Oblivious Priority Queue and Single-Source Shortest Path in the External Memory Setting

Arya Maheshwari Elaine Shi

aryam@cs.cmu.edu elaineshi@cmu.edu

Carnegie Mellon University

Abstract

The study of oblivious algorithms is concerned with designing privacy-preserving algorithms whose memory access patterns reveal nothing about the secret inputs. Such algorithms have been deployed at scale in production systems, most notably in Signal's private contact discovery service. So far, all practical implementations of oblivious algorithms (e.g., those by Signal and Meta) rely on trusted hardware and operate within the external-memory model of computation. While it is known how to *generically* compile an arbitrary program to execute obliviously on an external-memory target machine, such generic oblivious simulations trade asymptotical efficiency for generality and therefore are rarely used in practice. Instead, customized oblivious algorithms tailored for the computational tasks of interest are almost always favored.

In this paper, we explore the single-source shortest path (SSSP) problem, a fundamental algorithmic building block with broad applications in scheduling, routing, graph mining, resource allocation and flow optimization. We present an external-memory oblivious SSSP algorithm for undirected graphs that achieves I/O efficiency $O(V + \frac{E}{B} \log \frac{E}{M})$ and total work $O(E \log E)$ assuming $E = \Omega(V)$, where V denotes the number of vertices, E denotes the number of edges, and M and B represent the target machine's cache size and block size, respectively. Our algorithm almost matches the best known non-private external-memory algorithm for SSSP, up to a $\log \log E$ factor in the second term of the I/O bound. The remaining $\log \log E$ gap is conjectured to be an inherent barrier, since making the underlying priority queue oblivious requires an $\Omega(\log \log n)$ blowup in I/O cost, which is known to be inherent.

As a by-product, we develop an improved external-memory oblivious priority queue that supports **DecrKey** operations. Specifically, while the construction of Jafargholi et al. attains optimal I/O efficiency, it is suboptimal in total work under a strong notion of obliviousness—where the adversary can observe both block-level and word-level accesses. This stronger security guarantee is the current industry norm and explicitly required by companies such as Signal. We present a new oblivious priority queue that achieves optimality in both dimensions. Specifically, we achieve an I/O cost of $O(\frac{1}{B} \log \frac{n}{M})$ and total work $O(\log n)$ per query where n is the capacity of the priority queue.

Contents

1 Introduction **2**
- 1.1 From the Word-RAM to the External-Memory Model . . . 2
- 1.2 The Price of Generic Oblivious Simulation . . . 3
- 1.3 Our Results and Contributions . . . 4
- 1.4 Technical Highlights . . . 7
- 1.5 Additional Related Work . . . 9
- 1.6 Organization of the Paper . . . 9

2 Computational Model and Oblivious Simulation **9**
- 2.1 External-Memory Model . . . 10
- 2.2 Oblivious Simulation . . . 10
- 2.3 Definitions: Oblivious Priority Queue . . . 12
- 2.4 Definitions: Oblivious Single-Source Shortest Path . . . 13

3 Existing Oblivious Building Blocks **13**
- 3.1 Building Blocks Equivalent to Sorting . . . 13
- 3.2 Building Blocks Equivalent to Compaction . . . 14
- 3.3 Oblivious Hash Table for Non-Recurrent Lookups and Shuffled Inputs . . . 15

4 Optimal Strongly Oblivious Priority Queue **16**
- 4.1 Oblivious Deduplication for Randomly Shuffled Inputs . . . 16
- 4.2 Overview of External-Memory Oblivious Priority Queue . . . 17
- 4.3 Oblivious Implementation Bucket-Level Operations . . . 19

5 Oblivious Adjacency List Iterator **20**
- 5.1 Syntax . . . 21
- 5.2 Security Definition . . . 22
- 5.3 Construction . . . 24
- 5.4 Efficiency . . . 26

6 Oblivious Single-Source Shortest Path **26**
- 6.1 Additional Preliminaries: Meta-Algorithm for SSSP . . . 26
- 6.2 Overview of Our Oblivious SSSP Algorithm . . . 28
- 6.3 Detailed Construction of Oblivious SSSP . . . 29

A Obliviously Simulating an External-Memory Machine **35**

B Proofs for Oblivious SSSP **36**
- B.1 Security Proofs . . . 36
- B.2 Efficiency . . . 39

C Proofs for Oblivious Adjacency List Iterator **41**
- C.1 Correctness and Security Proofs . . . 41
- C.2 Efficiency . . . 43

D Proofs for Oblivious Deduplication **44**

1 Introduction

Data-oblivious algorithms [Gol87, GO96] (or oblivious algorithms for short) represent a class of privacy-preserving algorithms in the Random Access Machine (RAM) model whose memory access patterns reveal no information about the secret inputs. Recently, several industry leaders such as Meta [met] and Signal [sig] implemented or deployed oblivious algorithms in production systems. For example, Signal's private contact discovery service relies fundamentally on oblivious algorithms to ensure that its access patterns to the encrypted user database reveal nothing about a user's contacts.

