.

Setup and metrics. As mentioned, the empirical state-of-the-art is an earlier work by Wang et al. [44] (CCS'14) where they evaluated the concrete performance of their oblivious priority queue scheme in a secure multi-party computation scenario. We adopt *the same experimental setup and evaluation metrics as Wang et al. [44]* to best contrast with prior work.

Like Wang et al. [44], we consider an encrypted database scenario, where Alice stores the encrypted, oblivious priority queue, and Bob comes with the priority queue requests. After each query, Alice obtains the new state of the database, without learning the request Bob has performed, nor the answer.

We use an open-source, semi-honest garbled circuit backend called FlexSC [1]. As Wang et al. [44] point out, the bottleneck is the cost of generating and evaluating garble circuits; therefore, the number of symmetric encryptions (AES) is an indicative performance metric. The metric is also platform independent which facilitates reproducibility of the result. We note that modern processors with instruction-level AES support can compute 10^8 AES-128 operations per second. Just like Wang et al. [44], our evaluation assmes that the oblivious data structure is already set up in a preprocessing phase; and our evaluation focuses on the online cost per request.

Concrete instantiation and parameters. We use the Circuit-variant for our MPC-related evaluations. We adopt the same practical optimizations as suggested in the original Circuit ORAM work [42]: we choose a bucket capacity of 2 although the theoretical proofs need to assume a larger constant; we do not perform eviction on the path where an element has been extracted; and we perform two evictions based reverse-lexicographical ordering per request. As mentioned, in the tree-based ORAM and oblivious data structure line of work, the variants with provable stochastsic bounds always perform a constant factor worse than the schemes that have been implemented and are known to enjoy the best empirical performance. Again, we determine concrete security parameters using a standard methodology detailed in Section 6 — but this time we do it with Circuit ORAM's eviction algorithm. For a bucket size of 3 and using a reverse-lexicographical order for choosing eviction paths, we found that a root-bucket size of 33 is sufficient for achieving 2^{-80} security.

Evaluation results. We show the results in Figure 3. When the database contains 2^{30} entries, our scheme results in $7\times$ to $8\times$ fewer number of symmetric encryptions than the prior state of the art [44]. Since our improvement is asymptotic, the speedup will become greater with larger data sizes.

Acknowledgments

The author would like to thank Kai-Min Chung for insightful technical discussions. Daniel Rong and Harjasleen Malvai created the open source reference implementation available at `https://github.com/obliviousram/PathOHeap`. The author is grateful to the IEEE S&P'20 reviewers for their detailed, thoughtful, and insightful comments over two iterations, which helped tremendously in improving the paper. This work is supported in part by NSF CNS-1453634, an ONR YIP award, and a Packard Fellowship.

References

[1] https://github.com/wangxiao1254/FlexSC.

[2] https://github.com/data61/MP-SPDZ.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *STOC*, 1983.

[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. Cryptology ePrint Archive, Report 2018/892, 2018. https://eprint.iacr.org/2018/892.

[5] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. The privacy blanket of the shuffle model. In *CRYPTO*, 2019.

[6] K. E. Batcher. Sorting Networks and Their Applications. AFIPS '68 (Spring), 1968.

[7] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, 2016.

[8] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, 2018.

[9] T.-H. Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *ASIACRYPT*, 2018.

[10] Kamalika Chaudhuri and Nina Mishra. When random sampling preserves privacy. In Cynthia Dwork, editor, *CRYPTO*, 2006.

[11] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling, 04 2019.

[12] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *Asiacrypt*, 2014.

[13] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *USENIX Security*, 2015.

[14] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[15] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *SODA*, 2019.

[16] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[17] Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. In *STOC*, 2019.

[18] Johannes Gehrke, Michael Hay, Edward Lui, and Rafael Pass. Crowd-blending privacy. In *CRYPTO 2012*, 2012.

[19] 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 *PETS*, 2013.

[20] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[21] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[22] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in O(N Log N) time. In *STOC*, 2014.

