Cache-Oblivious and Data-Oblivious Sorting and Applications

T-H. Hubert Chan[*] Yue Guo[†] Wei-Kai Lin[†] Elaine Shi[†]

Abstract

Although external-memory sorting has been a classical algorithms abstraction and has been heavily studied in the literature, perhaps somewhat surprisingly, when *data-obliviousness* is a requirement, even very rudimentary questions remain open. Prior to our work, it is not even known how to construct a *comparison-based*, external-memory *oblivious* sorting algorithm that is optimal in IO-cost.

We make a significant step forward in our understanding of external-memory, oblivious sorting algorithms. Not only do we construct a *comparison-based*, external-memory oblivious sorting algorithm that is optimal in IO-cost, our algorithm is also *cache-agnostic* in that the algorithm need not know the storage hierarchy's internal parameters such as the cache and cache-line sizes. Our result immediately implies a *cache-agnostic* ORAM construction whose asymptotical IO-cost matches the best known *cache-aware* scheme.

Last but not the least, we propose and adopt a new and stronger security notion for external-memory, oblivious algorithms and argue that this new notion is desirable for resisting possible cache-timing attacks. Thus our work also lays a foundation for the study of oblivious algorithms in the cache-agnostic model.

[*]Department of Computer Science, The University of Hong Kong. `hubert@cs.hku.hk`

[†]Computer Science Department, Cornell University. `yg393@cornell.edu, wklin@cs.cornell.edu, runting@gmail.com`

1 Introduction

In data-oblivious algorithms, a client (also known as a CPU) performs some computation over sensitive data, such that a possibly computationally bounded adversary, who can observe data access patterns, gains no information about the input or the data. The study of data-oblivious algorithms was initiated by Goldreich and Ostrovsky in their ground-breaking work [28,29], where they showed that any algorithm can be obliviously simulated with poly-logarithmic blowup in runtime. In other words, there exists a compiler (also known as an Oblivious RAM (ORAM)) that compiles any algorithm into an oblivious counterpart that runs only poly-logarithmically slower than the original program. Since then, a large number of works further investigated oblivious algorithms: these works either designed customized schemes for specific tasks [21,33,34,41,45] (e.g., graph algorithms [11,34,41,45]) or improved generic ORAM constructions [15,32,38,53,55,56,59]. Oblivious algorithms and ORAM schemes have also been implemented in various applications such as outsourced storage [17,50,54,55,63], secure processors [24,25,39,42,51], and multi-party computation [41,45,59,60]. Finally, various works also proposed novel programming language techniques (e.g., type systems) that mechanically prove a program's obliviousness [39–41] such that a programmer who wishes to code up an oblivious algorithm does not make inadvertent mistakes that result in access pattern leakage.

Goldreich and Ostrovsky's original work considers a RAM machine where the CPU has only $O(1)$ private registers whose values are unobservable by the adversary. However, towards improving the performance, various subsequent works [31,32,55] have made the following important observation: an oblivious algorithm's overhead can be (sometimes asymptotically) reduced if the CPU has a large amount of private cache. Thus, either explicitly or implicitly, these works investigated oblivious algorithms in the *external-memory* model [3,31], where a CPU interacts with a 2-level storage hierarchy consisting of a cache and an external memory. Since the performance bottleneck comes from cache-memory interactions and not CPU-cache interactions, a primary performance metric in this setting is the number of *cache misses* also referred to as the *IO-cost* of an algorithm. In the literature, the atomic unit of access between the cache and the memory is called a *cache-line*. We use B to denote the number of words that are stored in a cache-line. We use M to denote the maximum number of words that can be stored in the cache.

Oblivious sorting [4,8,28,29] is perhaps the single most important building block used in a long line of research on oblivious algorithms [28,29,31,32,41,45]. In the external-memory model, this line of work culminated in Goodrich [31], who acheived an external-memory oblivious sorting algorithm that sorts N elements with $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ IO-cost, under some standard technical assumptions (including the standard tall cache assumption, i.e., $M \geq B^2$, and the "wide cache-line" assumption i.e., $B \geq \log^c N$ for any constant $c > 0.5$). Although it is well-understood that any *comparison-based*, external-memory sorting algorithm (even randomized, cache-aware, and non-oblivious algorithms) must incur at least $\Omega(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ IO-cost due to an elegant lower bound by Aggarwal and Vitter [3] — Goodrich's external-memory oblivious sort algorithm is in fact *not comparison-based*, since it uses an oblivious invertible Bloom Filter in which additions and subtractions are performed on the elements to be sorted.

Therefore, although external-memory sorting has always been a classical abstraction in the algorithms literature and has been investigated for more than three decades [3,27], when *data-obliviousness* is of concern, perhaps somewhat surprisingly, even very rudimentary questions like the following remain to be answered:

> Can we construct a *comparison-based*, external-memory oblivious sorting algorithm that is optimal in IO-cost?

In this paper, we seek to answer not only the above question, but also the following more challenging question:

> Can we construct a *comparison-based*, *cache-agnostic* oblivious sorting algorithm that is optimal in IO-cost?

An external-memory algorithm is said to be cache-oblivious (also referred to as *"cache-agnostic"* in this paper to avoid overloading the term oblivious) if the algorithm is unaware of the parameters of the underlying storage hierarchy such as M and B. In other words, the algorithm's code does not make use of such parameters. This notion has powerful consequences that are well-understood in the algorithms literature: first, a single cache-oblivious algorithm that has optimal performance will have optimal performance on any storage architecture; further, when deployed over a multi-level storage hierarchy, such an algorithm has optimal IO-cost in any level of the storage hierarchy, i.e., not just minimizing IO between the cache and the memory, but also between the memory and disk, between the local disk and remote networked storage, despite the fact that any two adjacent layers have very different M and B parameters. The cache-agnostic paradigm was first introduced by Frigo et al. [27], and investigated by numerous subsequent works [6,9,10,12,19,48,65]. However, none of these works are concerned about security (i.e., data-obliviousness). On the other hand, to the best of our knowledge, known external-memory, data-oblivious algorithms [31, 32, 34, 55] all make explicit use of the parameters M and B. Thus, one conceptual contribution of this work is to *initiate the study of IO-efficient oblivious algorithms in the cache-agnostic model.*

1.1 Our Results and Contributions

Algorithmic contributions. In this paper, we answer the aforementioned questions in the affirmative. We propose a *cache-agnostic* oblivious sorting algorithm that has *optimal* IO-cost in light of Aggarwal and Vitter's lower bound [3] on external-memory sorting (also under standard "tall cache" and "wide cache-line" assumptions like Goodrich [31]). Furthermore, our algorithm is more secure than that of Goodrich's in that we defend against an important cache-timing attack to which prior external-memory oblivious algorithms [31, 32, 34] are vulnerable; moreover, our algorithm is conceptually much simpler than that of Goodrich [31].

- *Stronger security and defense against cache-timing.* To the best of our knowledge, existing external-memory oblivious algorithms including Goodrich's oblivious sort [31] retain security only under a weak adversary that can only observe which cache-lines in memory are accessed upon cache misses. In particular, existing schemes no longer retain security if the adversary can additionally observe which cache-lines are accessed within the cache (i.e., when no cache miss occurs). In other words, known external-memory oblivious algorithms [31, 32, 34] are vulnerable to a well-known cache-timing attack [20, 52, 66, 67] that arises due to the time-sharing of on-chip caches among multiple processes. Specifically, an adversary who controls a piece of software co-resident on the same machine as the victim application, can selectively evict the victim application's cache-lines and thus infer which cache-lines the victim application is requesting through careful timing measurements.

  In contrast, our oblivious sorting algorithm retains security even when the adversary can observe the addresses of all memory words requested by the CPU irrespective of cache hits or misses. We stress that the ability to resist such cache-timing attacks is particularly important, not only because cache-timing attacks have been demonstrated repeatedly in a variety of application contexts where oblivious algorithms are of relevance [25], but also because the security community actually considers ORAM and oblivious algorithms as one of the few provably secure methods to defend against such cache-timing attacks!

- *Simplicity.* Our algorithm is conceptually much simpler than that of Goodrich [31]. Specifically, Goodrich's external-memory oblivious sort algorithm bootstraps in a blackbox manner from an asymptotically sub-optimal ORAM which in turn bootstraps in a blackbox manner from an asymptotically sub-optimal oblivious sort. In comparison, our algorithm is much simpler and completely avoids blackbox bootstrapping from complex building blocks such as ORAMs.

Since several works have shown how to construct generic ORAM from oblivious sort [28,29,32], an immediate implication of Goodrich's result is that there exists an external-memory ORAM such that for a logical memory containing N words, each logical memory access has IO-cost $O(\log N(1+\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}))$, which is also the state-of-the-art external-memory ORAM. For example, if the CPU has $M = N^\epsilon$ amount of private cache for some constant $0 < \epsilon < 1$, then ORAMs can incur as small as $O(\log N)$ blowup for each logical memory access.

We also improve the state-of-the-art external-memory ORAM: our cache-agnostic oblivious sorting algorithm immediately gives rise to a *cache-agnostic* ORAM construction which 1) matches the IO-cost of the best known external-memory ORAM scheme that is cache-aware [31,32]; and 2) unlike existing external-memory ORAM constructions, we additionally offer stronger security and defend against cache-timing attacks.

Our main results are informally summarized in the following theorems.

Theorem 1 (Cache-agnostic oblivious sort (informal)). *Assuming a tall cache and wide cache-lines, there exists a cache-agnostic, statistically secure oblivious comparison-based algorithm that sorts N elements in $\widetilde{O}(N\log N)$ time and $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ IO-cost, where $\widetilde{O}(\cdot)$ hides $\mathsf{poly}\log\log N$ factors.*

Theorem 2 (Cache-agnostic ORAM (informal)). *Assuming one-way functions, a tall cache, and wide cache-lines, there exists a computationally secure strongly-oblivious ORAM scheme in cache-agnostic model that consumes $O(N)$ space for a logical memory of N words, and each logical memory access takes $\widetilde{O}(\log^2 N)$ time and $O(\log N(1+\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}))$ IO-cost, where $\widetilde{O}(\cdot)$ hides $\mathsf{poly}\log\log N$ factors.*

We stress that although the above theorems are stated with the same wide cache-line assumption used in earlier works [31], our constructions actually still work even without this assumption, with the consequence of an additional $\mathsf{poly}\log\log N$ blowup in IO-cost.

Conceptual contributions. We make the following conceptual contributions: 1) we initiate the study of data-oblivious algorithms in the cache-agnostic paradigm that is well-established in the algorithms literature; and 2) we rethink the security definitions for oblivious algorithms in the external-memory model. We observe that earlier definitions and schemes suffer from an important weakness of being vulnerable to cache-timing attacks. Thus, we formulate a new notion of "strong obliviousness" in this paper (see Section 2.4.2). We show that although earlier "weakly oblivious" constructions can be augmented to satisfy our stronger security, such transformations would incur an additional $\Omega(\log M)$ blowup in runtime. In contrast, our oblivious sorting algorithm is optimal in runtime (up to $\mathsf{poly}\log\log$ factors) as well as optimal in IO-cost.

Therefore, we believe that our paper will lay a theoretical foundation for future research on oblivious algorithms in the cache-agnostic model.

1.2 Technical Highlights

Frigo et al. [27] were the first to propose a cache-agnostic sorting algorithm called Funnel Sort with optimal IO-cost. However, their algorithm is not data-oblivious. Below we give an alternative perspective of Funnel Sort that is more conducive to making the algorithm data-oblivious.

First, we describe a meta-algorithm based on a meta-binary tree in which each node produces a sorted stream of elements. Each leaf produces a stream containing a single input element (hence, the number of leaves is equal to the number of elements to be sorted); each internal node takes the two sorted streams from its two children, and merges them to produce a single sorted stream. Finally, the root returns a stream of sorted elements. One can recognize that the above description is a paraphrase of the well-known merge sort [37].

Depending on the order in which work is performed among different nodes in the meta-binary tree, the above meta-algorithm can be implemented in many different ways. Specifically, whenever a pair of sibling nodes both have unconsumed elements left, we can perform some work of merging these two sibling streams into the parent's stream. Funnel Sort [27] can be regarded as a particularly ingenious way to order the work among the nodes in such a meta-binary tree, with the principle of "strike while the iron is hot". In other words, if some elements are fetched into the cache and being worked on, it is desirable to work on these elements again as soon as possible — in this way, cache misses can be minimized.

One way to achieve our stated goal is to design a data-oblivious variant of Funnel Sort [27]. To achieve this, one must first understand why Funnel Sort is not data-oblivious. Upon careful examination, the non-obliviousness of Funnel Sort stems from "load imbalance". In Funnel Sort, when two sibling nodes are merged into the parent, their elements can be consumed at an uneven pace. The rate at which each stream is being consumed depends on the relative ordering of the initial elements, and the access patterns of the Funnel Sort algorithm leaks how fast each stream is consumed. To achieve obliviousness, one idea is to enforce load balance, such that for any pair of sibling nodes, their streams will be consumed at the same pace. Unfortunately, there does not seem to be a direct way of achieving load balance if the initial elements can be arbitrarily ordered.

Our key observation is the following: if the initial elements to be sorted are independently drawn uniformly at random from an appropriate range interval $[0, R)$ and we additionally insert sufficient "slack" (i.e., dummy elements) into the initial array, then indeed there is a way to achieve an approximate notion of load balance. More concretely, we consider a bucket version of the Funnel Sort algorithm, where we operate on buckets, each of which can contain polylogarithmic number of elements. A stream of buckets is said to be "bucket-sorted" (or sorted for short), if for every $i > 1$, any element in the i-th bucket is larger than any element in the $(i-1)$-st bucket. At the beginning, the input array is partitioned into such polylogarithmically-sized buckets that are about half-full, i.e., in each bucket, about half of the elements are dummies.

We now modify our meta-binary-tree to work with buckets, where each bucket can contain real and dummy elements. We use the convention that the leaves are at level 0 and the root is at level $\log_2 N$ (where for simplicity we assume that the number N of elements is a power of 2). Each leaf node is a bucket marked with the range $[0, R)$ where R is a power of 2 — this means that all elements in the bucket must be in the initial range $[0, R)$. Each node at level 1 consumes the two buckets from its two children and produces a stream of 2 buckets marked with the ranges $[0, \frac{R}{2})$ and $[\frac{R}{2}, R)$ respectively. Effectively elements in the two children leaf nodes are redistributed into the two buckets depending on which sub-range their keys fall in. More generally, each node at level i of the tree contains a stream of 2^i sorted buckets that divides up the range $[0, R)$ into 2^i equally sized portions, and effectively all elements contained in descendant leaves are redistributed into the respective bucket. In this way, we can view each parent node as the result of merge-sorting the buckets contained in its two children nodes. In Section 5, we show how to concretely instantiate the above idea such that except with negligible probability, the entire merge process maintains load balance, i.e., every merger consumes its two input streams equally fast and each bucket does not overflow. Now, if we implement such a meta-algorithm using the core ideas behind Funnel Sort to order the work, then we show that the resulting algorithm achieves good cache efficiency.

Unfortunately, the above idea allows us to sort only elements that are sampled uniformly at random, and does not allow us to sort an arbitrary array — moreover, the above algorithm is also *not comparison-based*. However, we may rely on an elegant observation made by Asharov et al. [7] to attain a *comparison-based* oblivious sorting algorithm for arbitrary input keys. Asharov et al. [7] showed that if one could construct obliviously and randomly permute an array such that the access patterns leak no information about the choice of the permutation, then one could construct an oblivious sorting algorithm in the following way: a) apply the oblivious random permutation to permute the input array; and b) rely on any non-oblivious comparison-based sorting algorithm to sort the permuted array. Thus, our idea is to 1) bucketize the input array and pad each bucket with sufficiently many dummies; 2) assign a uniform random key to each real element in all buckets; 3) bucket-sort the input array of buckets based on the elements' randomly chosen keys; and 4) obliviously and randomly permute the elements within each bucket, suppressing all dummies in the process. At this point, we have obtained an oblivious random permutation of the initial array. Finally, we apply a non-oblivious, cache-agnostic comparison-based sorting algorithm such as Funnel Sort itself [27] to the permuted array. This completes our oblivious sort procedure. It is interesting to observe that although in the permutation part of the algorithm, we adopt a non-comparison-based sorting algorithm on uniform random keys, our final sorting algorithm is indeed comparison-based since we rely only on the comparator operator of original input elements.