1.1 From the Word-RAM to the External-Memory Model

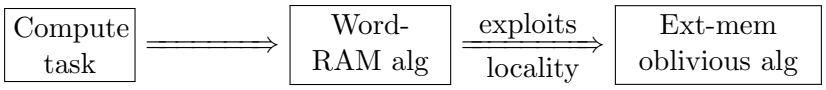
The texbook word-RAM model is a poor fit for hardware enclaves. Earlier, the study of oblivious algorithms [GO96, Gol87] was in part motivated by applications in storage outsourcing [RFK+15, SS13] and secure multi-party computation [GKK+12, LWN+15]. In these settings, it is customary to focus on the standard *word RAM* model — the model of computation implicitly adopted by most undergraduate algorithms textbooks. In the word RAM model, cost is typically measured by the number of bytes or *words* transferred between the CPU and memory, which also coincides with the running time of the algorithm. Unfortunately, the word RAM model turns out to be a poor fit for real-world deployments of oblivious algorithms that rely on hardware enclaves. In particular, the hardware-enclave execution model more closely matches the *external-memory* model [AV88], in which data transfers occur at *block* granularity rather than *word* granularity. That is, even if the CPU needs to access only a single word, it must nevertheless fetch the entire corresponding block from external memory. Specifically in the context of hardware enclaves, a block corresponds to a memory page, typically 4KB in size. Since the enclave is provisioned with only a limited amount of private cache, fetching (encrypted) data from outside the enclave triggers a page swap, which is serviced by the untrusted operating system. Since the page swap is a heavy-weight operation that involves encryption, decryption, and possibly context switch overhead, we care about minimizing not only the *total work* (i.e., computational overhead), but also the number of page swaps henceforth referred to as the *I/O cost*, following standard terminology [AV88].

With this model in mind, the classical line of work on external-memory algorithms [AV88, Dem02] focuses on designing algorithms that effectively exploit locality of access. In particular, once a block is fetched into the cache, it is desirable to perform as much useful work as possible before the block is evicted.

Oblivious algorithms for the external-memory model. Motivated by the aforementioned textbook-practice mismatch, it is important to study oblivious algorithms design for the external-memory model.

An external-memory machine, henceforth denoted $\mathsf{RAM}[N, M, B]$ is characterized by three parameters, where N denote the size of the memory tape, $M \geq B$ denotes the cache size, and B denotes the block size, all measured in *words*. The special case when $M = B = \Theta(1)$ corresponds to the *word RAM* model in which case the work and I/O cost coincide, collectively referred to as the *running time*. Some works on storage outsourcing [GMOT12b, GMOT12a] also explored the case where $M \geq B = \Theta(1)$ since the client device (e.g., laptop or smart phone) typically has larger local memory.

In the most general form, the design of oblivious algorithms is concerned with obliviously simulating an original (non-private) machine $\mathsf{RAM}'[N', M', B']$ on a target machine $\mathsf{RAM}[N, M, B]$.



(a) Approach 1: external-memory oblivious simulation of a word RAM



(b) Approach 2: external-memory oblivious simulation of an external-memory RAM

Figure 1: Generic oblivious simulation

1.2 The Price of Generic Oblivious Simulation

Generic external-memory oblivious simulation: sacrifice performance for generality. One naïve approach to designing external-memory oblivious algorithms is through generic oblivious simulation, which allows us to compile an arbitrary non-private algorithm to an oblivious counterpart. When the target machine is an external-memory machine, there are two general approaches of oblivious simulation (see also Figure 1):

1. *External-memory oblivious simulation of a word RAM.* The first approach is to design an insecure algorithm in the textbook word RAM model, and then obliviously simulate the word-RAM algorithm on an external-memory machine. Intuitively, this approach exploits the locality in the oblivious simulation itself to minimize the I/O cost, since the original algorithm is not necessarily designed with locality in mind.