[23] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[24] 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.

[25] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *SODA*, 2019.

[26] 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`.

[27] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. What can we learn privately? In *FOCS*, 2008.

[28] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *CRYPTO*, 2018.

[29] Zongpeng Li and Baochun Li. Network coding : The case of multiple unicast sessions. 2004.

[30] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.

[31] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S & P*, 2015.

[32] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, 2013.

[33] Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. In *CCS*, 2018.

[34] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *S & P*, 2015.

[35] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.

[36] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *S & P*, 2016.

[37] Sajin Sasy and Olga Ohrimenko. Oblivious sampling algorithms for private data analysis. In *NeurIPS*, 2019.

[38] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *S & P*, 2013.

[39] 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.

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

[41] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985.

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

[43] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *CCS*, 2014.

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

[45] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

[46] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

A Additional Background on Non-Recursive Path ORAM and Circuit ORAM

In this section, we review the (non-recursive) Path ORAM [39] and the Circuit ORAM [42] algorithm. The two algorithms are almost identical with the primary difference being how path evictions are performed. Path ORAM's eviction algorithm requires that the CPU be able to store an entire tree path (otherwise oblivious sorting would be required over the path). In comparison, Circuit ORAM's eviction can be performed by a CPU with a single register.

Data structure. A non-recursive Path ORAM [39] or Circuit ORAM [42], parametrized by a security parameter λ and a capacity parameter N, is formed as a binary tree with N leaves, where each node in the tree is called a bucket. To achieve a security failure probability of $\mathsf{negl}(\lambda)$, the root bucket $\mathcal{B}_{\text{root}}$ should be large enough to store $\omega(\log \lambda)$ blocks, whereas every internal bucket $\mathcal{B}$ holds a suitable constant number of 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 $(\perp, \perp, \perp)$.

Path invariant. The data structure always respects the following path invariant: a real block whose position label is pos must reside on the path from the root to the leaf numbered pos.

Algorithms. The (non-recursive) Path ORAM or 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 $\mathsf{Overflow}$ 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 $\mathcal{P}'.\mathsf{Evict}()$. We will discuss the path eviction algorithsm in more detail shortly in the paragraph "Path eviction".
3. Return pos;

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

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. Call $\mathcal{P}.\mathsf{Evict}()$ where $\mathcal{P}$ is the path defined by pos.
3. Return data.

Path eviction. In Path ORAM, the path eviction algorithm (denoted Evict) works as follows: the CPU fetches the entire path into its local registers and locally computes a new path as follows: pack all real blocks on the path as close to the leaf as possible while respecting the path invariant. Once the new path has been computed, the CPU writes the entire path back to memory.

One can now see that Path ORAM's eviction algorithm requires that the CPU cache the entire path including the super-logarithmically sized root bucket (otherwise expensive oblivious sorting must be applied over the eviction path). Circuit ORAM [42] improves upon Path ORAM and allows a CPU with $O(1)$ words of private cache to perform the eviction, making only $O(1)$ linear scans over the path. In this way, even with $O(1)$ CPU registers, Circuit ORAM's eviction algorithm completes in in $O(|\mathcal{B}_{\text{root}}| + \log N)$ time, i.e., proportional to the path length. We refer the reader

to Section 4 and the Circuit ORAM [42] paper for the details of its eviction algorithm. Note that Circuit ORAM's eviction algorithm differs slightly between a **Read** path where an element has just been removed, and a non-**Read** path selected for eviction.

Stochastic bounds. 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 (Read, addr). It is guaranteed that for a Read 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 Path ORAM's (or Circuit ORAM's) $\mathsf{pos} \leftarrow \textbf{Insert}(\mathsf{addr}, \mathsf{pos}, \mathsf{data})$ algorithm and records the pos that is returned.
- Whenever $\mathcal{C}$ receives a request of the form (Read, addr) from $\mathcal{A}$, it finds out the correct position label pos for addr and calls Path ORAM's (or Circuit ORAM's) $\mathsf{data} \leftarrow \textbf{Read}(\mathsf{addr}, \mathsf{pos})$ 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 Path ORAM (or Circuit ORAM) algorithm.