In summary, our key insight is to combine 1) techniques adopted by classical cache-agnostic algorithms for achieving cache efficiency [19, 27]; and 2) load-balancing ideas that are essential to many data-oblivious algorithms [7, 22, 56, 59]. One way to view our algorithm is that we rely on Funnel Sort twice:

1. First, we borrow the core ideas behind Funnel Sort to construct an oblivious random permutation — to achieve this we have to construct a bucketized version of Funnel Sort and prove load balancing properties.

2. Once we obtain an oblivious random permutation, we use Funnel Sort in a blackbox manner to sort the permuted array.

1.3 Practical Motivation and Justification of Our Model

In this section, we describe a concrete application scenario that is timely due to the wide-spread interest in trusted hardware and secure outsourcing. We argue why the marriage of "data-oblivious" and "cache-oblivious" is particularly compelling in this motivating application; and the same scenario also justifies the importance of cache-timing defense.

We are motivated by emerging "secure outsourcing" applications: imagine that a client wishes to securely outsource private data and computation to a cloud server such as Amazon AWS. The cloud server is equipped with a secure processor such as Intel's SGX [5, 35, 43]. To protect the client's privacy, any data residing in memory or on disk are encrypted such that they can only be decrypted and computed upon within the secure processor, and accesses to data are made oblivious through oblivious algorithms and ORAM schemes. Deploying oblivious algorithms and ORAM schemes in this setting presents the following challenges:

1. The cloud provider (e.g., Amazon AWS) typically implements storage backend that involves multiple storage media such as SSD drives, rotational hard-drives, and networked storage. Storage is often allocated on-demand to applications or virtual machines based on possibly complex scheduling policies that are outside the control of the cloud's tenants.

   This makes cache-oblivious algorithms particularly attractive, because the algorithm designer needs not know the storage parameters such as M and B and the algorithm's implementation will automatically achieve the same IO efficiency in every level of the storage hierarchy irrespective of the cloud provider's storage backend and resource allocation policies.

2. Since the mainstream commodity processors (even secure processors such as Intel's SGX) allow multiple processes or multiple virtual machines to time-share the same on-chip cache, a well-known cache-timing attack is possible where a malicious process (or virtual machine respectively) co-resides on the same physical machine as the victim process (or virtual machine respectively) can learn the victim process's internal secrets such as decryption keys [20, 52, 66, 67]. In such a cache-timing attack, the attacker process selectively evicts a subset of the victim process's cache-lines from the shared cache. Then, through timing measurements, the attacker is able to determine whether the victim process indeed accesses the evicted cache-lines — if so, cache misses would have happened and the execution would have slowed down.

   Such cache-timing attacks have led to wide-spread attention and concern in the security community, who has envisioned ORAM and oblivious algorithms as a provably secure solution to defend against such cache-timing. Unfortunately, existing works on external-memory oblivious algorithms [31, 32, 34] adopt a weaker security notion that makes these schemes vulnerable to cache-timing attacks. An important conceptual contribution of our work is that we rethink the security definition for external-memory oblivious algorithms, and propose a new notion of strong obliviousness that protects against such cache-timing attacks.

1.4 Related Works

Sorting. Sorting is perhaps one of the most fundamental algorithmic building blocks and has been studied for a long time. Sorting has been studied not only in the RAM model [37], but also in circuit model [4, 8, 30]. In particular, we know how to sort N elements using a circuit that is $O(N \log N)$ size [4, 30] as well as in $O(N \log N)$ time in the RAM model [37]. It is also well-known that any comparison-based sorting algorithm must incur $\Omega(N \log N)$ time to sort N elements [37].

In the external-memory RAM model, it is known that any comparison-based sorting algorithm must incur at least $\Omega(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ IO-cost to sort N elements [3]; furthermore, this lower bound is also applicable to randomized sorting algorithms that succeeds with high probability. If data obliviousness is not of concern, then we not only know how to design an external-memory sorting algorithm that matches this lower bound, but also know how to achieve this in a cache-agnostic manner [27]. Unfortunately, as mentioned earlier, when data obliviousness becomes a requirement, our knowledge is less complete — prior to this paper, we did not even know how to construct an optimal data-oblivious, comparison-based, external-memory sorting algorithm (even in the cache-aware model)!

Oblivious algorithms. As mentioned earlier, since the ground-breaking work of Goldreich and Ostrovsky [28, 29], there has been a rich line of work on oblivious algorithms [11, 30, 33, 34, 41, 45] and oblivious RAM [15, 32, 38, 53, 56, 59].

The original work of Goldreich and Ostrovsky [28, 29] focuses on the oblivious simulation of arbitrary programs in the "ordinary RAM model", where the CPU has $O(1)$ private registers that are unobservable by the adversary — in this setting, it is known that oblivious sort is achievable in $O(N \log N)$ runtime [4,30], and that an ORAM scheme can be constructed where each logical memory access requires $O(\frac{\log^2 N}{\log \log N})$ runtime for computational security [14,38], and $O(\log^2 N)$ time for statistical security [14,59].

It is very natural to consider a variant of Goldreich and Ostrovsky's model where the CPU could store more data in a private cache, e.g., up to N^ϵ memory words for some constant $0 < \epsilon < 1$ [32,55], and the adversary can only observe which cache-lines are accessed in between the cache and the external memory. This setting was considered frequently due to emerging cloud outsourcing applications: imagine that a client (e.g., a user's laptop or mobile phone) wishes to store a private dataset on an untrusted cloud server (e.g., Google Drive or Dropbox) [32,54,55,63]. Several earlier works have shown that under N^ϵ CPU private cache, one could obliviously sort $O(N)$ comparable elements in $O(N)$ time [32,55,62]; and thus one could construct a computationally secure ORAM scheme where each logical memory access consumes only $O(\log N)$ IO-cost [32]. Among these works, a subset [55,62] focused on the specific case of N^ϵ CPU private cache, whereas others [31,32] considered the more general case where the CPU's cache size is M for arbitrary choices of M — using standard terminology the algorithms literature, the latter line of research can be formulated as oblivious algorithms in the "external-memory" model [58]. To the best of our knowledge, all known external memory, oblivious algorithms are *cache-aware*, i.e., the algorithms explicitly make use of knowledge of the storage parameters M and B; and moreover, they consider only a weak notion of obliviousness where the adversary can observe an access only when it leads to a cache miss — and thus whenever cache timing attacks [20,52,66,67] are a potential threat, these existing schemes do not offer strong enough security.

Closely related works. In terms of algorithmic techniques, the closest works in nature are Bucket ORAM [22] and recent work by Asharov et al. [7]. The bucketized load balancing technique adopted in this paper was first described in the Bucket ORAM work [22] — although Bucket ORAM used the load balancing property for a very different purpose and not to construct an oblivious random permutation or sorting scheme; it is also not Bucket ORAM's goal to achieve good cache efficiency. Recently, Asharov et al. [7] propose an oblivious sorting scheme with good "data locality" — however, their notion of locality characterizes the number of discontiguous memory regions accessed, and thus is incomparable to our notion of cache efficiency. In fact, their algorithm performs rather poorly in terms of cache efficiency — to sort N comparable elements, their algorithm would result in $\widetilde{O}(\frac{1}{B} \cdot N \log N)$ cache misses where $\widetilde{O}$ hides $\mathsf{poly} \log \log$ factors. The idea of using Bucket ORAM's load balancing technique to obtain an oblivious random permutation was first described in the work by Asharov et al. [7]; however, as mentioned, their specific instantiation of this idea is not cache efficient.

2 Definitions and Preliminaries

2.1 External-Memory Algorithms

We consider algorithms that run on a CPU in the random access machine (RAM) model. We consider an *external-memory* model [3,26,58] where besides the CPU, there is a storage hierarchy consisting of a *cache* and an *external-memory*. The computation proceeds in CPU steps, where in each CPU step, the CPU performs some computation and makes one read or write request. We now clarify the terminology and the parameters that will be adopted throughout the paper.

We assume that our algorithms access data in atomic units called *words*, where each word consists of W bits. Words are indexed by addresses that are chosen from some space $[1..N]$. In each step of execution, a CPU performs some computation over its internal registers, and then makes a `read` or a `write` request to the storage hierarchy. A read request specifies the address of the word to be read, and a write request specifies the address of the word to be written as well as its new value. Throughout this paper, we assume that *the CPU has $O(1)$ number of internal registers*, which cannot be observed by the adversary.

Memory requests are served in the following manner:

- If the address requested is already in the cache, the CPU then interacts with the cache to complete the read or write request and thus no external memory read or write is incurred;
- Else if the address requested does not exist in the cache: 1) first, a *cache-line* containing the requested address is copied into the cache from external memory possibly evicting some existing cache-line from the cache in the process where the evicted cache-line is written back to memory; and 2) then the CPU interacts with the cache to complete the read or write request. Thus, a *cache-line* defines the atomic unit of access between the cache and the external memory.

Notation. Throughout this paper, we use the notation M to denote the cache size, i.e., the number of words the cache can contain; and we use the notation B to denote a cache-line size, i.e., the number of words contained in a cache-line. For notational simplicity, we assume that M and B are both powers of 2.

Cache associativity and replacement policy. The design of the cache can affect the IO-cost of an external-memory algorithm. In the *fully-associative* model, each cache-line from the memory can be placed in any of the $\frac{M}{B}$ slots in the cache. In an r-way associative model, the cache is divided into clusters each containing r cache-lines, and any cache-line can be placed in only one cluster (but can be placed anywhere within that cluster).

If there is no valid slot in the relevant cluster (or the entire cache in the case of full associativity), some cache-line will be evicted from the cluster back to the memory to make space — which cache-line is evicted is decided by what we call a "replacement policy". Common replacement policies in practical systems include Least Recently Used (LRU) and First-In-First-Out (FIFO) [1,19].

Performance metrics for external-memory algorithms. In the ordinary RAM model where a CPU interacts with a memory (i.e., a degenerate 1-level storage hierarchy consisting of only the memory), a standard performance metric is the *runtime* of the RAM algorithm, i.e., the number of interactions between the CPU and the memory.

The external-memory model is defined to better characterize an algorithm's performance in typical real-world system architectures, where accesses to the cache are much faster than accesses to the memory. Therefore, in the external memory model, we not only care about an algorithm's runtime, but also the number of external-memory accesses henceforth referred to as the IO-cost.

Definition 1 (Performance metrics of external memory algorithms)**.** The performance of an external-memory algorithm can be characterized by the following set of metrics:

- its runtime (also known as running time) is the number of times a word is transferred between the cache and the CPU;
- its IO-cost is the number of times a cache-line is transferred between the memory and the cache (thus IO-cost characterizes the number of cache misses);
- its space usage is defined to be N if the algorithm consumes the address space $[1..N]$.

External-memory algorithms for a multi-level storage hierarchy. The above definitions focus on two levels of the storage hierarchy involving only a cache and a memory. In general, we may also consider external-memory algorithms for a multi-level storage hierarchy — e.g., cache, memory, disk, and cloud storage. For each adjacent pair, the former acts as the "cache" for the latter; moreover the parameters M and B are defined separately for each adjacent pair of storage media. With such a multi-level storage hierarchy, it also makes sense to consider the IO-cost between any two adjacent layers — later we will mention that cache-oblivious algorithms have the following compelling advantage: if the algorithm has optimal IO-cost, then it has optimal IO-cost in any multi-level storage hierachy, for any set of parameters, and in between any two adjacent layers in the hierarchy.

We refer the reader to the online book by Vitter [58] for a more detailed introduction to the external-memory model.

2.2 Cache-Oblivious Algorithms

In the design of external-memory algorithms, knowledge of the parameters M and B can often help us minimize the IO-cost — for example, knowing M, the algorithm can fetch precisely M words of data to the cache, do as much work as possible on the M words before moving onto the next batch of M words.

Cache-oblivious algorithms (also referred to as *cache-agonostic* algorithms in this paper) were first introduced in the seminal work by Frigo et al. [27] and later explored in a sequence of subsequent works [6, 9, 10, 12, 19, 48, 65]. In principle, the idea is simple: design external-memory algorithms that are unaware of the parameters M and B — but as it turns out, this simple idea has powerful consequences [19]. As Erik Demaine explains in his excellent tutorial [19], cache-oblivious algorithms have the following compelling benefits: 1) we can have a single algorithm that performs well on any multi-level storage hierarchy without knowing any parameters of the hierarchy, knowing only the existence of a hierarchy; 2) we may design algorithms assuming a two-level hierarchy, and if the algorithm performs well between two levels of the storage hierarchy, it will automatically perform well between any two adjacent levels of the storage hierarchy; and 3) the algorithms are self-tuning, thus avoiding the pain of hand-tuning parameters whenever the algorithm is to be executed on a different system with different parameters than those intended; and thus a cache-oblivious algorithm is easily portable across platforms.

As mentioned, an external-memory algorithm is said to be *cache-oblivious* [19, 27, 48] if it is not aware of the parameters M and B (or in other words, the algorithm's code does not make use of the parameters M and B). Nonetheless, the IO-cost of a cache-oblivious algorithm is often expressed in terms of M and B and other parameters. For example, cache-oblivious (but not data-oblivious) sorting algorithms like funnel sort and cache-oblivious distribution sort are known [27, 48] such that N words can be sorted with $\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$ IO-cost — and due to the lower bound by Aggarwal and Vitter [3], this is also optimal for any external-memory sorting algorithm.

By contrast, we say that an external-memory algorithm is *cache-aware* if the algorithm's code depends on the parameters M and B which are provided as input to the algorithm.

2.3 Ideal Cache Assumptions and Justifications

The IO-cost of external-memory algorithms (including cache-oblivious algorithms) depend on the design of the cache, including its associativity and replacement policy. Throughout this paper, we will adopt the standard practice in the literature [6, 9, 19, 27, 48] and analyze our algorithms assuming an "ideal cache" that adopts an optimal replacement policy and is fully associative. It is important to justify why these assumptions extend to realistic storage architectures, despite the fact that realistic storage architectures are not "ideal". These justifications are standard and well-accepted by the algorithms community [6, 9, 19, 27, 48]. Specifically, Frigo et al. [27, 48] justify the ideal-cache model by proving that ideal-cache algorithms can be simulated on realistic storage hierarchies with degraded runtime — but in worst cases the slowdown is only a constant factor.

Below we borrow from the excellent exposition by Erik Demaine [19], and explain the ideal-cache model in more detail and justify why it is reasonable to adopt such a model in our analysis.

The first assumption: *optimal replacement* specifies that the page-replacement strategy knows the future and always evicts the page that will be accessed farthest in the future. Of course, real-world caches do not know the future, and employ more realistic page replacement strategies such as evicting the least-recently-used block (LRU) or evicting the oldest block (FIFO). The lemma is well-known and states that LRU and FIFO perform at most $O(1)$ factor worse than optimal replacement in terms of IO-cost.