   Asharov et al. [AEKL25] showed that if $M \geq B \geq \log^{1+\epsilon} \lambda$ where λ is the security parameter, we can obliviously simulate a word RAM with space N and running time T, incurring $O(T \cdot \log_B\left(\frac{N}{M}\right))$ I/O cost and $O(T \cdot B^\epsilon \cdot \log N \cdot \log_B N)$ total work (for any constant $\epsilon > 0$). This I/O bound is known to be optimal (if we insist on the generality) due to the lower bound of Komargodski and Lin [KL21].

2. *External-memory oblivious simulation of an external-memory RAM.* The second approach is to first design a non-private algorithm in the external-memory model, and then obliviously simulate the resulting algorithm on an external-memory machine. Unlike the first approach, this method seeks to exploit any inherent locality in the original computational task itself, rather than focusing solely on optimizing the oblivious simulation.

   More specifically, one can extend the result of Asharov et al. [AKL+22], and show the following: an external-memory $\mathsf{RAM}'[N, M, B]$ that consumes T number of I/Os and $W \geq T$ total work can be obliviously simulated on $\mathsf{RAM}[O(N), O(M), B]$, incurring an I/O cost of $O(T \cdot \log \frac{N}{M})$ and total work $O(W \cdot \log M + T \cdot \log \frac{N}{B})$, as long as $M \geq \log^c \lambda$ for some suitable constant $c > 1$. Further, both the I/O cost and work are optimal [KL21]. See Section A for a more detailed explanation.

Generic oblivious simulation, however, sacrifices asymptotical efficiency in exchange for generality. As mentioned above, generic oblivious simulation incurs upto logarithmic multiplicative overhead relative to the non-oblivious baseline. Moreover, such blowup is inherent if we insist on the generality due to known lower bounds [GO96, LN18, KL21]. The holy grail question in this line of work is whether we can design oblivious algorithms whose performance (nearly) matches the non-oblivious baseline — for specific computational tasks, this dream is not necessarily ruled out by the generic oblivious simulation lower bounds [GO96, LN18, KL21]. In this paper, we take a step

toward this objective by demonstrating such results for priority queues and single-source shortest path.

We emphasize that practitioners also care about designing customized oblivious algorithms tailored to the computational task at hand. For example, Signal adopts a customized oblivious map rather than generic oblivious simulation of a non-oblivious map [sig]. Open-source efforts such as Oblivious STL [ost] aim to provide oblivious counterparts of rich algorithmic building blocks and data structures commonly found in standard libaries such as C++'s STL or Rust's `std`.

External-memory oblivious algorithms design: a nascent field. Unfortunately, unlike the vast classical literature on (non-private) external-memory algorithms, our understanding of oblivious external-memory algorithms remains limited [LSX19,RS21,JLS21]. To date, known results cover only a handful of computational tasks. Shi and Ramachandran [RS21] showed that sorting n elements can be accomplished with $O(n \log n)$ work and $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ I/O cost, both of which are known to be optimal [AV88, FHLS19]. Jafargholi et al. [JLS21] constructed an oblivious priority queue of capacity n where each operation incurs an I/O cost $O(\frac{1}{B} \log \frac{n}{M})$, which is asymptotically optimal due to the lower bound of Jacob et al. [JLN19]. Although a somewhat broader line of work has explored the design of oblivious algorithms for various computational tasks on a *word RAM* [BSA13, WNL+14, LWN+15], naïvely adapting these constructions to an external-memory machine [TGS23] is unlikely to achieve optimal performance.

1.3 Our Results and Contributions

Given the status quo and the pressing real-life demand (e.g., Oblivious STL [ost]), there is an urgent need to develop optimal external-memory oblivious algorithms for a broad class of fundamental algorithmic primitives and data structures. Our work makes the following new contributions.

Priority queue, revisited. We begin by revisiting the external-memory oblivious priority queue. Although the construction of Jafargholi et al. [JLS21] achieves optimality in I/O cost [JLN19], it fails to achieve optimality in total work when a strong notion of obliviousness is required. Specifically, on an external-memory target machine (e.g., a server equipped with trusted hardware), the adversary (e.g., the operating system on the server) can not only observe which blocks are being requested [XCP15], but also which words within a block are being accessed [RTSS09, DMWS12]. A *strongly oblivious* algorithm [LSX19] (also referred to as *doubly oblivious* in some literature [MPC+18]) guarantees that neither block-level nor word-level accesses reveal any information about the secret inputs (see Section 2.2 for details). We emphasize that strong obliviousness is the current industry norm, demanded in real-world deployments of oblivious algorithms such as in Signal's implementation.