The following theorem holds for both Path ORAM and Circuit ORAM (with different constants inside the Ω-notation), and the proofs are presented in the respective papers [39, 42]:

Theorem 7 (Overflow probability for Path ORAM [39] and Circuit ORAM [42])**.** *Assume that the Path ORAM or Circuit ORAM scheme is parametrized with a non-root bucket size of* 5*, and a root bucket size denoted* $|\mathcal{B}_{\text{root}}|$*. For any conforming adversary* $\mathcal{A}$ *issuing* T *queries, the probability that the above experiment encounters* Overflow *is upper bounded by* $T \cdot \exp(-\Omega(|\mathcal{B}_{\text{root}}|))$*.*

B Oblivious Simulation: Proof of Theorem 2

We now prove Theorem 2. 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 $\mathsf{ref} := (\mathsf{pos}, \tau)$ 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 $\textbf{Delete}(\mathsf{ref})$ where $\mathsf{ref} := (\mathsf{pos}, \tau)$ 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_∞ is a perfectly oblivious simulation of $\mathcal{F}_{\text{pq}}$.

Lemma 1. *Under the modified notion of access pattern,* PQ_∞ *is a* 1*-oblivious simulation of* $\mathcal{F}_{\text{pq}}$.

Proof. The simulator Sim is defined 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 $\textbf{Ideal}^{\mathcal{A}}$ and $\textbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$ are identically distributed, we make the following observations:

Fact 2. $\mathcal{A}$ *always receives the correct answer upon a* `findmin` *request in the experiment* $\textbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$.

Proof. In the experiment $\textbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$, 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 $\textbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$ or $\textbf{Ideal}^{\mathcal{A}}$, 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 $\textbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$, 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 $\textbf{Ideal}^{\mathcal{A}}$, the adversary $\mathcal{A}$ sees a random path chosen right now by Sim.

It is not hard to see that even in the experiment $\textbf{Real}^{\mathsf{PQ}_\infty, \mathcal{A}}$, 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* $\textbf{Real}^{\mathsf{PQ}, \mathcal{A}}$*, 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 $\textbf{Real}^{\mathsf{PQ}, \mathcal{A}}$ to encounter $\mathsf{Overflow}$ with probability ν, we can easily construct an adversary $\mathcal{A}'$ that cause the (non-recursive) $\mathsf{PathORAM}$ to encounter $\mathsf{Overflow}$ with probability ν too. Specifically,

- $\mathcal{A}'$ invokes a non-recursive $\mathsf{PathORAM}$ parametrized also with N and λ — note that in the binary tree of $\mathsf{PathORAM}$ every bucket has the same capacity as the corresponding node in our PQ algorithm.
- Furthermore, $\mathcal{A}'$ 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`, $\mathcal{A}'$ returns the $\emptyset$ access patterns to $\mathcal{A}$;
- Whenever $\mathcal{A}$ submits an `insert` request, $\mathcal{A}'$ may arbitrarily choose this element's logical address addr to be any fresh address from the range $\{0, 1, \ldots, N-1\}$ that is different from the address of any element inserted but not extracted so far; now $\mathcal{A}'$ 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, $\mathcal{A}'$ obtains the PathORAM's access patterns that are fully determined by two eviction paths $\mathcal{P}$ and $\mathcal{P}'$, $\mathcal{A}'$ returns these paths' identifiers to $\mathcal{A}$.
- Whenever $\mathcal{A}$ issues a `delete` request on an element inserted at time τ, $\mathcal{A}'$ finds out the correct logical address addr of this element and submits a $(\mathsf{Read}, \mathsf{addr})$ request to its challenger. As a result, $\mathcal{A}'$ obtains the PathORAM's access patterns that are fully determined by a read path, $\mathcal{A}'$ 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 PathORAM's random coins — will also encounter overflow.