Lemma 1 (Lemma 12 of Frigo et al. [27])**.** *If an algorithm incurs C IO-cost on a cache of size $M/2$ with optimal replacement, then it incurs at most $2C$ IO-cost on a cache of size M with LRU or FIFO replacement (and with the same block size B).*

The second assumption: *full associativity* says that any block can be stored anywhere in cache. In contrast, real-world caches often have limited associativity r where r is a small constant, e.g., 1-way associative (i.e., direct mapped) 2-way, 4-way, or 8-way associative. It is also well-known in the algorithms community that one can build a compiler (that is aware of M and B), such that when given an algorithm Alg incurs C IO-cost with a fully associative cache of size M (assuming LRU replacement), the compiler transforms it to an algorithm Alg' that incurs $O(C)$ IO-cost with an r-way associative cache of size $O(M)$ where $r = O(1)$. We stress that even though this compiler is cache aware, it does not undermine the "cache-oblivious" paradigm. In some sense, each architecture can provide this compiler itself and the design of algorithms can nonetheless follow the cache-oblivious paradigm.

Lemma 2 (Lemma 16 of Frigo et al. [27])**.** *For any constant r, there exists a compiler $\mathsf{compile}$, for any cache-oblivious algorithm Alg that runs on a fully associative LRU cache with parameters (M, B) incurring IO-cost C, $\mathsf{compile}$ takes the parameters (M, B) as input and compiles Alg into another algorithm Alg' such that Alg' runs on a r-way associative LRU cache with the parameters $(O(M), B)$ incurring expected IO-cost $O(C)$.*

Tall cache assumption. When studying external-memory algorithms, a standard and realistic assumption commonly adopted is the "tall cache" assumption [6, 10, 19, 27, 31, 48] — we assume that cache is *taller* than it's width. That is, the number of cache line, M/B, is greater than the size of one cache line, B, where M is the size of cache; or simply stated, $M \geq B^2$.

Wide cache-line assumption (optional). The literature on external-memory algorithm [9, 13, 31, 44, 47, 49, 61, 64] sometimes also makes an additional "wide cache-line assumption", i.e., assuming that $B \geq \log^c N$ where c is a constant (typical works assume that $c > 1$) and N is space consumed by the algorithm. In this paper, to get optimality w.r.t. Aggarwal and Vitter's external-memory sorting lower bound [3], we need to assume that that $B \geq \log^{0.51} N$. However, we stress that our algorithms work nonetheless even *without* the wide cache-line assumption, albeit with an extra $\mathsf{poly} \log\log\lambda$ blowup in IO-cost.

In comparison, Goodrich [31] assumes a slightly weaker version of "wide cache-line": they assume $B \geq \log^{\epsilon} N$ for any constant $\epsilon > 0$.

2.4 Data-Oblivious, External-Memory Algorithms

Unlike cache-oblivious, data-oblivious is a security notion. Informally speaking, an algorithm is said to be data-oblivious, iff for any two inputs, the algorithm's resulting memory access patterns are indistinguishable — in other words, the algorithm's access patterns to data do not leak any information about the inputs. The study of data-oblivious algorithms was initiated by the groundbreaking work by Goldreich and Ostrovsky [28,29] who showed that any (ordinary) RAM algorithm can be obliviously simulated with only polylogarithmic blowup in runtime.

2.4.1 Terminology and Notations

Disambiguation. Henceforth, to avoid overloading the term "oblivious", we shall use the terminology *cache-agnostic* algorithms in place of *cache-oblivious*.

Negligible functions. A function $\epsilon(\cdot)$ is said to be *negligible* if for every polynomial $p(\cdot)$, there exists some λ_0 such that $\epsilon(\lambda) \leq \frac{1}{p(\lambda)}$ for all $\lambda \geq \lambda_0$.

Statistical and computational indistinguishability. For an ensemble of distributions $\{D_\lambda\}$ (parametrized with λ), we denote by $x \leftarrow D_\lambda$ a sampling of an instance according to the distribution D_λ. Given two ensembles of distributions $\{X_\lambda\}$ and $\{Y_\lambda\}$, we say that the two ensembles are statistically (or computationally resp.) indistinguishable, often written as $\{X_\lambda\} \overset{\epsilon(\lambda)}{\equiv} \{Y_\lambda\}$, iff for any unbounded (or non-uniform p.p.t. resp.) adversary $\mathcal{A}$,

$$\left| \Pr_{x \leftarrow X_\lambda} \left[\mathcal{A}(1^\lambda, x) = 1 \right] - \Pr_{y \leftarrow Y_\lambda} \left[\mathcal{A}(1^\lambda, y) = 1 \right] \right| \leq \epsilon(\lambda)$$

2.4.2 Strongly Oblivious Algorithms and ORAM

Adversarial model. For strongly oblivious algorithms and ORAMs, we assume that the adversary can observe the logical address accessed by an algorithm in each CPU step, and whether each access is a read or write. This means that even when the CPU is accessing words in the cache, the adversary can observe which address (within a cache-line) is being requested. In practice, this allows us to model a very strong adversary that is capable of cache-timing attacks, i.e., a malicious operating system running on an SGX processor [5, 16, 35, 36, 43] where on-chip cache is shared among multiple processes. In comparison, to the best of our knowledge, all prior works on external-memory oblivious algorithms and ORAMs assume that the adversary observes accesses only at the *cache-line* granularity but not the full addresses (i.e., as if the adversary sits on the bus between the cache and the memory where addresses are transmitted in the clear but data contents are encrypted).

Below we define the oblivious algorithms we care about, namely, oblivious sorting and oblivious random permutation; and moreover, we define oblivious simulation of RAM programs (all in the external-memory model).

Oblivious simulation of a stateless functionality. In our paper, we will need two building blocks, oblivious sort and oblivious random permutation. We define what it means for a (possibly randomized) algorithm to be oblivious.

Given a stateless, possibly randomized functionality f, and a leakage function $\mathsf{leakage}$, we say that Alg obliviously simulates f if Alg correctly computes the same (possibly randomized) function as f except with negligible probability for all inputs, and moreover, the sequence of addresses requested by Alg (as well as whether each request is a read or write) does not leak anything beyond the allowed leakage. More formally, we have the following definition.

Let $\textsc{Real}^{\mathsf{Alg}}(1^\lambda, I) := (y, \mathsf{addresses})$ be a pair of random variables where y denotes of the outcome of executing $\mathsf{Alg}(1^\lambda, I)$ on input I, and and $\mathsf{addresses}$ represents the addresses incurred during the execution in all CPU steps (including whether each address is a read or write request). Let $\mathcal{F}^{f,\mathsf{leakage}}$ be a wrapper functionality that outputs a pair $f(I;\rho), \mathsf{leakage}(I;\rho)$ where the same randomness ρ is given to the leakage function and to f (we note that $\mathsf{leakage}$ might consume some additional random coins; nevertheless, it receives the randomness f uses to compute the function). The simulator receives this leakage, and has to simulate the addresses as in the real, without knowing the input or output of the function. Formally:

Definition 2 (Strongly oblivious simulation of a stateless (non-reactive) functionality)**.** We say that the stateless algorithm Alg strongly-obliviously simulates a stateless, possibly randomized functionality f w.r.t. to the leakage function $\mathsf{leakage} : \{0,1\}^* \to \{0,1\}^*$, iff there exists a p.p.t. simulator Sim and a negligible function $\epsilon(\cdot)$, such that for any λ and I,

$$\textsc{Real}^{\mathsf{Alg}}(1^\lambda, I) \overset{\epsilon(\lambda)}{\equiv} \{y, \mathsf{Sim}(1^\lambda, L) \mid (y, L) \overset{\$}{\leftarrow} \mathcal{F}^{f,\mathsf{leakage}}(I)\}$$

Depending on whether $\overset{\epsilon(\lambda)}{\equiv}$ refers to computational or statistical indistinguishability, we say Alg strongly-obliviously simulates f w.r.t. $\mathsf{leakage}$ with either *computational* or *statistical* security. Furthermore, if the above terms are identically distributed, then we say that Alg strongly-obliviously simulates f w.r.t. $\mathsf{leakage}$ with *perfect* security.

Intuitively, the above definition requires indistinguishability of the joint distribution of the output of the computation and the addresses accessed (including whether each access is a read or write). Note that here we handle correctness and obliviousness in a single definition.

Given the above definition which captures both deterministic and randomized functionalities, we define two specific oblivious algorithms that we care about, oblivious sorting and oblivious random permutation — note that the former obliviously simulates a deterministic functionality, i.e., the ideal sorting functionality that sorts an input array; while the latter obliviously simulates a randomized functionality, i.e., the ideal random permutation. In both oblivious sort and oblivious random permutation, the only allowable leakage is the length of the input array I.

Let $\mathcal{F}_{\text{sort}}$ be an ideal sorting functionality that takes an input array I and outputs a correctly sorted version.

Definition 3 (Oblivious sort)**.** We say that an algorithm osort is a computationally (or statistically resp.) strongly-oblivious sorting algorithm, iff osort strongly-obliviously simulates $\mathcal{F}_{\text{sort}}$ w.r.t. $\mathsf{leakage}(I;\rho) := |I|$ with computational (or statistical resp.) security.

Let $\mathcal{F}_{\text{perm}}$ be an ideal random permutation functionality that takes an input array I and outputs a randomly permuted array.

Definition 4 (Oblivious random permutation)**.** We say that an algorithm orp is a computationally (or statistically resp.) strongly-oblivious random permutation algorithm, iff orp strongly-obliviously simulates $\mathcal{F}_{\text{perm}}$ w.r.t. $\mathsf{leakage}(I;\rho) := |I|$ with computational (or statistical resp.) security.

Oblivious RAM. So far, we have focused on oblivious simulation of stateless functionalities such as sorting and random permutation. To define Oblivious RAM (ORAM), we will consider stateful algorithms. A stateful algorithm can be activated multiple times over time each time receiving some input and returning some output; moreover, the algorithm stores persistent state in between multiple activations.

Oblivious RAM is a stateful algorithm that obliviously simulates an ideal logical memory that always returns the last value written when an address is requested. More formally , let $\mathcal{F}_{\text{mem}}$ denote the ideal logical memory such that on receiving an input $I := (\mathsf{op}, \mathsf{addr}, \mathsf{data})$ where each op is either `read` or `write`, $\mathcal{F}_{\text{mem}}$ outputs the last value written to addr; or if nothing has been written to addr, it outputs 0. We define an adaptively secure, composable notion for oblivious ORAM below.

Definition 5 (Adaptively secure ORAM)**.** We say that a stateful algorithm oram is a strongly-oblivious ORAM algorithm with computational security iff there exists a p.p.t. simulator Sim, such that for any non-uniform p.p.t. adversary $\mathcal{A}$, $\mathcal{A}$'s view in the following two experiments, $\mathsf{Expt}^{\text{real},\mathsf{oram}}_{\mathcal{A}}$ and $\mathsf{Expt}^{\text{ideal},f}_{\mathcal{A},\mathsf{Sim}}$ are computationally indistinguishable:

```
$\underline{\mathsf{Expt}^{\text{real},\mathsf{oram}}_{\mathcal{A}}(1^\lambda)}$:
$\mathsf{out}_0 = \mathsf{addresses}_0 = \bot$
For $i = 1, 2, \ldots \mathsf{poly}(\lambda)$:
    $I_i \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{i-1}, \mathsf{addresses}_{i-1})$
    $\mathsf{out}_i, \mathsf{addresses}_i \leftarrow \mathsf{oram}(I_i)$
```

```
$\underline{\mathsf{Expt}^{\text{ideal},f}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)}$:
$\mathsf{out}_0 = \mathsf{addresses}_0 = \bot$
For $i = 1, 2, \ldots \mathsf{poly}(\lambda)$:
    $I_i \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{i-1}, \mathsf{addresses}_{i-1})$
    $\mathsf{out}_i \leftarrow \mathcal{F}_{\text{mem}}(I_i)$, $\mathsf{addresses}_i \leftarrow \mathsf{Sim}(1^\lambda)$
```

In the above definition, if we replace computational indistinguishability with statistical indistinguishability and remove the requirement for the adversary to be polynomially bounded, then we then say that oram is a strongly oblivious RAM algorithm with statistical security.

We remark that most existing works on ORAM [28,29,38,53,56,59] often adopt a weaker version of Definition 5 that is not composable. However, it is not difficult to observe that almost all known ORAM constructions [28, 29, 38, 53, 56, 59] also satisfy the stronger notion, i.e., our Definition 5.

2.4.3 Weakly Oblivious Algorithms and ORAMs

To the best of our knowledge, all prior works on external-memory oblivious algorithms and ORAM [31, 32, 34] achieve a weaker notion of security where the adversary can observe only which cache-lines are being transmitted between the cache and memory (and whether each operation is a read or a write). If the CPU operates on a cache-line within the cache, such actions are unobservable by the adversary. As argued earlier, this weaker notion is possibly vulnerable to cache-timing attacks [20, 52, 66, 67]. In the vast majority of commodity CPUs today, the CPU's on-chip cache is time-shared among multiple processes. A malicious process or operating system can cause a subset of the victim process's cache-lines to evict, and through timing measurements, the adversary can then infer whether the victim process accessed those cache-lines.

To compare with existing works, we also formally define this weak notion of security.

Definition 6 (Weakly oblivious simulation)**.** We say that an algorithm Alg weakly obliviously simulates a stateless functionality f iff the earlier Definition 2 is satisfies but where the random variable $\mathsf{Addresses}$ is now replaced with the following observable traces between cache and memory:

1. in every CPU step, which cache-line is transmitted between the cache and memory; and

2. whether each of the above is a read or write operation.

Similar to Definition 2, we can define computational and statistical security respectively.

Note that in the above definition of weak obliviousness, the adversary is allowed to observe the CPU step (i.e., time) in which a certain cache-line is transmitted between CPU and memory — in fact this is important for the following Proposition 1 to hold.

It is not hard to see that strong obliviousness implies weak obliviousness since given the full set of logical addresses requested by the CPU, one can easily simulate the aforementioned observable traces between the cache and the memory.

Fact 1 (Strong obliviousness implies weak obliviousness)**.** *Suppose that an algorithm* Alg *strongly obliviously simulates a stateless functionality* f *with computational (or statistical resp.) security, it holds that* Alg *weakly obliviously simulates a stateless functionality* f *with computational (or statistical resp.) security as well. Similarly, suppose that* oram *is a strongly oblivious ORAM algorithm with computationally (or statistical resp.) security, then* oram *is also a weakly oblivious ORAM algorithm with computational (or statistical resp.) security.*

Proof. Straightforward since the simulator for the weakly oblivious algorithm can simply run the simulator for the strongly oblivious algorithm and then execute the cache algorithm to generate the simulated traces between the cache and the memory. □

Given a weakly-oblivious algorithm or a weakly-oblivious ORAM scheme, there is a trivial method to obtain a strongly-oblivious counterpart, but incurring additional blowup in runtime (but not in IO-cost), and consuming just a constant factor more cache size. Specifically, we can compile the algorithm into one where all contents in the cache are stored in an ORAM scheme (where the atomic unit of access is a memory word). Since best known ORAM schemes [14, 32, 38, 59] incur $O(1)$ blowup in space, we need an $O(M)$-sized cache to obliviously simulate a M-sized cache. Further, recall that best known ORAM schemes [14, 32, 38, 59] incur roughly $O(\log^2 M)$ runtime blowup in comparison with the original RAM. This gives rise to the following proposition:

Proposition 1. *Suppose that an algorithm* Alg *weakly-obliviously simulates a stateless functionality* f *w.r.t. to some leakage* $\mathsf{leakage}$ *with computational (or statistical resp.) security, and on an ideal cache with parameters* M *and* B *consumes runtime* T *and IO-cost* C*. Then, there is an algorithm* Alg' *that strongly-obliviously simulates* f *w.r.t.* $\mathsf{leakage}$ *with computational (or statistical resp.) security, and on an ideal cache with parameters* $O(M)$ *and* $O(B)$*,* Alg' *consumes runtime* $O(T \log^2 M)$ *and IO-cost* C*.*

A similar theorem holds for the case of ORAM (or oblivious simulation of stateful functionalities in general) — we omit the concrete statement.