We propose a new oblivious priority queue that achieves optimality in both metrics, as stated in the following theorem:

Theorem 1.1 (External-memory oblivious priority queue)**.** *Assume the existence of pseudorandom functions and that $M \geq \log^c \lambda$ for some suitable constant $c > 1$. Then, there exists a cache-agnostic, strongly oblivious priority queue supporting* **ExtractMin**, **FindMin**, **Insert**, **DecrKey**, *and* **Delete** *operations, where each operation incurs an amortized I/O cost of $(\frac{1}{B} \log \frac{n}{M})$ and amortized work $O(\log n)$, where n is a parameter denoting the maximum capacity of the priority queue.*

Table 1: Comparison of our oblivious priority queue with known baselines. n denotes the maximum capacity of the priority queue. We abuse the notation $\omega(1)$ to denote an arbitrarily small super-constant function in n. For simplicity, assume that the the hash function in Jafargholi et al. [JLS21] is implemented with a PRF. For Jafargholi et al. [JLS21] and our work, we assume that a PRF evaluation is unit cost.

Scheme	I/O cost	Total work	Strongly oblivious?
Jacob et al. [JLN19] (lower bound)	$\Omega\left(\frac{1}{B}\log\frac{n}{M}\right)$	$\Omega(\log n)$	–
Jafargholi et al. [JLS21]	$O\left(\frac{1}{B}\log\frac{n}{M}\right)$	$O(\log n)$	×
Shi [Shi20] + van Emde Boas layout [vEBKZ76]	$O\left(\log_B\frac{n}{M}\right)$	$\log n \cdot \omega(1)$	✓
Ours	$O\left(\frac{1}{B}\log\frac{n}{M}\right)$	$O(\log n)$	✓

Above, we say that an algorithm is *cache-agnostic*[1] if the algorithm is unaware of the architecture-specific parameters M and B. In other words, the same algorithm works universally for all architectures. Further, if the algorithm achieves optimality in I/O cost, then on a multi-level storage architecture, optimal I/O is achieved at every level of the storage hierarchy.

Theorem 1.1 is optimal in both I/O cost and total work when strong obliviousness is required. Further, it nearly matches the best known (non-private) external-memory priority queue [JL19], leaving only a $\log\log n$ multiplicative gap in the I/O bound. This $\log\log n$ gap is as an inherent price to pay for (even the weaker notion of) obliviousness, since oblivious priority queues are subject to the aforementioned lower bound of Jacob et al. [JLN19]. We also note that the pseudorandom function (PRF) assumption in Theorem 1.1 (and our later results using this oblivious priority queue) stems from the use of a PRF in the oblivious hash table construction of Asharov et al. [AKL+20] that we use in our construction (see Section 1.4 and Theorem 3.1 for details). Like Asharov et al. [AKL+20] and prior work on computationally secure ORAM (starting from Goldreich and Ostrovsky [GO96]), we assume that a single PRF evaluation can be done in unit cost in terms of total work.[2] In particular, our oblivious priority queue requires amortized $O(\log n)$ PRFs per operation, but this is subsumed by the $O(\log n)$ amortized work otherwise incurred.

We summarize the comparison of Theorem 1.1 with baselines for oblivious priority queues (including the lower bound) in Table 1. In the table we state the work achieved by the exact construction described by Jafargholi et al. [JLS21], which does *not* satisfy strong obliviousness. We discuss straightforward modifications to make their construction strongly oblivious in Section 1.4, where we emphasize that these modifications lead to sub-optimal total work. Our oblivious priority queue, which instead builds on the construction of Jafargholi et al. [JLS21] via new primitives, is presented in Section 4, where we prove Theorem 1.1 as Corollary 4.5.

Single-source shortest path. Single-source shortest path (SSSP) [Dij59] is a fundamental graph algorithm that has numerous applications in graph optimization, GPS and mapping systems, airline and railway scheduling, and so on. We consider the SSSP problem for undirected graphs with non-negative edge weights. Below we use V to denote the number of vertices and E to denote the number of edges of the graph. Absent any privacy requirement, it is known how to achieve SSSP

[1] The classical algorithms literature uses the term "cache-oblivious" [Dem02]. We replace it with "cache-agnostic" as in [RS21] to avoid overloading the word "oblivious."