Thus, the lemma follows directly from Theorem 7 of Appendix A. □

We can now prove Theorem 2. From Lemma 1 and Lemma 2 we have that PQ is a $(1-\epsilon)$-oblivious simulation of $\mathcal{F}_{\text{pq}}$ under the modified notion of access patterns. Since the modified access patterns and the original accesses patterns have a one-to-one correspondence, we conclude that PQ is a $(1-\epsilon)$-oblivious simulation of $\mathcal{F}_{\text{pq}}$ under the original notion of access patterns too.

C Deferred Proofs for Applications

C.1 Deferred Proofs for Oblivious Sort

We now prove Theorem 5. We can construct a simulator Sim' by leveraging PQ's simulator denoted Sim. Our simulator Sim' calls $\mathsf{Sim}(1^\lambda, N, \texttt{insert})$ for n number of times and then calls $\mathsf{Sim}(1^\lambda, N, \texttt{extractmin})$ for n number of times, and outputs the sequence of access patterns.

We first consider a hybrid execution denoted **Hyb** which replaces the PQ in the algorithm with $\mathcal{F}_{\text{PQ}}$ and outputs the resulting output array and the simulated access patterns output from Sim'. By the definition of oblivious simulation, the real-world execution (i.e., joint distribution of output and real-world addresses) has negligible statistical distance from **Hyb**. The proof now follows by observing that the output of **Hyb** is identically distributed as the ideal-world execution.

C.2 Deferred Proofs for Oblivious Streaming Sampler

We now prove Theorem 6. Let Sim be the simulator for PQ. Let Sim' denote the following simulator: when the m-th item arrives:

- if $m \le k$, output the access pattern for writing down a label of $\omega(\lambda)$ bits for the new item, and call and output what $\mathsf{Sim}(1^\lambda, N, \texttt{insert})$ outputs;
- else if $m > k$, output the access pattern for writing down a label of $\omega(\lambda)$ bits for the new item, call $\mathsf{Sim}(1^\lambda, N, \texttt{extractmin})$, followed by $\mathsf{Sim}(1^\lambda, N, \texttt{insert})$, and output the respective outputs;
- finally, when the stream ends, call $\mathsf{Sim}(1^\lambda, N, \texttt{extractmin})$ for a total of k times and output the respective outputs.

$\mathbf{Hyb_1}$. Consider a hybrid execution where we replace PQ with the ideal functionality $\mathcal{F}_{\text{pq}}$. Let **Real** denote the joint distribution of the real-world output and the memory access patterns; let $\mathbf{Hyb}_1$ denote the joint distribution of the output from the hybrid execution, and the simulated memory access patterns output by Sim'. Due to the definition of oblivious simulation, **Real** and $\mathbf{Hyb}_1$ have negligible statistical distance.

$\mathbf{Hyb_2}$. Consider the following hybrid where the incoming stream is processed by an ideal functionality $\mathcal{F}_{\text{AlgR}}$ which internally executes the Algorithm R [41] on the incoming stream, and when the entire stream is all consumed, it outputs the stored items in a random order. Let $\mathbf{Hyb}_2$ denote the joint distribution of the output from $\mathcal{F}_{\text{AlgR}}$ in this hybrid, and the simulated memory access patterns output by Sim'.

Conditioned on no label collision, then in $\mathbf{Hyb}_1$, every time an **Extract** request is made, an independent random element existing in the heap is extracted. Therefore, the distribution $\mathbf{Hyb}_1$ conditioned on no label collisions is identical to that of $\mathbf{Hyb}_2$. Since the labels are $\omega(\log \lambda)$ bits each, the probability of having label collisions is negligibly small since T is polynomially bounded in λ. $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_2$ have negligible statistical distance.