3 Preliminaries

3.1 Known Results: External-Memory Oblivious Algorithms and ORAMs

In this section, we review some known results on external-memory oblivious algorithms and ORAMs. To the best of our knowledge, all known external-memory oblivious algorithms and ORAM constructions are for the *cache-aware* model [31, 32], where the algorithm is allowed to know the parameters M and B. Further, all existing works on external-memory oblivious algorithms and ORAM [31, 32, 34] adopt the weak obliviousness notion (see Section 2.4.3) which makes these algorithms vulnerable to cache-timing attacks. As mentioned earlier in Section 2.4.3, although a weakly-oblivious external-memory algorithm can be compiled to a strongly-oblivious one, the generic transformation incurs $\mathsf{poly}\log M$ blowup in runtime (and no blowup in IO-cost).

Cache-aware, data-oblivious algorithms and ORAM. Below we state known results on external-memory oblivious algorithms and ORAM constructions. Goodrich and Mitzenmacher [31, 32] propose several cache-aware weakly-oblivious algorithms, which are then utilized to construct an IO-efficient ORAM (Theorem 4). Since their algorithms are weakly-oblivious, in general we can apply Proposition 1 to compile their algorithms to strongly-oblivious ones, resulting in $\log^2 M$ blowup in runtime. However, we observe that for these earlier algorithms [31, 32], since the operations they perform on the cache contents are relatively simple, it is possible to obliviously simulate these operations using only oblivious sorting rather than generic ORAM — thus incurring only $O(\log M)$ blowup in runtime. We now state the known results in light of this.

Theorem 3 (External-memory, cache-aware oblivious sort [31, 32])**.** *Assuming wide cache-line, there exists a computationally secure, weakly-oblivious external-memory sorting algorithm that sorts N memory words in $O(N \log N)$ time and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ IO-cost.*

Corollary 1. *There exists a computationally secure, strongly-oblivious external-memory sorting algorithm that sorts N memory words in $O(N \log N \log M)$ time and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ IO-cost.*

Theorem 4 (External-memory, cache-aware ORAM [31, 32])**.** *Assuming wide cache line, there exists a computationally secure, weakly-oblivious external-memory ORAM algorithm such that for a logical memory of N words, each logical memory access requires $O(\log^2 N)$ time and $O(\log N(1 + \frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}))$ IO-cost.*

Corollary 2. *There exists a computationally secure, strongly-oblivious external-memory ORAM algorithm such that for a logical memory of N words, each logical memory access requires $O(\log^2 N \log M)$ time and $O(\log N(1 + \frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}))$ IO-cost.*

Cache-agnostic oblivious algorithms and ORAM. To the best of our knowledge, no prior works have investigated data-oblivious algorithms and ORAMs in the cache-oblivious model. A trivial way to construct cache-agnostic oblivious sort or ORAM is to simply rely an on ordinary oblivious sort or ORAM scheme where the atomic unit of access is a memory word. To the best of our knowledge, for known ordinary oblivious sorting [4, 8, 30] algorithms and ORAM schemes [28, 29, 32, 53, 56, 59], basically every CPU step can incur a cache miss, and thus the runtime of these algorithms would be their IO-cost as well. Thus ordinary oblivious sorting and ORAM results imply the following theorems:

Theorem 5 (Trivial cache-agnostic oblivious sort [4, 30])**.** *There exists a cache-agnostic sorting algorithm that is perfectly strongly-oblivious, and sorts N elements in $O(N \log N)$ time and $O(N \log N)$ IO-cost.*

Theorem 6 (Trivial cache-agnostic ORAM [14, 38])**.** *Assume that one-way function exists. There exists a cache-agnostic ORAM scheme that is computationally strongly-oblivious, such that each logical memory access incurs $O(\frac{\log^2 N}{\log \log N})$ runtime and $O(\frac{\log^2 N}{\log \log N})$ IO-cost where N is the total number of logical memory words.*

A slightly more non-trivial way to construct a cache-oblivious ORAM algorithm is to rely on a tree-based ORAM scheme such as Circuit ORAM [59] where the unit of access is a single memory word, and additionally we apply the standard van Emde Boas layout [57] to produce a more cache efficient implementation. For computational security, the best instantiation is to additionally apply the position map compression technique originally proposed by Fletcher et al. [23] and later redescribed by Chan and Shi [14] to obtain a computationally secure Circuit ORAM scheme where each logical memory access incurs $O(\frac{\log^2 N}{\log\log N})$ runtime. If we apply the van Emde Boas layout to the resulting computationally secure Circuit ORAM variant, each logical memory access would incur $O(\frac{\log N \log_B N}{\log\log N})$ IO-cost.

We thus obtain the following theorem.

Theorem 7 (Alternate trivial cache-agnostic ORAM [14, 19, 57])**.** *Assume that one-way function exists. There exists a cache-agnostic ORAM scheme that is computationally strongly-oblivious, such that for a logical memory of N words, each logical memory request completes in $O(\frac{\log^2 N}{\log\log N})$ runtime and $O(\frac{\log N \log_B N}{\log\log N})$ IO-cost.*

3.2 Trivial Oblivious Random Permutation

We describe a building block, TrivialORP, an oblivious random permutation which we later use to permute in small buckets. The algorithm $\mathsf{TrivialORP}(1^\lambda, A)$ works as follows, where A is an input list to be permuted.

Procedure $\mathsf{TrivialORP}(1^\lambda, A)$:

- Assign a random key of $\ell(\lambda) = \omega(\log \lambda)$ bits to each element of X;
- Sort all elements by their keys using oblivious bitonic sort, and output the result.

  If there are more than one element assigned with the same key, then abort.

Lemma 3. *The* TrivialORP *algorithm is a statistically strongly-oblivious random permutation. Moreover, given a list of n elements and any super-constant function $\alpha(\lambda)$, the algorithm runs in time $O(n \log^2 n \cdot \alpha(\lambda))$, space $O(n)$ and has asymptotic IO-cost $O(\frac{n}{B} \log^2 \frac{n}{M} \cdot \alpha(\lambda))$.*

Proof. The statistical obliviousness follows from the collision probability of random keys. The choice of ℓ ensures that when the random keys are picked independently for n elements, the probability of key collision is $\mathsf{negl}(\lambda)$. When there is no key collision, the algorithm completes after one application of bitonic sort on the n elements. The time and IO-cost is dominated by bitonic sort on n keys, which is given in Appendix A.1, but the size of each key is $\alpha(\lambda)$ words. ☐

4 Oblivious Random Permutation without IO Efficiency

We first describe a meta-algorithm for realizing a statistically-secure oblivious random permutation (ORP). At the moment, we focus on proving that the algorithm indeed realizes a statistically-secure ORP without worrying about IO-efficiency. Our meta-algorithm in this section essentially provides a logical specification of the algorithm, and the logical specification can be instantiated in various ways. Different implementations can differ in terms of IO-efficiency and later sections will describe a particular, IO-efficient implementation of this algorithm.

Intuition. Our high-level idea is to assign a random key to each element. We next rely on a variant of oblivious sorting called "multiplicity-revealing sort" to group all elements with the same key to the same group. If we choose the parameters carefully, we can make the number of elements in each group relatively small, e.g., roughly $\mathsf{poly}\log\lambda$ elements each group (except with negligible probability). It then suffices to obliviously and randomly permute the elements within each group. This can be achieved through a possibly less efficient ORP algorithm — since this ORP algorithm is only applied to small problems of $\mathsf{poly}\log\lambda$ in size, it will not impact the overall performance too much.

4.1 Definition: Multiplicity-Revealing Sort

We define a building block called "multiplicity-revealing sort" on randomly chosen inputs. Let A be an input array of length $N = |A|$, in which each element is tagged with a key chosen independently uniformly at random from some totally ordered set $\mathcal{R}$ of size $R := |\mathcal{R}| \leq \mathsf{poly}(\lambda, N)$, for some fixed polynomial in λ and N. We use $\vec{\chi} \in \mathcal{R}^N$ to denote the vector of keys received by the elements in A.

A multiplicity-revealing sort algorithm with respect to the uniform distribution on the set $\mathcal{R}$ of keys, henceforth denoted $\mathsf{mrsort}(1^\lambda, A, \vec{\chi})$ satisfies the following: there exists a negligible function $\mathsf{negl}(\cdot)$ such that for any input array A, for any λ, for all but at most $\mathsf{negl}(\lambda)$ fraction of random keys $\vec{\chi}$ in $\mathcal{R}^N$:

- *Correctness.* The procedure $\mathsf{mrsort}(1^\lambda, A, \vec{\chi})$ correctly sorts the elements in the input array A by the assigned keys $\vec{\chi}$, and outputs the multiplicity of each key in $\vec{\chi}$;
- *Security.* The access pattern of $\mathsf{mrsort}(1^\lambda, A, \vec{\chi})$ is efficiently computable and uniquely determined given the multi-set of keys received by the elements, i.e., the multiplicity of each key in $\vec{\chi}$.

For the negligible fraction of random keys $\vec{\chi}$ where the above correctness and security requirements fail, the algorithm may return an arbitrary answer and its access patterns can also be arbitrary.

4.2 From Multiplicity-Revealing Sort to Oblivious Random Permutation

Assuming that there is a multiplicity-revealing sort algorithm denoted mrsort, we can then construct an oblivious random permutation henceforth denoted $\mathsf{MetaORP}$. Below, we describe this meta-algorithm $\mathsf{MetaORP}$, and prove its obliviousness. In Section 4.3, we will describe a mrsort meta-algorithm; and later in Section 5 we will discuss how to implement this meta-algorithm in a cache-efficient manner.

Algorithm $\mathsf{MetaORP}$. Let $\mathsf{TrivialORP}$ denote a statistically strongly-oblivious random permutation scheme (see Section 3.2), and let mrsort denote a multiplicity-revealing sort algorithm as defined above. Our meta ORP algorithm denoted $\mathsf{MetaORP}(1^\lambda, A)$ proceeds in the following steps.

1. Given a list of N elements, assign an independent key uniformly at random from $\mathcal{R}$ (where $|\mathcal{R}| = \mathsf{poly}(\lambda, N)$ for some fixed polynomial $\mathsf{poly}(\cdot)$ in λ and N) to each element of the input array A, let $\vec{\chi} \in \mathcal{R}^N$ denote the vector of random keys assigned to the elements.
2. Invoke an instance of $\mathsf{mrsort}(1^\lambda, A, \vec{\chi})$ to sort the elements in A by the assigned keys and reveal the multiplicity of each key in $\vec{\chi}$.
3. For each group of elements receiving the same random key, fork an instance of $\mathsf{TrivialORP}$ on that group — notice that the procedure naturally leaks the size of each group, but as we prove below, the algorithm satisfies statistical obliviousness despite this leakage (intuitively, this leakage can be simulated without knowing the inputs). Output the outcome of this step.

Lemma 4 (Statistical obliviousness of $\mathsf{MetaORP}$)**.** *The above construction* $\mathsf{MetaORP}$ *is a statistically strongly-oblivious random permutation.*

Proof. It suffices to show that if the mrsort algorithm satisfied both correctness and security requirements on all possible choices of $\vec{\chi}$, then the above construction $\mathsf{MetaORP}$ would be a statistical oblivious random permutation. Since our real mrsort fails on $\mathsf{negl}(\lambda)$ fraction of the sample paths $\vec{\chi}$ (either in terms of correctness or security), it would then follow (by union bound) that our real $\mathsf{MetaORP}$ is statistically oblivious as well. We henceforth proceed pretending that mrsort were perfect, i.e., correctness and security are retained for all choices of $\vec{\chi}$.

Henceforth, all elements with the key i are said to belong to group i. Let $G_i(\vec{\chi}, A)$ denote the elements in the i-th group given the random key choice $\vec{\chi}$ and the input array A. For any fixed input array A, we consider the following sequence of hybrids, and we show that adjacent hybrids are statistically close.

Real. We consider the following real-world distribution:

$$\big(\{\mathsf{TrivialORP}_i(G_i(\vec{\chi}, A))\}_{i\in\mathcal{R}}, \quad \{\mathsf{Addr}_i^{\mathrm{orp}}(G_i(\vec{\chi}, A))\}_{i\in\mathcal{R}}, \quad \mathsf{Addr}_i^{\mathrm{mrsort}}(\vec{\chi})\big)$$

where $(\mathsf{TrivialORP}_i(G_i(\vec{\chi}, A)), \mathsf{Addr}_i^{\mathrm{orp}}(G_i(\vec{\chi}, A)))$ is the end result of permuting the i-th groupand the addresses for applying $\mathsf{TrivialORP}$ to the i-th group. The addresses of mrsort is determined by the load of each group — and the latter information is contained in the random variable $\{\mathsf{TrivialORP}_i(G_i(\vec{\chi}, A)), \mathsf{Addr}_i^{\mathrm{orp}}(G_i(\vec{\chi}, A)\}_{i\in\mathcal{R}}$. Therefore henceforth we simply consider the following joint distribution instead:

$$\mathsf{Real}(A) := (\{\mathsf{TrivialORP}_i(G_i(\vec{\chi}, A))\}_{i\in\mathcal{R}}, \quad \{\mathsf{Addr}_i^{\mathrm{orp}}(G_i(\vec{\chi}, A))\}_{i\in\mathcal{R}})$$

Hybrid 1. Let

$$\mathsf{Hyb}_1(A) := \Big(\{\mathcal{F}_{\mathrm{perm}}(G_i(\vec{\chi}, A))\}_{i\in\mathcal{R}}, \quad \{\mathsf{Sim}^{\mathrm{orp}}(1^\lambda, |G_i(\vec{\chi}, A)|)\}_{i\in\mathcal{R}}\Big)$$

Due to the statistical strong obliviousness of $\mathsf{TrivialORP}$, $\mathsf{Real}(A)$ and $\mathsf{Hyb}_1(A)$ are statistically close for any A.

Ideal. Let $\mathsf{SimLen}(1^\lambda, |A|)$ be the following simulator: sample $|A|$ number of random keys from $\mathcal{R}$. For each $i \in \mathcal{R}$, output how many keys are equal to i, i.e., the load of group i. Henceforth we let SimLen_i denote the i-th component of the vector SimLen.

Let

$$\mathsf{Ideal}(A) := \Big(\mathcal{F}_{\mathrm{perm}}(A), \quad \{\mathsf{Sim}^{\mathrm{orp}}(1^\lambda, \mathsf{SimLen}_i(1^\lambda, |A|))\}_{i\in\mathcal{R}}\Big)$$

Observe that in $\mathsf{Ideal}(A)$, the collection of algorithms, namely, $R := |\mathcal{R}|$ instances of $\mathsf{Sim}^{\mathrm{orp}}$, R instances of SimLen, as well as a single instance of $\mathsf{Sim}^{\mathrm{sort}}$ together comprise an ideal-world simulator that need not know A and only knows $|A|$.

Thus it would suffice to prove the following claim.

Claim 1. *For any input array A, $\mathsf{Hyb}_1(A)$ and $\mathsf{Ideal}(A)$ are identically distributed.*

Proof. It suffices to prove that the follow distributions are identical for any input A where $\vec{\chi}$ is sampled at random.

$$\{\mathcal{F}_{\mathrm{perm}}(G_i(\vec{\chi}, A))\}_{i\in\mathcal{R}}, \{|G_i(\vec{\chi}, A)|\}_{i\in\mathcal{R}} \equiv \mathcal{F}_{\mathrm{perm}}(A), \mathsf{SimLen}(1^\lambda, |A|)$$

In other words, we would like to prove the following random experiments are identically distributed.