[2] Like Asharov et al. [AKL+20], we also assume PRF evaluations incur no I/O cost.

with an I/O cost of $O(V + E \cdot \mathsf{IO}_{\mathrm{pq}}(E))$ [KS96], where $\mathsf{IO}_{\mathrm{pq}}(E)$ denotes the per-query I/O cost of a priority queue of capacity E. Plugging in the recent priority queue of Jiang and Larsen [JL19], the resulting SSSP algorithm achieves an I/O cost of $O(V + \frac{E}{B} \log \frac{E}{B} / \log\log E)$.

To the best of our knowledge, our work is the first to study how to design an oblivious SSSP algorithm in the external-memory model. We devise a strongly oblivious, cache-agnostic algorithm that *almost matches the best known non-private I/O bound*, up to a $\log\log E$ multiplicative factor. This $\log\log E$ gap arises from the inherent price to pay for obliviousness in the underlying priority queue. Therefore, although there is no known oblivious lower bound for SSSP, we foresee barriers towards overcoming this $\log\log E$ gap. In particular, any approach towards solving SSSP that requires E queries to a priority queue must suffer from this extra $\log\log E$ factor (relative to non-private) or use a fundamentally different algorithm. Our result is summarized in the following theorem:

Theorem 1.2 (External-memory oblivious SSSP; see Theorem 6.2)**.** *Assume the existence of a pseudorandom function and the standard tall cache assumption that $M \geq B^2$ and $M \geq \log^c \lambda$ for some suitable constant $c > 1$. There exists a strongly oblivious, cache-agnostic SSSP algorithm for undirected, non-negative edge weight graphs that achieves an I/O bound of $O\left(V + \frac{E+V}{B}\left(\log \frac{E}{M} + \log_{\frac{M}{B}} \frac{E+V}{B}\right)\right)$, and total work $O((E+V)\log(E+V))$.*

In the standard regime of $E = \Omega(V)$, the bounds in Theorem 1.2 simplify to $O(V + \frac{E}{B} \log \frac{E}{M})$ I/O cost and $O(E \log E)$ total work. In Table 2, we compare our results with some baseline approaches including

1. Generic oblivious simulation [AEKL25] of a word-RAM SSSP algorithm [FT87];
2. Generic oblivious simulation [AKL+22] of the best known external-memory SSSP algorithm [JL19] (see also Section A); and
3. Naïvely implementing on an external-memory machine the best known oblivious SSSP algorithm designed for a word RAM — here, we apply the standard van Emde Boas layout [vEBKZ76] to the ORAM tree used in Appan et al. [AHR24].

Table 2: Comparison of our oblivious SSSP algorithm with known baselines. For clarity, we assume $E = \Omega(V)$ when expressing the bounds.

Scheme	I/O cost	Total work
EM-OSim of SSSP [AEKL25] + [FT87]	$\log_B(E/M) \cdot O(E + V \log V)$	$O(B^\epsilon \cdot \log E \cdot \log_B E \cdot (E + V \log V))$
OSim of EM-SSSP [AKL+22] + [JL19]	$\log \frac{E}{M} \cdot O\left(V + \frac{E}{B \log\log E} \log \frac{E}{B}\right)$	$\log \frac{E}{B} \cdot O\left(V + \frac{E \log(E/B)}{B \log\log E}\right) + \log M \cdot O(V + E \log E)$
Word-RAM O-SSSP [AHR24] + [vEBKZ76]	$O(E \log_B E)$	$O(E \log E)$
Ours	$O(V + \frac{E}{B} \log \frac{E}{M})$	$O(E \log E)$
Non-private [JL19]	$O(V + \frac{E}{B \log\log E} \log \frac{E}{M})$	$O(E \log E)$

Additional results. As a by-product and to help contextualize our main results, we extend prior work [AKL+22] to support the oblivious simulation of an arbitrary external-memory algorithm on an external-memory machine (see Approach 2 of Figure 1). This complements the recent work of Asharov et al. [AEKL25] which obliviously simulates a word RAM on an external-memory target machine (see Approach 1 of Figure 1). Intuitively, Asharov et al. [AEKL25] exploits the locality *within* the oblivious simulation algorithm itself to improve the I/O efficiency, but their approach cannot leverage locality in an application-aware manner. In contrast, our approach starts with an external-memory algorithm that exploits locality in a problem-specific manner, so that the oblivious simulation itself need not worry about locality.

For most computational tasks that admit efficient external-memory algorithms (i.e., problems that intrinsically exhibit ample locality), our generic compilation typically yields better asymptotic I/O bounds than that of Asharov et al. [AEKL25] — see Table 2 for an example. In contrast, the approach of Asharov et al. [AEKL25] is more suitable for problem with little to no intrinsic locality, where it is preferable to exploit locality within the oblivious simulation itself.