Ideal. Now consider the ideal-world execution and let **Ideal** denote the joint distribution of the output from $\mathcal{F}_{\text{sample}}$ in the ideal-world execution, and the simulated memory access patterns output by Sim'. Vitter [41] proved that running Algorithm R (i.e., $\mathcal{F}_{\text{AlgR}}$) on the stream will produce the same output distribution as running $\mathcal{F}_{\text{sample}}$ on the stream. Therefore, $\mathbf{Hyb}_2$ and **Ideal** are identically distributed.

Summarizing the above, we conclude that **Real** and **Ideal** have negligible statistical distance which implies the theorem.

D Achieving Type-Hiding Security

So far our algorithm hides the contents of the items inserted into the priority queue but does not hide the type of the requests. Recall that in our security definitions earlier in Section 2.2, the simulator in the ideal world receives the sequence of request types (but not the actual requests). In practice, sometimes it is desirable to consider a stronger notion of security where we additionally require that the request types must be hidden too. Formally, we can modify the security definition in Section 2.2: now the ideal-world simulator no longer receives the sequence of request types but only an upper bound on space N and the length of the request sequence T.

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 incurs logarithmic cost too (instead of constant).

Remark 5. As a practical optimization, one can take the *smallest super-sequence* of the access patterns of all types of requests, and incur only this super-sequence access patterns for every request; doing either useful or dummy work with each physical access depending on what the request is. Since there are only constant types of requests, this super-sequence over all request types can be identified a-priori through exhaustive search; moreover, in comparison with the request with the longest access pattern, the super-sequence access pattern is only $O(1)$ longer.

Remark 6. For the unknown-T case (see Section 3.5), we need to push or pop from the unconsumed-identifier stack depending on the type of the request, note that the oblivious stack by Wang et al. [44] can easily support push-dummy or pop-dummy operations to emulate the access patterns of a push or pop operation without performing any real work.

E Concrete Performance for Oblivious Sorting

We now present concrete performance for oblivious sorting considering a cloud outsourcing scenario.

E.1 Optimizations for Path Oblivious Sort

In our experiment, we performed the following straightforward optimizations to the basic Path Oblivious Sorting algorithm. It is easy to see that these modifications do not break security. Recall that the algorithm has an **Insert** phase during which we insert n elements into a Path Oblivious Heap, and an **ExtractMin** phase during which we extract the minimum element from the heap one by one.

1. During the **Insert** phase, we do not update the subtree-min labels.
2. After all n elements are inserted, we calculate all tree nodes' subtree-min labels.
3. During the **ExtractMin** phase, do not perform eviction but update the subtree-min labels on the read path.

It is not hard to see that these optimizations do not change our provable guarantees since it does not introduce extra overflow in comparison with the unoptimized version for every randomness tape adopted by the algorithm.

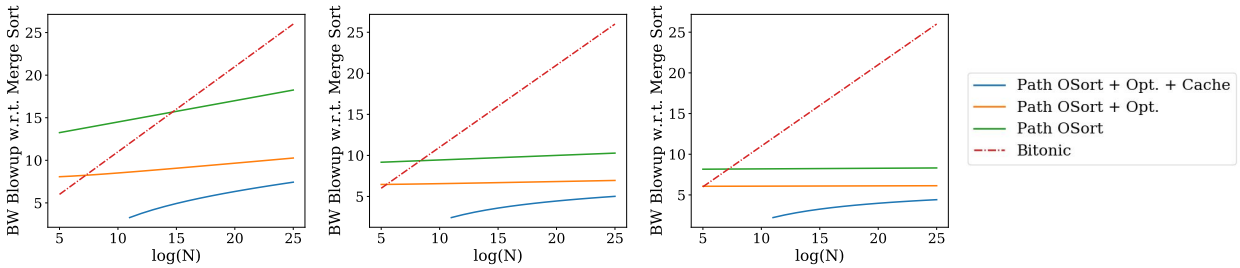
E.2 Evaluation Results

We measure the bandwidth blowup relative to an insecure baseline, that is, Merge Sort.