- Experiment 0: Assign a random number from $\mathcal{R}$ to each element of A, group all elements with key i into group i (the ordering within a group may be arbitrary), and randomly permute each group. Now output the final permutation X and the each group's load Y.
- Experiment 1: Output a random permutation X' of the array A. Then, assign a random key from $\mathcal{R}$ to each of $|A|$ many elements, and for each i, output how many elements receive i, denoted as Y'.

Let $N = |A|$. It suffices to show that $(X, Y) \equiv (X', Y')$ where $\equiv$ denotes identically distributed. To show this, we need to prove that $X \equiv X'$, $Y \equiv Y'$, and both pairs X, Y and X', Y' are independent. The fact that $X \equiv X'$, $Y \equiv Y'$, and X', Y' independent are all obvious. It remains to prove that X and Y are independent. For X, fix any $y^* \in \mathsf{supp}(Y)$ and any permutation x^* of A, where $\mathsf{supp}(Y)$ denotes the support of Y. Writing each group's load y^* as $(\ell_1, \ell_2, \dots, \ell_R)$, where $R = |\mathcal{R}|$, we have

$$\Pr[X = x^* | Y = (\ell_1, \ell_2, \dots, \ell_R)] = \left[(\ell_1! \cdot \binom{N}{\ell_1})(\ell_2! \cdot \binom{N-\ell_1}{\ell_2}) \dots (\ell_R! \cdot \binom{\ell_R}{\ell_R})\right]^{-1} = \frac{1}{N!}.$$

It follows that $\Pr[X = x^*] = \sum_{y \in \mathsf{supp}(Y)} \Pr[X = x^* | Y = y] \Pr[Y = y] = \frac{1}{N!}$. Therefore, X is independent from Y. □

□

4.3 Choice of Random Keys and Multiplicity-Revealing Sorting Algorithm

In this section, we describe a meta-algorithm that realizes multiplicity-revealing sort. This meta-algorithm can be implemented in various ways and various implementations differ in their cache efficiency. Later in Section 5 we will describe a concrete, cache-efficient implementation that relies on the core ideas behind Funnel Sort [27].

Intuition. We shall implement multiplicity-revealing sort using the bucket sort[1] framework. In the following, we define bucket sort, describe a meta-algorithm to perform bucket sort. We then show how to realize multiplicity-revealing sort using bucket sort as a building block, and provide its analysis.

Definition 7 (Bucket Sort)**.** A bucket with capacity Z is an array that contains Z elements, where each element is either real or dummy. Furthermore, every real element has an ℓ-bit key, and dummy elements do not. A *bucket sort* algorithm, parametrized by ℓ and Z, satisfies the following syntax:

- The input to bucket sort is a list of 2^ℓ buckets denoted $\{\mathsf{B}_i : i \in [2^\ell]\}$, where each bucket contains exactly Z elements, either real or dummy, and each real element has a ℓ-bit key.
- The output is either a list of 2^ℓ buckets denoted $\{\widehat{\mathsf{B}}_i : i \in [2^\ell]\}$ or a failure indicator denoted $\mathsf{Overflow}$. If the algorithm succeeds, then 1) each output bucket $\widehat{\mathsf{B}}_i$ where $i \in [2^\ell]$ contains all the real elements in the input whose keys are equal to i, padded with dummies to a capacity of Z; and 2) moreover, for each output bucket, all real elements must appear before all the dummy ones.

[1] In the literature, bucket sort refers to the sorting framework in which elements with the same key are put in the same bucket, after which elements in each bucket are sorted. Here, we use bucket sort to mean just the initial process of placing elements with the same key in the same bucket.

Data-oblivious subroutine: $\mathsf{MergeSplit}$. We use the following subroutine $\mathsf{MergeSplit}(\mathsf{B}_0, \mathsf{B}_1, t)$. Intuitively, the $\mathsf{MergeSplit}$ subroutine takes in two buckets in which all real elements share a t-bit prefix in their keys, and the subroutine classifies these elements based on the $(t+1)$-st bit of their keys. In the two output buckets, all real elements' keys share the same $(t+1)$-st bit.

More formally, $\mathsf{MergeSplit}$ is the following procedure:

- $\mathsf{MergeSplit}$ takes two input buckets B_0 and B_1 and a parameter t such that the keys of all the real elements in the input buckets have the same length-t prefix.
- Relying on (oblivious) bitonic sort [8], the $\mathsf{MergeSplit}$ algorithm returns two output buckets $(\widehat{\mathsf{B}}_0, \widehat{\mathsf{B}}_1)$ such that bucket $\widehat{\mathsf{B}}_0$ contains the real elements in the input whose keys have 0 as the $(t+1)$-st bit, while bucket $\widehat{\mathsf{B}}_1$ contains those whose keys have 1 as $(t+1)$-st bit. In both output buckets, all the real elements appear before all the dummy ones.

  If either bucket's capacity is exceeded, the operation throws an $\mathsf{Overflow}$ exception.

  It is not hard to see that this can be accomplished using only one bitonic sort. We defer the full details to Appendix A.2.

Meta-algorithm for bucket sort. We describe a meta-algorithm to perform bucket sort using the primitive $\mathsf{MergeSplit}$. This meta-algorithm is similar to the techniques used by Asharov et al. [7], however they do not care about cache efficiency. Different implementations of the meta-algorithm will produce the same output, but can have different performance such as memory usage and IO-efficiency. Recall that the input is a list of 2^ℓ buckets. To describe the abstraction, we make use of a complete binary tree with 2^ℓ leaves, where each node has a FIFO queue that can hold buckets. The leaves are at level 0 and the root is at level ℓ. We use the terms *child*, *parent* and *sibling* associated with a binary tree in their usual sense.

Algorithm 1 Meta-Algorithm for Bucket Sort

```
1: procedure Meta-BucketSort($\{\mathsf{B}_i : i \in [2^\ell]\}$)        // The input is a list of $2^\ell$ buckets.
2:     Construct a binary tree with $2^\ell$ leaves, and initialize an empty FIFO queue for each
       node in the tree.
3:     Insert each of the $2^\ell$ input buckets into the queue of the corresponding leaf node.
4:     while $\exists$ two sibling nodes $x$ and $y$ whose queues are both non-empty do
5:         $(\mathsf{B}_0, \mathsf{B}_1) \leftarrow$ Remove one bucket from each queue of the nodes $x$ and $y$.
6:         If the nodes $x$ and $y$ are at level $i$, $(\widehat{\mathsf{B}}_0, \widehat{\mathsf{B}}_1) \leftarrow \mathsf{MergeSplit}(\mathsf{B}_0, \mathsf{B}_1, i)$.
                   // The whole procedure aborts with Overflow if MergeSplit throws an Overflow exception.
7:         Insert the buckets $\widehat{\mathsf{B}}_0, \widehat{\mathsf{B}}_1$ into the queue of the parent of $x$ and $y$ in the order of $\widehat{\mathsf{B}}_0$
           followed by $\widehat{\mathsf{B}}_1$.
8:     return the list of buckets in the queue of the root node.
```

Non-determinism of meta-algorithm. In line 4 there is a choice to be made to select which pair of sibling nodes to process. While any arbitrary sequence of choices will lead to the same output and runtime, a careful choice is needed for an IO-efficient implementation, which is postponed to Section 5, as well as the runtime. Here, we note that there exists some input such that the meta-algorithm fails for any choice made in line 4, and we will analyze the failure probability soon after implementing multiplicity-revealing sort.

Multiplicity-revealing sort from bucket sort. Recall that a multiplicity-revealing sort takes as input security parameter 1^λ, an array A of length N (containing only real elements), and uniform random keys $\vec{\chi} \in \mathcal{R}^N$ assigned to all elements in A. Let ℓ be the largest integer such that $2^\ell \leq \lfloor 4n / \max\{\omega(\log \lambda), \Omega(\log N)\} \rfloor$, and let $\mathcal{R} = \{0,1\}^\ell$. The following procedure mrsort realizes a multiplicity-revealing sort w.r.t. $\mathcal{R}$.

Procedure $\mathsf{mrsort}(1^\lambda, A, \vec{\chi})$ **w.r.t.** $\mathcal{R} = \{0,1\}^\ell$**:**

- Set bucket capacity $Z := \lceil \frac{4N}{2^\ell} \rceil$ and create 2^ℓ buckets.
- Distribute N real input elements among 2^ℓ buckets $\{\mathsf{B}_i : i \in [2^\ell]\}$, where each bucket contains at most $\lceil \frac{N}{2^\ell} \rceil$ real elements and then is padded with dummy elements to a capacity of Z.
- Perform Meta-BucketSort on the list of buckets as in Definition 7.
- If Meta-BucketSort throws Overflow, output a failure indicator $\perp$. Otherwise, for each output bucket (which is an array in which all real elements appear before all dummies), return all real elements contained in the bucket — in this process we reveal how many real elements are contained in each bucket (i.e., the multiplicity of each key).

Observe that mrsort fails only when Meta-BucketSort throws Overflow. Given that input keys $\vec{\chi}$ are uniformly random, it is sufficient to analyze the failure probability of mrsort due to Overflow in the meta-algorithm Meta-BucketSort. We use the following version of the Chernoff Bound.

Proposition 2 (Chernoff Bound)**.** *Suppose X is a sum of independent $\{0,1\}$-random variables. Then, for any $\beta \geq 2$, $\Pr[X \geq \beta E[X]] \leq \exp(-\frac{\beta E[X]}{6})$.*

Lemma 5 (Failure Probability of mrsort)**.** *The multiplicity-revealing sort,* mrsort*, using ℓ-bit random keys, bucket capacity $Z \geq \frac{4N}{2^\ell}$, and 2^ℓ buckets on a list of N elements fails with probability at most $\ell 2^\ell \cdot e^{-\frac{Z}{6}}$.*

Proof. Recall that in the meta-algorithm, there are 2^ℓ leaf nodes. We analyze the probability that a particular bucket B to be inserted into the queue of some node x has its capacity exceeded.

Suppose node x is at level i. Then, it follows that B can contain real elements that initially reside in the buckets in the descendant leaf nodes of x. Recall that there are at most $2^i \cdot \frac{N}{2^\ell}$ such real elements, each of which receives a key whose length-i prefix is compatible with B independently with probability $\frac{1}{2^i}$. Therefore, the expected number of real elements that B contains is at most $\frac{N}{2^\ell}$. Hence, Chernoff Bound (Proposition 2) implies that bucket B has its capacity Z exceeded with probability at most $e^{-\frac{Z}{6}}$.

Then, a union bound over all buckets from all non-leaf nodes shows that the meta-algorithm fails due to overflow with probability at most $\ell 2^\ell \cdot e^{-\frac{Z}{6}}$. ☐

Corollary 3. *Let λ be the security parameter and $N = \mathsf{poly}(\lambda)$ be the number of elements. The algorithm* mrsort *is a multiplicity-revealing sort w.r.t. the uniform distribution on the set $\mathcal{R} = \{0,1\}^\ell$.*

Proof. Note that ℓ is the largest integer such that $2^\ell \leq \lfloor 4N / \max\{\omega(\log \lambda), \Omega(\log N)\} \rfloor$. Hence, $Z \geq \max\{\omega(\log \lambda), \Omega(\log N)\}$, and then the procedure mrsort fails with probability at most $\mathsf{negl}(\lambda)$ by Lemma 5. It follows that mrsort fails on a negligible fraction of $\vec{\chi} \in \mathcal{R}$.

The correctness of mrsort follows from the above failure fraction and the correctness of Meta-BucketSort. To show the security, observe the access pattern of mrsort is fixed except for outputting real elements in each bucket, which can be efficiently and deterministically computed by the multiplicity of each key in $\vec{\chi}$. ☐

5 IO-Efficient Bucket Sort and Oblivious Random Permutation

5.1 Intuition

We describe an IO-efficient implementation of Algorithm 1. The technical meat of this section is how to resolve the non-determinism in the meta-algorithm (Algorithm 1), i.e., how do we order the work in the meta-binary-tree of Algorithm 1 such that we can achieve good cache efficiency? Here we are inspired by the Funnel Sort algorithm [27] that is a cache-agnostic, comparison-based sorting algorithm but not data oblivious.

Recall that Algorithm 1 describes a meta-binary-tree where each node produces a list of sorted buckets. Each leaf node is a list containing just a single bucket, whereas every parent node can be considered as the result of merge-sorting the two lists of buckets contained in its children. Henceforth a list of buckets is said to be sorted if every element in bucket $i+1$ is larger than every element in bucket i.

If we consider any "triangle" in this meta-binary-tree, there are two types of such triangles:

- Triangles built up from the leaf level: such a triangle basically sorts all of the input buckets (where each leaf node carries a single input bucket). Henceforth such a triangle from the leaf level up is nominally referred to as BucketFunnelSort.

  As a special case, the outer-most sort problem that we are trying to solve can be viewed as the triangle formed by the entire tree, and thus is an instance of BucketFunnelSort.

- Triangles that are internal to the tree (not including the leaf level): such a triangle takes as input several lists each containing multiple sorted buckets, and outputs a sorted list of buckets. Henceforth such an internal triangle is nominally referred to as BucketFunnelMerge.

In our detailed algorithm description below, the key insight is to express the order in which work is performed in a recursive form, using the notations BucketFunnelMerge and BucketFunnelSort to refer to different triangles in the meta-binary-tree.

5.2 Detailed Algorithm

Notations. For integer $t \geq 1$ and $0 \leq i < 2^t$, let $\mathsf{Binary}_t(i) \in \{0,1\}^t$ denote the t-bit binary representation of the integer i, with the most significant bit on the left.

Syntax of BucketFunnelSort and BucketFunnelMerge. Earlier we described how to view BucketFunnelSort and BucketFunnelMerge as triangles in the meta-binary-tree of Algorithm 1. We now define their syntax more precisely.

- BucketFunnelSort takes in 2^t lists each consisting of a single bucket (equivalently we also say that BucketFunnelSort takes in 2^t unsorted buckets), and outputs a single list of 2^t sorted buckets.

- BucketFunnelMerge, on the other hand, is parametrized by two variables r and t satisfying $r + t \leq \ell$ where ℓ is the bit-length of sort-key (i.e., the random key assigned to each element in our oblivious random permutation). BucketFunnelMerge takes in 2^r lists of sorted buckets; for $j \in [2^r]$, the jth list contains 2^t buckets $\{\mathsf{B}_i^{(j)} : i \in [2^t]\}$, where all real elements in bucket $\mathsf{B}_i^{(j)}$ have keys with prefix $\mathsf{Binary}_t(i)$. The output is a list of 2^{r+t} buckets $\{\widehat{\mathsf{B}}_i : i \in [2^{r+t}]\}$, where $\widehat{\mathsf{B}}_i$ contains all the real elements in the input whose keys have the prefix $\mathsf{Binary}_{r+t}(i)$.

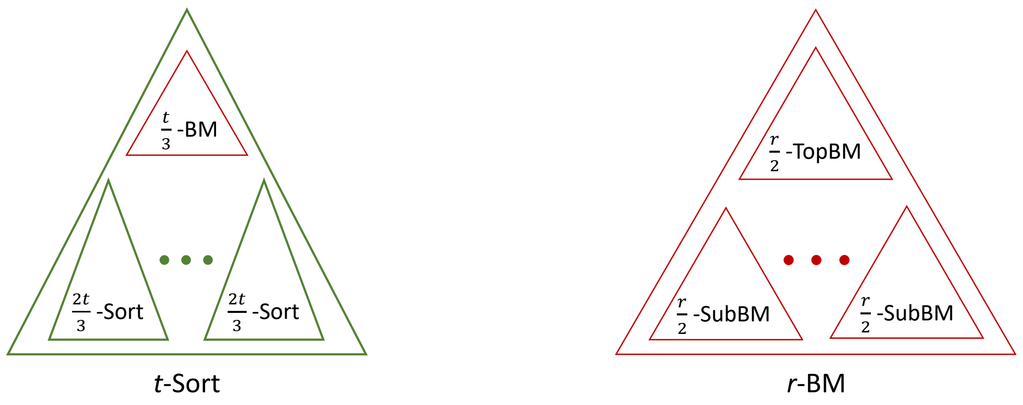


Figure 1: The recursive structure of BucketFunnelSort (left) and BucketFunnelMerge (right). Every **green** triangle is an instance of BucketFunnelSort (see Section 5.2.1), and every red triangle is an instance of BucketFunnelMerge (which is implemented by a BucketMerger data structure, see Section 5.2.2). A t-Sort has 2^t input streams and an r-BM has 2^r input streams. For simplicity, the figure ignores rounding issues, which are handled in detailed algorithm description.