1.4 Technical Highlights

1.4.1 Techniques for the Priority Queue

To achieve Theorem 1.1, we build upon the algorithmic framework of Jafargholi et al. [JLS21], which maintains a binary tree of buckets, each of size roughly M. Although their construction achieves optimal I/O complexity, it fails to achieve optimality in total work since the bucket-level operations would incur an extra $\log M$ or $\operatorname{poly}\log M$ multiplicative factor if strong obliviousness is desired. Specifically, the buckets need to support either a hash table or a deduplication procedure which can be realized through sorting. A straightforward oblivious realization of hashing or sorting would require polylogarithmic work for each lookup, and $M \log M$ work for sorting M elements. On the other hand, achieving optimality in total work for the priority queue requires constant-work hash table lookups and linear-work deduplication.

Fortunately, we can draw on the core techniques behind the optimal Oblivious RAM (ORAM) construction recently proposed by Asharov et al. [AKL+20]. Our idea is to maintain the invariant that before and after we operate on a bucket, the bucket's contents are always randomly shuffled using coins unknown to the adversary. Asharov et al. [AKL+20] observed that for randomly shuffled inputs of polylogarithmic size or larger, one can build a hash table in linear work, with each lookup requiring only $O(1)$ work, plus some additional techniques to handle an $O(\log \lambda)$-sized stash.[3] Leveraging this oblivious hash table, we show in Section 4.1 how to realize oblivious deduplication for randomly shuffled inputs in linear work, provided that the input size is at least polylogarithmic.

We then use this oblivious deduplication procedure to implement the bucket-level operations (see Section 4.3) , thereby achieving optimal total work while respecting strong obliviousness. Finally, we adjust the parameters of the construction to additionally satisfy the cache-agnostic property.

1.4.2 Techniques for SSSP

We observe that some naïve approaches for SSSP fail to achieve our desired asymptotics.

Challenge 1: Obliviously emulating Dijkstra's algorithm is inherently costly. The first strawman idea is to attempt to obliviously emulate a known (non-private) SSSP algorithm designed

[3]The stash is a (random) logarithmic-sized subset of elements that the hash table does not store in its main data structure and instead expects the user handle separately; see the description of the oblivious hash table building block in Section 3.3 for details.

for the word-RAM model (e.g., the textbook Dijkstra's algorithm [Dij59]), by replacing its internal data structures with external-memory oblivious implementations. However, this approach suffers from a fundamental limitation since it requires $\Theta(E)$ random accesses to a visited array that records if each vertex has been expanded. Obliviously implementing this visited array will incur at least $\Omega(E \cdot \log \frac{V}{M} / \log B)$ I/O cost due to the lower bound of Komargodski and Lin [KL21].

Instead, our starting point is the (non-private) external-memory SSSP algorithm of Kumar and Schwabe [KS96], which we review in detail in Section 6.1. Kumar and Schwabe's algorithm relies on an elegant two-priority-queue trick to avoid the $\Theta(E)$ random accesses to the visited array. We emphasize, however, that simply substituting priority queues with oblivious counterparts [JLS21] does not directly yield an oblivious algorithm. We have to tackle two additional challenges:

Challenge 2: Oblivious accesses to adjacency list. The main challenge lies in obliviously accessing the adjacency list without leaking private information about the graph. The adjacency list cannot be simply placed in an ORAM due to the known external-memory ORAM lower bound [KL21]. The non-private algorithm of Kumar and Schwabe [KS96] stores each vertex's neighbors in contiguous memory to exploit the locality in the accesses, such that all accesses to the adjacency list can be accomplished with $O(\frac{E}{B} + V)$ block transfers, where the V part reflects the jumps that that occur whenever a new vertex is to be processed. These jumps directly reveal the degree of the vertices being processed.

To address this challenge, we draw inspiration from the recent work of Ostrovsky [Ost24], who proposed a secure computation protocol for Dijkstra's algorithm. Specifically, we essentially regularize the graph by expanding the vertex set to $O(V)$ vertices so that every vertex has exactly $D = \Theta(E/V)$ neighbors. In this new graph, every vertex's neighbors will be stored in a contiguous segment, and the set of segments is randomly permuted along with an appropriate number of filler segments. A vertex v in the original graph with k neighbors will be split into $\lceil k/D \rceil$ vertices $v'_1, \ldots, v'_{\lceil k/D \rceil}$ in the regularized graph; further, every vertex v'_i will remember where the next instance v'_{i+1} is in the permuted list, so that we can easily jump to the segment corresponding to v'_{i+1}. This way, as long as every vertex in the regularized graph is accessed at most once, the adversary observes only random accesses at the segment granularity. Further, all adjacency list accesses can be accomplished with $O(\frac{E}{B} + V)$ block transfers. In Section 5, we construct a new primitive called an oblivious adjacency list iterator, which formally captures the above intuition.