Results. Figure 4 shows our simulation results and comparison with the *de facto* implementation choice bitonic sort [6]. In this figure, the solid lines show the performance of Path Oblivious Sort whereas the dotted line shows the performance of bitonic sort. We simulated three variants of Path Oblivious Sort:

1. the naïve variant that uses Path Oblivious Heap as a blackbox (see Section 5.1) with path-caching;
2. an optimized variant described in Appendix E.1, also with path-caching; and
3. an optimized variant with path+treetop caching to match the client-side storage of Jafargholi et al. [26];

The figure shows that when the payload is much larger than the comparison key, our Path Oblivious Sort algorithm has only 6× bandwidth blowup relative to the insecure Merge Sort; and has about 4.5× bandwidth blowup relative to Merge Sort if the client caches the smallest $6 \sim 7$ levels of the tree.



(a) Payload = 32 bits **(b)** Payload = 256 bits **(c)** Payload = 2048 bits

Figure 4: Concrete Performance for Path Oblivious Sort. The green and orange curves represent Path Oblivious Sort (with or without practical optimizations) where the client performs path-caching. The blue curve represents Path Oblivious Sort when the client performs path+treetop caching to match the client storage of storage as Jafargholi et al. [26]. We did not draw the curve for Jafargholi et al. [26] since it performs worse than bitonic sort for the parameters we simulated and thus the curve is outside the chart's visible region. In all figures, the comparison key is 32 bits and the per-request failure probability is at most 2^{-80}.

Comparison with bitonic sort. In our simulations, the break-even point w.r.t. bitonic sort is between $n = 2^7$ to $n = 2^{15}$ depending on the payload[6] and comparison key ratio. From this figure, we can see that bitonic sort is a logarithmic factor worse asymptotically in comparison with our algorithm.

Note that for our algorithm, if the bandwidth were exactly $cn \log n \cdot$ (key size + payload size) for some constant c, then the lines should be horizontal. In reality they are not horizontal because the our algorithm has some metadata such as the position label (whose size grows proportional to $\log n$). The slight positive slope of our algorithms' curves stem from the fact that the metadata ratio becomes slightly larger as $\log n$ increases. As expected, when the payload to metadata ratio becomes large, our curves become flatter.

In terms of number of round-trips, our Path Oblivious Sort algorithm requires $O(n)$ number of round-trips assuming that the client can store at least one tree-path — in fact, under logarithmic client-side storage, Bitonic sort incurs asymptotically logarithmically more roundtrips than our algorithm. However, Bitonic sort has fewer roundtrips when the client storage is larger. As Chan et al. [8] show, Bitonic sort can be implemented $O((n/M) \cdot \log^2(n/M))$ number of roundtrips where M denotes the number of entries the client can locally store. In this sense, our Path Oblivious Sort algorithm is preferrable when the network's round-trip latency and the client-side storage are both small, i.e., when the total bandwidth consumed becomes the dominant metric.

Comparison with Jafargholi et al. [26]. We use the simple (and more efficient) version of Jafargholi et al. [26]'s oblivious priority queue algorithm (supporting only **Insert** and **ExtractMin**) to realize oblivious sort. As mentioned earlier, for their algorithm to have good practical performance the client must be able to cache 3 buckets at a time, and each bucket needs to store roughly 536 entries to obtain a per-request failure probability of 2^{-80}. Using path+treetop caching, we can tune our local storage to match theirs as represented by the blue curve in Figure 4.

Under this setting, their algorithm has roughly 96× slowdown relative to Merge Sort whereas

[6]Note that one cannot just sort the comparison key and later on re-arrange the payload accordingly, since naïvely re-arranging the payload leaks information. The oblivious sorting must be applied to the payload too.

our algorithm achieves roughly 4.5× to 6× slowdown relative to Merge Sort depending on how large the payload is. For the range of N simulated, Jafargholi et al. [26]'s never broke even with bitonic sort — even though the latter is asymptotically worse its practical performance is better for the range of N we simulated. For this reason, we did not plot their algorithm in Figure 4 (it would be outside the chart's visible region).