5.2.1 Bucket Funnel Sort

In Algorithm 2, we describe how to recursively break a larger instance of BucketFunnelSort into smaller instances of BucketFunnelSort and an instance of BucketFunnelMerge. The recursion implicitly defines the order in which work in the meta-binary-tree of Algorithm 1 is performed. The algorithm is reminiscent of Funnel Sort but here we operate on the granularity of buckets (containing real and dummy elements) rather than individual elements — and as mentioned earlier, this is necessary for load balancing.

Algorithm 2 Bucket Funnel Sort

```
1: procedure BucketFunnelSort($\{\mathsf{B}_i : i \in [2^t]\}$)    // The input is a list of $2^t$ buckets. Recall that in the meta-algorithm, there is an FIFO queue for each node of the binary tree with $2^t$ leaves.
2:     if $t \leq 2$ then
3:         return output buckets by (oblivious) bitonic sort.
                               // The meta-algorithm can be implemented arbitrarily for the base case.
4:     Let $t = 3r + s$, where $0 \leq s \leq 2$.
5:     for $j$ from 0 to $2^r - 1$ do
6:         Define the list $\mathcal{L}^{(j)} := \{\mathsf{B}_i^{(j)} : i \in [2^{2r+s}]\}$, where $\mathsf{B}_i^{(j)} = \mathsf{B}_{j \cdot 2^{2r+s}+i}$.
7:         $\widehat{\mathcal{L}}^{(j)} \leftarrow$ BucketFunnelSort($\mathcal{L}^{(j)}$).    // Sequentially, each subtree of height $2r + s$ is recursively processed to produce the queue at the corresponding level-$(2r + s)$ node.
8:     return BucketFunnelMerge($\{\widehat{\mathcal{L}}^{(j)} : j \in [2^r]\}$)
                    // The $2^r$ queues for the nodes at level $2r + s$ are processed by BucketFunnelMerge.
```

The recursive structure of BucketFunnelSort is depicted in Figure 1 (left).

5.2.2 Bucket Funnel Merge

We now describe how to recursively break up the work needed by a BucketFunnelMerge instance. We will rely on a data structure called a BucketMerger to implement each BucketFunnelMerge instance.

A BucketMerger provides three operations, Init, Load, and Flush, where Init allocates space and is called exactly once upfront; Load connects the data structure with appropriate input lists; and finally Flush is an operation that can be called multiple times — every time it is called, a certain number of buckets from each of the input streams are consumed and a (partial) list of sorted buckets is produced. The details of the above operations are as follows:

- Init(r, t): This operation assigns enough memory for the bucket merger that is supposed to take 2^r input bucket lists, where each list can be viewed as an input stream. Each list should contain the same number $c \cdot 2^{2r}$ of buckets, which is a multiple of 2^{2r}. Only the parameter r is needed for memory allocation. The parameter t is saved and used later in the operation Flush. Moreover, there exists some k such that for all $j \in [2^r]$ and $i \in [c \cdot 2^{2r}]$ all the real elements in the i-th bucket in list $\mathcal{L}^{(j)}$ have keys with prefix $\mathsf{Binary}_t(k + i)$.
- Load$(\{\mathcal{L}^{(j)} : j \in [2^r]\})$: This operation connects the 2^r lists to the input streams of the bucket merger.
- Flush$()$: If the input streams still have remaining buckets, 2^{2r} buckets from each stream will be consumed by the bucket merger. The real elements in these 2^{3r} buckets will be distributed into 2^{3r} output buckets such that the keys of the real elements in each output bucket have the same length $t + r$ prefix; moreover, the output buckets are sorted according to the lexicographical order of this prefix. With respect to the meta-algorithm, each call of Flush$()$ corresponds to inserting 2^{3r} buckets into the queue of the corresponding level-$(r + t)$ node.

Algorithm 3 Bucket Funnel Merge

```
1: procedure BucketFunnelMerge( $\{\mathcal{L}^{(j)} := \{\mathsf{B}_i^{(j)} : i \in [2^t]\} : j \in [2^r]\}$ )    // The input is $2^r$ lists, each of which contains $2^t$ buckets.
2:     BM ← BucketMerger.Init$(r, t)$.
3:     BM.Load$(\{\mathcal{L}^{(j)} : j \in [2^r]\})$
4:     while the input streams of BM have remaining buckets do
5:         BM.Flush()                                              // return output buckets
```

Internal structure of bucket merger. We next describe the data structure of bucket merger. Recall that a bucket merger is initialized with Init(r, t), where 2^r is the number of input lists.

For the base case $r = 1$, the data structure just needs to have enough space to perform (oblivious) bitonic sort on elements from 2 buckets.

For the case $r \geq 2$, we let $p := \lfloor \frac{r}{2} \rfloor$ and $q := \lceil \frac{r}{2} \rceil$, and the data structure consists of the following:

- A 2^p number of sub-bucket mergers $\{\mathsf{SubBM}_j : j \in [2^p]\}$, each of which is supposed to receive 2^q of the 2^r input streams.
- For each of the sub-bucket merger SubBM_j, assign buffer space Buffer_j of 2^{3q} buckets to receive its output.
- One top bucket merger TopBM whose input streams take buckets from the output of the sub-bucket mergers.

The recursive structure of BucketFunnelMerge is depicted in Figure 1 (right).

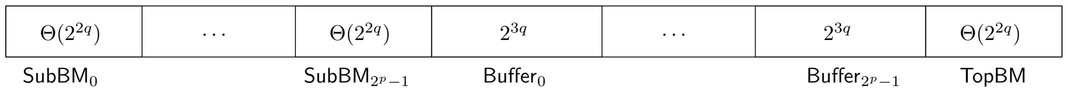


Figure 2: Procedure Init(r, t) allocates the memory layout of an (r, t)-BucketMerger, where $p := \lfloor \frac{r}{2} \rfloor$ and $q := \lceil \frac{r}{2} \rceil$. Buffer is an FIFO implemented with an array. SubBM and TopBM are further allocated recursively. SubBM, Buffer, and TopBM are allocated together to minimize IO-cost.

Algorithm. Algorithms 4, 5 and 6 describe the implementation for the above three operations of BucketFunnelMerge. The operations Init and Load are straightforward and follow the recursive structure of a bucket merger.

For the operation Flush of an (r, t)-bucket merger, observe that 2^{2r} from each of the 2^r input streams should be consumed. The intuition is to divide the merge problem to $\sqrt{2^r}$ sub-problems, where each sub-problem is recursively handled by a sub-bucket merger that consumes $\sqrt{2^r}$ input streams. Then, the output streams for the sub-bucket mergers are recursively merged by the top bucket merger. (The case when r is not an even number is handled carefully in Algorithm 6.)

Observe that space is allocated contiguously for each of the the top bucket merger and the sub-bucket merger. Hence, with respect to the binary tree in Algorithm 1, the nodes are allocated in van Emde Boas layout [57], which is a standard way to store a binary tree for IO efficiency. This gives an intuition for why the bucket merger has efficient IO-cost.

Algorithm 4 Bucket Merger Init (see also Figure 2)

```
1: procedure Init( $r, t$ )   // The parameter $t$ is not needed for space assignment, but the merger needs
   to save its value later for Flush.
2:     if $r = 1$ then
3:         Allocate enough space to perform bitonic sort on elements from 2 buckets.
4:         return
5:     Let $p := \lfloor \frac{r}{2} \rfloor$ and $q := \lceil \frac{r}{2} \rceil$.                    // $r \geq 2$
6:     for $j$ from 0 to $2^p - 1$ do
7:         SubBM$_j$ ← BucketMerger.Init$(q, t)$
8:         Buffer$_j$ ← assign space for $2^{3q}$ buckets
9:     TopBM ← BucketMerger.Init$(p, q + t)$
10:    return
```

Lemma 6. *An (r, t)-bucket merger occupies $\Theta(2^{2r} Z)$ memory, where Z is the bucket capacity.*

Proof. From Algorithm 4, we readily have the recurrence for the memory usage of an (r, t)-bucket merger:

$$S(r) = S(\left\lfloor \frac{r}{2} \right\rfloor) + 2^{\lfloor \frac{r}{2} \rfloor} \cdot (S(\left\lceil \frac{r}{2} \right\rceil) + 2^{3 \cdot \lceil \frac{r}{2} \rceil} \cdot Z),$$

with the base case $S(1) = \Theta(Z)$. Solving the recurrence gives $S(r) = \Theta(2^{2r} Z)$. □

Algorithm 5 Bucket Merger Load

```
1: procedure Load($\{\mathcal{L}^{(i)} : i \in [2^r]\}$ )    // The parameter $r$ is saved when the bucket merger is initialized.
2:     if $r = 1$ then
3:         The bucket merger is notified to process the buckets from the two input streams in
       subsequent Flush operations.
4:         return
5:     Let $p := \lfloor \frac{r}{2} \rfloor$. and $q := \lceil \frac{r}{2} \rceil$.    // $r \geq 2$
6:     for $j$ from 0 to $2^p - 1$ do
7:         SubBM$_j$.Load($\{\mathcal{L}^{(i)} : j \cdot 2^q \leq i < (j+1) \cdot 2^q\}$)
8:     return
```

Algorithm 6 Bucket Merger Flush (see also Figure 3)

```
1: procedure Flush( )    // The parameters $r$ and $t$ are saved when the bucket merger is initialized;
   each input stream is supposed to contain at least $2^{2r}$ buckets.
2:     if $r = 1$ then
3:         Repeat the following 4 times:
             • Take one bucket from each of the two input streams: $(\mathsf{B}_0, \mathsf{B}_1)$; all real elements
               from these two buckets have the same length-$t$ prefix in their keys.
             • output two buckets from MergeSplit($\mathsf{B}_0, \mathsf{B}_1, t$).    // Can throw Overflow
               exception if either bucket's capacity is exceeded.
4:         return
5:     Let $p := \lfloor \frac{r}{2} \rfloor$. and $q := \lceil \frac{r}{2} \rceil$.    // $r \geq 2$
6:     for $x$ from 1 to $2^{2r-2q}$ do    // Repeat $2^{2r-2q}$ times.
7:         for $j$ from 0 to $2^p - 1$ do
8:             Buffer$_j$ ← SubBM$_j$.Flush()
9:         TopBM.Load($\{\text{Buffer}_j : j \in [2^p]\}$)
10:        for $y$ from 1 to $2^{3q-2p}$ do    // Repeat $2^{3q-2p}$ times.
11:            TopBM.Flush()    // $2^{3p}$ buckets are output for each TopBM.Flush().
```

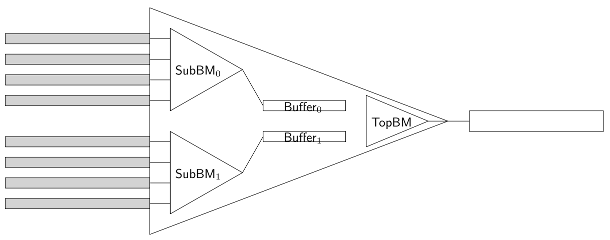
5.3 IO-Efficiency Analysis

We first consider the case when the cache size $M = \Omega(Z)$ is large enough. The case for small M is easier and will be handled later in Theorem 8.

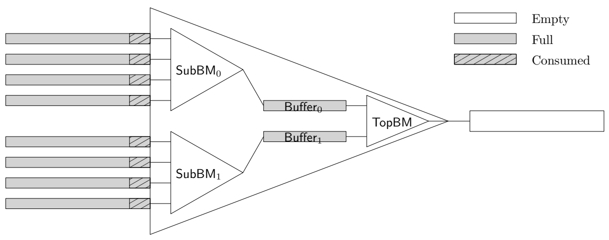
Lemma 7 (Base Case of Bucket Merger)**.** *Suppose r is an integer such that an $(r, *)$-bucket merger* BM *can fit into a cache of size $M \geq B^2$, but an $(2r, *)$-bucket merger cannot, where B is the size of a cache line. Then, one call of* BM.Flush() *has IO-cost $O(\frac{2^{3r} \cdot Z}{B})$, assuming optimal cache replacement policy.*

Proof. Denote $J := 2^r$. By the assumption on r, Lemma 6 for the size of a bucket merger implies that $J^2 Z \leq \Theta(M) \leq J^4 Z$.

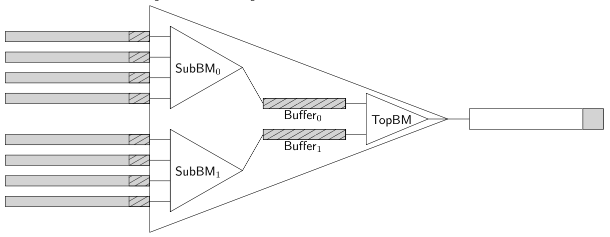
Since the space of the $(r, *)$-bucket merger BM resides in contiguous memory, loading the space allocated to BM into the cache has IO-cost $O(\frac{J^2 Z}{B})$. Loading the first bucket from each of J input streams has IO-cost $O(J \lceil \frac{Z}{B} \rceil)$, which is bounded by $O(\frac{J^3 Z}{B})$ by two cases: if $Z \geq B$, it holds directly; otherwise, $O(J \lceil \frac{Z}{B} \rceil) = O(J)$, by $\Theta(B^2) \leq \Theta(M) \leq J^4 Z$, we have $O(J) \leq O(\frac{J^3 Z}{B})$.



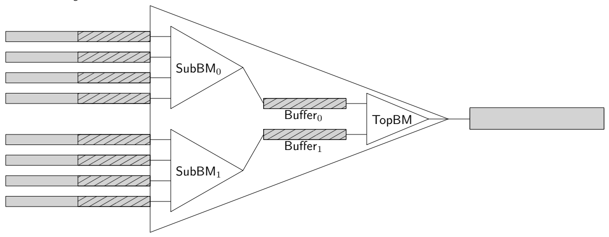
(a) Calling $\mathsf{Init}(r,t)$ sets $p = \lfloor \frac{r}{2} \rfloor$, $q = \lceil \frac{r}{2} \rceil$, and allocates SubBM_j, Buffer_j, and TopBM recursively.



(b) Calling $\mathsf{SubBM}_j.\mathsf{Flush}()$ for each j fills each Buffer_j.



(c) Repeating $\mathsf{TopBM.Flush}()$ for 2^{3q-2p} times consumes all contents in buffers.



(d) Repeating (b) and (c) 2^{2r-2q} times finishes $\mathsf{Flush}()$.

Figure 3: The recursive procedure $\mathsf{Flush}()$ on an $(r,*)$-$\mathsf{BucketMerger}$, where $r = 3$.

Next, observe that throughout the execution of $\mathsf{BM.Flush}()$, each memory location corresponding to the input streams and the output will be accessed at most once. Therefore, because of optimal cache replacement policy, the data structure for BM will remain in the cache during the execution of $\mathsf{BM.Flush}()$.