Challenge 3: Hide which data structure is accessed. Another challenge is that in Kumar and Schwabe [KS96]'s algorithm, there are several data structures, including two prority queues and an adjacency list. If we simply replace each primitive with an oblivious counterpart, the adversary can still observe which data structure is being accessed in each time step, and this leaks information about the graph structure.

To tackle this challenge, we devise a fixed schedule such that in each time step, which data structure is accessed is pre-determined and independent of the graph structure; moreover, we ensure that the fixed scheduling does not increase the asymptotical performance bounds. To make this idea work, we have to extend the underlying data structures to support phantom operations, which emulate the access pattern of a normal operation but without causing any side effects. Specifically, for the aforementioned oblivious adjacency list, we introduce an appropriate number of filler vertices (with corresponding filler segments) to support such phantom operations. In Section 6 we put all of these ideas together to obtain our oblivious SSSP algorithm (Algorithm 2).

1.5 Additional Related Work

Although our paper does not focus on the special case of *sparse* graphs, we review recent progress on improvements for sparse graphs in the (non-private) algorithms community. The results mentioned below adopt the word RAM model. Duan et al [DMSY23] improve the classic $O(E+V\log V)$ bound from Dijkstra's algorithm with Fibonacci heaps [Dij59,FT87] by giving a randomized algorithm with $O(E\sqrt{\log V \log\log V})$ work. Yan [Yan25] derandomizes their algorithm to obtain a deterministic SSSP algorithm with the same efficiency. Similar improvements have very recently been obtained for directed graphs as well [DMSY26]. We emphasize that these bounds only provide an improvement in the sparse case of $E = o\left(V\sqrt{\frac{\log V}{\log\log V}}\right)$, and hence we still use $O(E + V\log V)$ in Table 2 as a baseline. However, in light of the continued interest from the algorithms community, we pose it as an open question whether for these settings (e.g., directed or sparse graphs), one can design oblivious algorithms whose performance closely matches the non-oblivious baselines.

Furthermore, in the (non-private) external-memory setting, techniques for directed SSSP are meaningfully different from the undirected case. To the best of our knowledge, the state-of-the-art (in terms of I/O cost) non-private directed SSSP algorithm uses a *buffered repository tree* [ABD+02, CR04] (in place of the additional priority queue introduced by Kumar and Schwabe [KS96] for the undirected case). We focus only on undirected graphs in this paper; developing an oblivious algorithm with I/O cost nearly matching the non-private baseline for directed graphs remains an interesting open problem. In particular, efficient oblivious analogs to the buffered repository tree are not known.

Meanwhile, Ostrovsky [Ost24] propose a secure computation protocol for computing the Dijkstra's algorithm. His work focuses on designing an interactive protocol, and the result does not directly translate to any improvement for oblivious SSSP in either the word RAM or external-memory model relative to the prior state of the art.

1.6 Organization of the Paper

Section 4 The remainder of the paper is organized as follows. Section 2 formally introduces our computational model and the notion of obliviousness, and Section 3 describes the existing oblivious building blocks used in our constructions. Our new priority queue construction is then presented in Section 4. Section 6.1 formally introduces the SSSP problem and the (non-private) external-memory algorithm that we build on. In Section 5 we introduce the oblivious adjacency list iterator primitive, and we then present our new oblivious SSSP algorithm in Section 6. Finally, Section A gives additional explanation on external-memory oblivious simulation of an external-memory RAM. Deferred proofs are given in Section B through Section D.

2 Computational Model and Oblivious Simulation

Notation. We use $[m]$ to denote the set $\{1,\ldots,m\}$ and $\deg(u)$ to denote the number of edges incident to a vertex u. We often abuse notation and use $g(n) = O(f(n))\omega(1)$ to mean that for any $\alpha(n) = \omega(1)$, the function $g(n)$ satisfies $g(n) = O(\alpha(n)f(n))$. We use "$_$" to denote a term or variable that is not material in a given algorithm, e.g., for an operation $\mathsf{Op}()$ that returns two values $a, b \leftarrow \mathsf{Op}()$, we write $v_a, _ \leftarrow \mathsf{Op}()$ when we do not need to use the second component of the return value. When writing work and I/O bounds for a graph $G = (V, E)$, we will often abuse the notation V, E to mean the *sizes* $|V|, |E|$ of the vertex and edge sets as well. We use $\mathsf{negl}(\cdot)$ to denote a negligible function, i.e. for every constant $c > 0$, there exists an $n_0 \in \mathbb{N}$ such that for all $n > n_0$, $|\mathsf{negl}(n)| < n^{-c}$.