Hence, it suffices to analyze the IO-cost for reading the inputs and writing the output. There are J input streams, each of which has J^2 buckets; hence, the IO-cost is $J \cdot O(\left\lceil \frac{J^2 Z}{B} \right\rceil) = O(\frac{J^3 Z}{B})$. Similarly, J^3 buckets will be sent to a contiguous region in the memory, and hence has IO-cost $O(\frac{J^3 Z}{B})$. □

Lemma 8 (IO-Cost of Bucket Merger)**.** *Suppose r is an integer such that an $(r,*)$-bucket merger* BM *cannot fit into a cache of size $M \geq B^2$, where B is the size of a cache line. Then, one call of* $\mathsf{BM.Flush}()$ *has IO-cost $O(\frac{2^{3r} \cdot rZ}{B \log \frac{M}{Z}})$, assuming optimal cache replacement policy.*

Proof. Consider the procedure $\mathsf{BM.Flush}()$ described in Algorithm 6. Recall that $p := \left\lfloor \frac{r}{2} \right\rfloor$ and $q := \left\lceil \frac{r}{2} \right\rceil$.

Then, there are in total 2^{3p} calls to $\mathsf{SubBM.Flush}()$, which belong to $(q,*)$-bucket mergers. Moreover, there are in total 2^{3q} calls to $\mathsf{TopBM.Flush}()$, which belongs to a $(p,*)$-bucket merger.

Hence, we have the following recurrence for the IO-cost:

$$C_{\mathsf{m}}(r) \leq 2^{3p} \cdot C_{\mathsf{m}}(q) + 2^{3q} \cdot C_{\mathsf{m}}(p),$$

with the base case $C_{\mathsf{m}}(r_0) = O(\frac{2^{3r_0} \cdot Z}{B})$ from Lemma 7, when the size of an $(r_0,*)$-bucket merger can fit into the cache, i.e., $2^{2r_0} \cdot Z = \Theta(M)$.

Solving the recurrence gives $C_{\mathsf{m}}(r) = O(\frac{2^{3r} \cdot rZ}{B \log \frac{M}{Z}})$. □

Lemma 9 (IO-Cost of Bucket Funnel Sort)**.** *Given a list of 2^t buckets of capacity Z, the IO-cost of bucket funnel sort in Algorithm 2 is $O(2^t \cdot t \cdot \frac{Z}{B} \cdot \frac{1}{\log \frac{M}{Z}})$, assuming that $M \geq \Omega(B^2 + Z)$ and optimal cache replacement policy is used.*

Proof. Referring to Algorithm 2, we write $t = 3r + s$, where $r = \lfloor \frac{t}{3} \rfloor$ and $s = t \mod 3$. Then, the recursive structure of the algorithm readily gives the recurrence for the IO-cost of bucket funnel sort:

$$C_{\mathsf{s}}(t) \leq 2^r \cdot C_{\mathsf{s}}(2r + s) + 2^s \cdot C_{\mathsf{m}}(r),$$

where $C_{\mathsf{m}}(r)$ is the IO-cost of $\mathsf{Flush}()$ by an $(r, *)$-bucket merger, and the base case is $C_{\mathsf{s}}(t_0) = O(\frac{Z}{B})$ when $t_0 \leq 2$.

Using the bound for $C_{\mathsf{m}}(r)$ in Lemma 8, solving the recurrence gives $C_{\mathsf{s}}(t) = O(\frac{2^t \cdot tZ}{B \log \frac{M}{Z}})$. □

Theorem 8 (Performance of Bucket Sort)**.** *Assuming that $M \geq \Omega(B^2)$ and optimal cache replacement policy is used, there exists a cache-oblivious implementation of bucket sort of 2^t buckets of capacity Z that has IO-cost:*

- $O(2^t \cdot t \cdot \frac{Z}{B} \log^2 \frac{Z}{M})$*, if the cache size $M < \Theta(Z)$ cannot contain the workspace to perform bitonic sort in* $\mathsf{MergeSplit}$*;*
- $O(2^t \cdot t \cdot \frac{Z}{B} \cdot \frac{1}{\log \frac{M}{Z}})$*, if $M \geq \Theta(Z)$ is large enough.*

Moreover, the runtime is $O(2^t \cdot t \cdot Z \log^2 Z)$.

Proof. The runtime is analyzed by considering Algorithm 1, the meta-algorithm of bucket sort. Observe that the meta-algorithm calls MergeSplit for $O(2^t \cdot t)$ times, each of which calls bitonic sort on $\Theta(Z)$ elements a constant number of times. Hence, the bound on runtime follows readily.

In the cache-oblivious implementation, BucketFunnelSort, when $M \geq \Theta(Z)$ is large enough, the required bound on the IO-cost is given in Lemma 9.

If the cache size M is not large enough to contain the workspace needed for MergeSplit, then, in Algorithm 1, each bitonic sort has IO-cost $O(\frac{Z}{B} \log^2 \frac{Z}{M})$ from Lemma 12. Hence, it follows that any reasonable implementation of the meta-algorithm should have the desired IO-cost. □

5.4 Analysis of Memory Usage

Observe that for an input list of 2^t buckets, Algorithm 1 essentially calls MergeSplit for $O(2^t \cdot t)$ times. Hence, there is an in-place implementation that processes the nodes in the binary tree in increasing order of levels. However, this implementation does not have good IO-efficiency. The next lemma analyzes the memory usage of bucket funnel sort.

Lemma 10 (Memory Usage of Bucket Funnel Sort)**.** *On an input list of 2^t buckets of capacity Z, bucket funnel sort uses $O(2^t \cdot Z)$ memory.*

Proof. By Lemma 6, an $(r, *)$-bucket merger BM takes $O(2^{2r}Z)$ memory. Moreover, one call of BM.Flush() needs $O(2^{3r}Z)$ memory to write the output.

For $t \geq 3$, let $r = \lfloor \frac{t}{3} \rfloor$ and $s = t \mod 3$. Then, the recursive structure of Algorithm 2 gives the recurrence for the memory usage of bucket funnel sort:

$$S(t) \leq S(2r + s) + O(2^{3r}Z),$$

where the base case is $S(2) = O(Z)$. Solving the recurrence gives $S(t) = O(2^t \cdot Z)$. □

5.5 IO-Efficient Oblivious Random Permutation

Due to Section 4.2, our cache-agnostic, IO-efficient, Bucket Sort immediately gives rise to a cache-agnostic, IO-efficient Oblivious Random Permutation.

Specifically, we will instantiate the MetaORP algorithm described in Section 4.2 where mrsort can be realized by our IO-efficient Bucket Sort described in this section, and TrivialORP can be instantiated as in Lemma 3 of Section 3.2.

We thus obtain the following theorem.

Theorem 9 (Cache-Agnostic Oblivious Random Permutation)**.** *There exists a cache-agnostic, strongly oblivious random permutation algorithm with statistical security such that*

- *Assuming $M = \Omega(B^2)$, the algorithm permutes N elements in $O(N \log N (\log\log\lambda)^2)$ time, $O(N)$ space and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} (\log\log\lambda)^3)$ IO-cost.*
- *Assuming $M = \Omega(B^2)$, and $B = \Omega(\log^{0.51} N)$, the algorithm permutes N elements in $O(N \log N (\log\log\lambda)^2)$ time, $O(N)$ space and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ IO-cost.*

Proof. We consider MetaORP implemented with bucket sort and TrivialORP for the following cases. If $N \le Z$, then all elements fit into one bucket, and so the performance is asymptotically equivalent to performing bitonic sort on Z elements, which is implied directly from Lemma 12.

Otherwise, in MetaORP, observe that other than bucket sort, we assign random keys to each element, concatenate real elements after bucket sort, and apply random statistically oblivious permutation to elements in each of 2^ℓ buckets. By Corollary 3, each random key is an $O(\log N)$-bit binary string, which can be stored in constant words. Hence, the IO-cost is dominated by the IO-cost of bucket sort, which is given later in Theorem 8, observing that $2^\ell = O(\frac{N}{Z})$ gives the required bounds on the IO-cost. The runtime is also dominated by bucket sort because TrivialORP works on input size $Z = \mathsf{poly}\log\lambda$.

The failure probability is implied by plugging in failure probability of mrsort (Corollary 3) and TrivialORP (Lemma 3) into MetaORP in Lemma 4. □

6 From Oblivious Random Permutation to Oblivious Sort

Asharov et al. [7] observed that one way to construct (randomized) oblivious sort is through an oblivious random permutation as a stepping stone. The idea is to first obliviously and randomly permute the input array, and then apply any non-oblivious, comparison-based sorting algorithm. In our paper, we would like the non-oblivious, comparison-based sorting algorithm to also be cache-oblivious and small in IO-cost. We thus adopt the non-oblivious Funnel Sort by Frigo et al. [27]. For completeness, we formally state the observations made by Asharov et al. [7].

Recall that an oblivious random permutation is an algorithm that obliviously simulates a random permutation functionality henceforth denoted $\mathcal{F}_{\text{perm}}$, i.e., the functionality receives an array, chooses a random permutation and apply it on the elements of the array ("shuffling" it). The only allowed leakage is the length of the input.

Constructing oblivious sort from oblivious random permutation. Given an oblivious random permutation algorithm ORP and a non-oblivious comparison-based sorting algorithm (e.g., FunnelSort), one can easily construct an oblivious sorting algorithm as follows.

1. Given an input array X, let $Y := \mathsf{ORP}(X)$.
2. Now, sort Y using a (non-oblivious) comparison-based sorting algorithm. Formally speaking, a sorting algorithm is comparison-based if the physical access pattern depends only on the relative ranking of elements in the input. A technical condition we need is that no two

elements have the same rank, which can be ensured by resolving any tie consistently using the IDs of the elements, for instance.

If both the oblivious random permutation ORP and the non-oblivious sorting algorithm has good IO-efficiency, then so will the resulting oblivious sorting algorithm. Therefore, we will instantiate the above algorithm using our novel IO-efficient oblivious random permutation which we construct in Section 4, and using Funnel Sort as the non-oblivious sorting algorithm.

Lemma 11 (From ORP to oblivious sort)**.** *Suppose that* ORP *is a statistically (or perfectly resp.) strongly-oblivious random permutation algorithm and* sort *is a comparison based sorting algorithm, then the* $\mathsf{osort}(\cdot) := \mathsf{sort}(\mathsf{ORP}(\cdot))$ *is a statistically (or perfectly resp.) strongly-oblivious sorting algorithm.*

The proof of this Lemma is deferred to Appendix B.

Instantiation of oblivious sort. We instantiate oblivious sort using our oblivious random permutation (Theorem 9) and Funnel Sort [19,27].

The following theorem states the result of Funnel Sort.

Theorem 10 (Funnel sort, Theorem 6 in [19])**.** *Assuming $M = \Omega(B^2)$, funnel sort sorts N comparable elements in $O(N \log N)$ time, $O(N)$ space and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ IO-cost.*

We thus obtain the following corollary immediately.

Corollary 4 (Performance of cache-agnostic oblivious sort)**.** *There exists a cache-agnostic, strongly oblivious sorting algorithm with statistical security such that*

- *Assuming $M = \Omega(B^2)$, the algorithm sorts N elements by comparison in $O(N \log N (\log\log\lambda)^2)$ time, $O(N)$ space and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} (\log\log\lambda)^3)$ IO-cost.*

- *Assuming $M = \Omega(B^2)$, and $B = \Omega(\log^{0.51} N)$, the algorithm sorts N elements by comparison in $O(N \log N (\log\log\lambda)^2)$ time, $O(N)$ space and $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ IO-cost.*

7 Applications of Cache-Agnostic Oblivious Sort

7.1 Oblivious RAM

As mentioned, Oblivious RAM (ORAM) was originally proposed by Goldreich and Ostrovsky [28, 29]. In their seminal work, they showed a framework for constructing computationally secure ORAM schemes henceforth referred to as the *hierarchical ORAM* framework. Hierarchical ORAMs were later improved by several others [32,38,62]. The most essential building block in constructing hierarchical ORAMs is *oblivious sorting*; and thus our results on cache-agnostic oblivious sorting immediately gives rise to a *cache-agnostic* ORAM scheme whose asymptotical performance matches the best-known, cache-aware, external-memory ORAM scheme [32].

Below, we first give an overview of the hierarchical ORAM by Goodrich and Mitzenmacher [32] and improved later due to a better external-memory oblivious sorting algorithm by Goodrich [31] — the resulting construction by combining the two is the best known external-memory ORAM scheme (but is cache-aware). We then argue that by replacing their sorting algorithm with ours, we can obtain the same asymptotical result but in a cache-agnostic manner. Moreover, our ORAM algorithm is *strongly obivious* but the earlier construction [31,32] is only weakly oblivious and thus prone to a possible cache timing attack.

Background on hierarchical ORAM. We describe the hierarchical ORAM framework. Data is stored in memory in a hierarchical data structure containing $L := \log N + 1$ levels, where level $i \in \{0, 1, \ldots, L\}$ is a hash table that can store at most 2^i (real) elements. The original work by Goldreich and Ostrovsky [28,29] adopts an ordinary "balls-and-bins" hashing scheme; but Goodrich and Mitzenmacher [32] observed that this hash table can be replaced by a Cuckoo hash table to obtain tighter asymptotics. Upon a logical memory request for some logical address addr, the CPU looks for the element from level 0 all the way to level L. The CPU relies on a pseudorandom function to calculate $O(1)$ hash table locations in every level where the logical address addr is allowed to reside. Once the logical address addr is found in some level, for the remaining levels, the CPU visits a random set of hash table locations. Thus the read phase requires reading $O(\log N)$ random locations in memory.

The element being requested is removed from its original location during the aforementioned fetch phase. After the fetch phase, a maintain phase is necessary to reshuffle data. During the maintain phase, the fetched element (whose value may be updated if this is a write request), along with consecutive full levels $0, 1, \ldots, i$, are shuffled into the first empty level $i+1$, and the hash table in level $i+1$ is obliviously rebuilt. This oblivious rebuild operation is achieved through oblivious sorting. Specifically in Goodrich and Mitzenmacher's construction, to rebuild a hash table of size 2^i, we would need roughly $O(\log N)$ oblivious sorts, initially on an array of length 2^i, but the length of the array decreases exponentially with every oblivious sort. On average, each hash table of size 2^i is rebuilt every 2^i memory requests. Thus, on average every memory requests pays for

$$\log N \cdot \frac{\mathrm{OSort}(\lambda, N)}{N}$$

cost for oblivious sort, where $\frac{\mathrm{OSort}(\lambda,N)}{N}$ is the cost *per element* for each oblivious sort.

The following theorem is proven by Goodrich and Mitzenmacher [32].

Theorem 11 (Goodrich and Mitzenmacher ORAM [32])**.** *Assume that one-way functions exist. If there exists a (computationally or statistically) weakly-oblivious sorting algorithm that runs in time $T(n, \lambda, M, B)$ and IO-cost $C(n, \lambda, M, B)$ for sorting n elements where λ is the security parameter and M and B are the cache parameters, then, there exists a computationally secure, weakly-oblivious ORAM scheme such that for a logical memory of N words, each logical memory access requires $O(\log N + \frac{T(N,\lambda,M,B)}{N} \log N)$ time and $O(\log N + \frac{C(N,\lambda,M,B)}{N} \log N)$ IO-cost.*

Note that in the above theorem the $\log N$ term corresponds to the fetch-phase cost, and the $\frac{T(N,\lambda,M,B)}{N} \log N$ and $\frac{C(N,\lambda,M,B)}{N} \log N$ terms correspond to the maintain-phase costs that arise from oblivious sorting.

Corollary 5. *Assume that one-way functions exist. If there exists a cache-agnostic, (computationally or statistically) strongly-oblivious sorting algorithm that runs in time $T(n, \lambda, M, B)$ and IO-cost $C(n, \lambda, M, B)$ for sorting n elements where λ is the security parameter and M and B are the cache parameters, then, there exists a cache-agnostic, computationally secure, weakly-oblivious ORAM scheme such that for a logical memory of N words, each logical memory access requires $O(\log N + \frac{T(N,\lambda,M,B)}{N} \log N)$ time and $O(\log N + \frac{C(N,\lambda,M,B)}{N} \log N)$ IO-cost.*

Proof. Immediate due to Goodrich and Mitzenmacher [32]. Although their work does not care about being cache agnostic or strongly oblivious, it is not difficult to see that if they started with a cache agnostic and strongly oblivious sorting scheme, then their resulting ORAM construction would retain the same properties. $\square$

Now, combining Corollary 5 and Corollary 4, we immediately obtain the following theorem, where we obtain a cache-agnostic, strongly oblivious ORAM whose asymptotical IO-cost matches the best known cache-aware, weakly oblivious ORAM construction [31, 32].

Theorem 12 (Cache-agnostic ORAM). *Assume that one-way functions exist. There exists a computationally secure, strongly-oblivious ORAM scheme in the cache-agnostic model such that for a logical memory of $N \geq \lambda$ words, the ORAM consumes $O(N)$ space, and furthermore,*

- *Assuming $M \geq \Omega(B^2)$, then each logical memory request takes $O(\log^2 N (\log \log N)^2)$ time and $O(\log N + \frac{1}{B} \log N \log_{\frac{M}{B}} \frac{N}{B} (\log \log N)^3))$ IO-cost.*

- *Assuming $M \geq \Omega(B^2)$ and $B \geq \Omega(\log^{0.51} N)$, then each logical memory request takes $O(\log^2 N (\log \log N)^2)$ time and $O(\log N + \frac{1}{B} \log N \log_{\frac{M}{B}} \frac{N}{B}))$ IO-cost.*

Proof. Straightforward due to Corollary 5 and Corollary 4. □

7.2 Other Applications

Several earlier works [32, 34, 41, 45] pointed out that given oblivious sorting, we can compile any program expressed in a MapReduce [18] or GraphLab [2] abstraction (assuming a certain nice properties hold for the reduce or aggregation function) into an efficient oblivious counterpart that is asymptotically faster than generic ORAM compilation. These works [41, 45] also pointed out that many useful algorithms such as Page Rank [46] and matrix factorization can be expressed in efficiently in MapReduce and/or GraphLab abstractions. Therefore, our cache-agnostic oblivious sorting algorithm can also be leveraged to construct efficient oblivious algorithms for any algorithm that is efficiently expressible in MapReduce or GraphLab abstractions (assuming certain nice properties for the aggregate or reduce function).

8 Conclusion and Open Questions

We are the first to propose an external-memory, comparison-based oblivious sorting algorithm that has optimal IO-cost under the standard tall cache and wide cache-line assumptions; not only so, our construction is cache-agnostic in that the algorithm need not know the parameters of the storage architecture. Our results on sorting immediately gives rise to a new, *cache-agnostic* Oblivious RAM algorithm that matches the best known cache-aware result.

We also rethink security definitions for oblivious algorithms in the external-memory model. We point out a weakness in the security definition of earlier works on external-memory oblivious algorithms — thus making earlier algorithms vulnerable to a possible cache-timing attack. We thus propose a stronger notion of security and prove that our algorithms satisfy the stronger security notion. Thus our work lays a foundation for the study of new oblivious algorithms in the cache-agnostic model. We leave open several interesting questions:

- Can we construct a *deterministic*, external-memory oblivious sorting algorithm that is optimal in IO-cost?

- Can we construct a *deterministic*, *cache-agnostic* oblivious sorting algorithm that is optimal in IO-cost?

Acknowledgments

We are grateful to Ben Fisch and Dan Boneh for suggesting ORAM as a means of defending against cache-timing attacks. We thank Kartik Nayak and Ling Ren for suggesting the notion of cache obliviousness in the context of ORAM. Elaine Shi would like to thank Emil Stefanov for proposing Bucket ORAM [22] on January 21, 2014 whose core idea led to the bucket sort construction.

This work is supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a DARPA Safeware grant (subcontractor under IBM), a Sloan Fellowship, Google Faculty Research Awards, a Baidu Research Award, and a VMWare Research Award.

References

[1] Cache replacement policies. `https://en.wikipedia.org/wiki/Cache_replacement_policies`.

[2] Graphlab. `http://graphlab.org`.

[3] Alok Aggarwal and S. Vitter, Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[4] M. Ajtai, J. Komlós, and E. Szemerédi. An O(N Log N) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[5] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.

[6] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.

[7] Gilad Asharov, Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Oblivious computation with data locality. 2017.

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

[9] M. Bender, E. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. *SIAM Journal on Computing*, 35(2):341–358, January 2005.

[10] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[11] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 207–218. ACM, 2013.

[12] Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In *International Colloquium on Automata, Languages, and Programming*, pages 426–438. Springer, 2002.

[13] Gerth Stolting Brodal and Rolf Fagerberg. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[14] Hubert Chan and Elaine Shi. Circuit OPRAM: A unifying framework for computationally and statistically secure ORAMs and OPRAMs. `https://eprint.iacr.org/2016/1084.pdf`.

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

[16] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/2016/086`.

[17] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, August 2014. USENIX Association.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[19] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[20] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[21] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, pages 13–22, 2010.

[22] Christopher W. Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. *IACR Cryptology ePrint Archive*, 2015:1065, 2015.

[23] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *ASPLOS*, 2015.

[24] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[25] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[26] Robert W. Floyd. Permuting Information in Idealized Two-Level Storage. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 105–109. Springer US, 1972. DOI: 10.1007/978-1-4684-2001-2_10.

[27] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[28] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

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

[30] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14.

[31] Michael T. Goodrich. Data-oblivious External-memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 379–388, New York, NY, USA, 2011. ACM.

[32] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 576–587, 2011.

[33] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[34] Michael T Goodrich and Joseph A Simons. Data-oblivious graph algorithms in outsourced external memory. In *International Conference on Combinatorial Optimization and Applications*, pages 241–257. Springer, 2014.

[35] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP*, 2013.

[36] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX)*, Jun 2015. Reference no. 332680-002.

[37] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[38] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[39] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, March 2015.

[40] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 51–65, 2013.

[41] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015.

[42] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[43] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 13:10, 2013.

[44] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear i/o. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 723–735, London, UK, UK, 2002. Springer-Verlag.

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

[46] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, 1998.

[47] Rasmus Pagh and Srinivasa Rao Satti. Secondary indexing in one dimension: Beyond b-trees and bitmap indexes. In *Proceedings of the Twenty-eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '09, pages 177–186, New York, NY, USA, 2009. ACM.

[48] Harald Prokop. *Cache-Oblivious Algorithms.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[49] Sanguthevar Rajasekaran and Sandeep Sen. Optimal and practical algorithms for sorting on the pdm. *IEEE Trans. Comput.*, 57(4):547–561, April 2008.

[50] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. Cryptology ePrint Archive, Report 2014/997, 2014. `http://eprint.iacr.org/`.

[51] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[52] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, New York, NY, USA, 2009. ACM.

[53] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[54] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)*, 2013.

[55] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[56] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[57] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10(1):99–127, 1976.

[58] Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.*, 33(2):209–271, June 2001.

[59] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. Cryptology ePrint Archive, Report 2014/672, 2014.

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

[61] Zhewei Wei, Ke Yi, and Qin Zhang. Dynamic external hashing: The limit of buffering. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 253–259, New York, NY, USA, 2009. ACM.

[62] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security (CCS)*, pages 139–148, 2008.

[63] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[64] Ke Yi and Qin Zhang. On the Cell Probe Complexity of Dynamic Membership. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 123–133, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[65] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-oblivious mesh layouts. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 886–893, New York, NY, USA, 2005. ACM.

[66] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.

[67] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *CCS*, 2014.

Appendix

A Details of Some Simple Building Blocks

A.1 Bitonic Sort in the Cache-Agnostic Model

Bitonic sort [8] is a perfectly oblivious and in-place deterministic sorting algorithm. It takes an array of elements and a boolean flag, and sorts the elements in non-decreasing order if flag is true and in non-increasing order otherwise. Bitonic sort uses a subroutine bitonic merge, which is similar to bitonic sort, except that the input array needs to be bitonic. For completeness, we give the description of the algorithm. Since we use bitonic sort on smaller sub-instances (of the original problem of size N), we denote the size of the sub-problem by n, which we assume is a power of 2 for notational convenience.

Algorithm 7 Bitonic Sort

```
1: procedure BitonicSort($A[i..i+n)$, flag)    // The input is a sub-array of $A$, where $n$ is a power of 2; the
   procedure sorts the sub-array $A[i..i+n)$ in non-decreasing order if flag is true, and non-increasing otherwise.
2:     if $n = 1$ then                                                        // base case
3:         return
                                                                               // $n \geq 2$
4:     BitonicSort($A[i..i+\frac{n}{2}), \neg$flag);
5:     BitonicSort($A[i+\frac{n}{2}, i+n)$, flag);
6:     BitonicMerge($A[i..i+n)$, flag);
7:     return
```

Algorithm 8 Bitonic Merge

```
1: procedure BitonicMerge($A[i..i+n)$, flag)    // The input is a sub-array of $A$ that is
   bitonic, where $n$ is a power of 2; the procedure sorts the sub-array $A[i..i+n)$ in non-decreasing order if
   flag is true, and non-increasing otherwise.
2:     if $n = 1$ then                                                        // base case
3:         return
                                                                               // $n \geq 2$
4:     for $j$ from 0 to $\frac{n}{2} - 1$ do
5:         if $(A[i+j] > A[i+j+\frac{n}{2}]) \equiv$ flag then
6:             Swap($A[i+j], A[i+j+\frac{n}{2}]$);
7:     BitonicMerge($A[i..i+\frac{n}{2})$, flag);
8:     BitonicMerge($A[i+\frac{n}{2}, i+n)$, flag);
9:     return
```

Lemma 12 (IO-cost of Bitonic Sort)**.** *If n elements fit into the cache of size M, bitonic sort on n elements has IO-cost $O(\lceil \frac{n}{B} \rceil)$, where B is the size of a cache line; otherwise, it has IO-cost $O(\frac{n}{B} \log^2 \frac{n}{M})$.*

The special case $M = B = 1$ implies that the algorithm runs in time $O(n \log^2 n)$.

Proof. Since bitonic sort is an in-place sorting algorithm, if the n elements can fit into the cache, the IO-cost just comes from reading the input.

We next analyze the case when $n > M$. Referring to Algorithm 8, the recurrence for the IO-cost of bitonic merge is given by:

$$C_{\mathsf{m}}(n) \leq O(\frac{n}{B}) + 2 \cdot C_{\mathsf{m}}(\frac{n}{2}),$$

where the $O(\frac{n}{B})$ term comes from the linear scan in line 4 in Algorithm 8, and the base case is $C_{\mathsf{m}}(n_0) = O(\lceil \frac{n_0}{B} \rceil)$ for $n_0 \leq M$. Solving the recurrence gives $C_{\mathsf{m}}(n) = O(\frac{n}{B} \log \frac{n}{M})$.

Referring to Algorithm 7, the recurrence for the IO-cost of bitonic sort is given by:

$$C_{\mathsf{s}}(n) \leq 2 \cdot C_{\mathsf{s}}(\frac{n}{2}) + C_{\mathsf{m}}(n),$$

where the base case is $C_{\mathsf{s}}(n_0) = O(\lceil \frac{n_0}{B} \rceil)$ for $n_0 \leq M$. Solving the recurrence gives $C_{\mathsf{s}}(n) = O(\frac{n}{B} \log^2 \frac{n}{M})$.

□

A.2 MergeSplit Algorithm

We now spell out the MergeSplit building block consumed by our Meta-BucketSort algorithm (see Section 4.3). $\mathsf{MergeSplit}(\mathsf{B}_0, \mathsf{B}_1, t)$ is a data-oblivious subroutine, which takes two input buckets in which all real elements share a t-bit prefix in their keys, and redistributes the elements into two output buckets based on the $(t+1)$-st bit in the keys of real elements. Further, in each output buckets, the real elements appear before the dummy ones. This MergeSplit subroutine can be realized using $O(1)$ linear scans and a single bitonic sort:

1. In one linear scan, count the number of real elements that should go to each output bucket.
2. In a second linear scan, tag each element to indicate which bucket it should go to, i.e., tag real element with the $t+1$-st bit of its key, and tag appropriate number of dummy elements with 0 and 1 so that there will be exactly Z elements tagged with each tag.

   If the number of real elements that should go to one bucket exceeds Z, then throw the Overflow exception.
3. Bitonic sort the two buckets by their tags. If two elements have the same tag, then real elements should appear before dummies.

B Proof of Lemma 11

In Section 6, we describe how to construct oblivious sort using oblivious random permutation as a building block. Now we give the proof of Lemma 11.

Proof. Our goal is to prove that given a statistically (or perfectly resp.) oblivious random permutation algorithm ORP and a non-oblivious, comparison based sorting algorithm sort, there exists a simulator simulates the access pattern of $\mathsf{osort}(X) := \mathsf{sort}(\mathsf{ORP}(X))$.

Real. We consider the following real-world distribution:

$$\mathsf{Real}(X) := \big(\mathsf{sort}(Y), \quad \mathsf{Addr}^{\text{orp}}(X), \mathsf{Addr}^{\text{sort}}(Y)\big), \quad \text{where } Y = \mathsf{ORP}(X),$$

$\mathsf{Addr}^{\text{orp}}(X)$ and $\mathsf{Addr}^{\text{sort}}(\mathsf{ORP}(X))$ denote the addresses of applying ORP followed by applying sort on the output of ORP.

Hybrid 1. Let

$$\mathsf{Hyb}_1(X) := \Big(\mathsf{sort}(Y), \quad \mathsf{Sim}^{\text{orp}}(1^\lambda, |X|), \mathsf{Addr}^{\text{sort}}(Y)\Big), \quad \text{where } Y = \mathcal{F}_{\text{perm}}(X),$$

$\mathcal{F}_{\text{perm}}$ and $\mathsf{Sim}^{\text{orp}}$ are the ideal functionality and the simulator defined by oblivious random permutation. By the definition of oblivious random permutation $\mathsf{Real}(X)$ and $\mathsf{Hyb}_1(X)$ are statistically close (or identical, resp.).

Ideal. Let

$$\mathsf{Ideal}(X) := \Big(\mathcal{F}_{\text{sort}}(X), \quad \mathsf{Sim}^{\text{orp}}(1^\lambda, |X|), \mathsf{Addr}^{\text{sort}}(\mathcal{F}_{\text{perm}}(1, 2, \ldots, |X|))\Big).$$

To show that $\mathsf{Hyb}_1(X)$ and $\mathsf{Ideal}(X)$ are identitcal, observe that $\mathsf{sort}(\mathcal{F}_{\text{perm}}(X))$ and $\mathcal{F}_{\text{sort}}(X)$ are identical. In addition, the outputs in both Hyb_1 and Ideal depends only on X, which implies that they are independent from addresses. Observe that sort is comparison-based, $\mathsf{Addr}^{\text{sort}}$ depends only on the relative order of input array. It follows that $\mathsf{Addr}^{\text{sort}}(\mathcal{F}_{\text{perm}}(X))$ and $\mathsf{Addr}^{\text{sort}}(\mathcal{F}_{\text{perm}}(1, 2, \ldots, |X|))$ are identical because the ordering of $\mathcal{F}_{\text{perm}}(X)$ and $\mathcal{F}_{\text{perm}}(1, 2, \ldots, |X|)$ have identical distribution. To complete, the simulator of osort is $\mathsf{Sim}^{\text{orp}}(1^\lambda, |X|), \mathsf{Addr}^{\text{sort}}(\mathcal{F}_{\text{perm}}(1, 2, \ldots, |X|))$. □