2.1 External-Memory Model

We assume a standard Random Access Machine (RAM) with an external memory. Like the standard literature, we assume that the word size $w = \Omega(\log n)$ where n denotes the RAM's space, i.e., every memory word is large enough to store a memory address. The CPU has constant number of registers, and in every time step, the CPU can perform some computation over its register contents, and read and write memory. Like in the standard algorithms literature, we assume that the CPU can perform an addition or a bit-wise boolean operation on $O(1)$ words in unit time. We assume that the CPU has M words of cache. Any time the CPU wants to read or write data that is not present in the cache, it must fetch the corresponding data from the external memory. The atomic unit for reading and writing external memory is called a *block*, whose size is B words. Even if the CPU only wants to read/write a single word, it must still fetch the entire block containing the word into its cache to perform the operation.

Metrics. In the external memory model, we care about two performance metrics. The number of CPU steps taken by an algorithm is called the algorithm's *work*, and the number of cache misses (i.e., the number of blocks transmitted between the cache and the external memory) is called the *I/O cost*.

Cache-agnostic model. Frigo et al. [FLPR99] introduced a stronger version of the external-memory model in which algorithms do not know the underlying parameters M and B of the memory hierarchy. In the (non-oblivious) algorithms literature this model is generally called the *cache-oblivious* model; we refer to it as the *cache-agnostic* model to avoid overloading the usage of the word "oblivious." The cache-agnostic model essentially asks for *universal* algorithms that achieve optimal performance on any concrete architecture. Moreover, on a multi-level storage hierarchy (e.g., CPU, cache, memory, disk), the model asks for algorithms that achieve optimal I/O cost at any storage layer.

Trusted hardware as an instantiation of the external-memory model. Several industry leaders such as Signal, Meta, and Ethereum have either deployed or announced their plans to deploy oblivious algorithms using hardware enclaves (e.g., Intel's SGX) [met, sig, But]. A hardware enclave can be viewed as a hardware-enforced sandbox. Typically, data is encrypted at rest and in transit, and decrypted only inside the hardware enclave, where computation on secret data takes place. Many real-world computations require accesses to a large data set that cannot fit within the enclave's private memory [sig, But]. Therefore, the enclave often needs to read and write (encrypted) data residing either in external memory or on disk. This is often accomplished using a page swap operation (implemented through a system call called `OCall`). Since an `OCall` typically involves context switching, encryption and decryption, and possible disk swaps, it is a relatively heavy-weight operation. For this reason, the granularity of data fetched during an `OCall` is typically at least the size of a memory page (4KB in most architectures today).

Therefore, the external-memory model perfectly captures the trusted hardware model of computation, where M corresponds to the size of the enclave's protected memory, B corresponds to the page size, and I/O costs correspond to the number of page swaps in and outside the enclave.

2.2 Oblivious Simulation

Observable access patterns. As in the standard literature on oblivious algorithms, we assume that the data contents themselves are protected via a cryptographic mechanism (e.g., encryption)

and are therefore unobservable to the adversary. However, as the program executes, the adversary may still observe its access patterns, which include the following information:

(1) *Block-level accesses*: which blocks in external memory are visited in each time step, and whether the operation is a read or write;

(2) *Word-level accesses*: which word within the CPU's cache is accessed in each time step, and whether the operation is a read or write;

(3) *Instruction type*: the *type* of instruction executed in each time step (though not its operands).

On a real-world hardware enclave such as Intel's SGX, block-level accesses are directly exposed to the operating system which acts as an intermediary for serving the page swaps. Word-level accesses can be observed through cache-timing attacks [RTSS09, DMWS12], and instruction types can leak through timing measurements. Like the standard literature, we assume that for the same CPU instruction (not including memory operations), its execution time is fixed and independent of its operands; however, different CPU instructions may exhibit different execution times (e.g., multiplication can take more cycles than addition).

Oblivious simulation. At a high level, oblivious simulation captures the security requirement that even if the adversary can observe the access patterns during a program's execution, it still cannot infer the secret inputs to the program. Formally, oblivious simulation can be defined using a simulation paradigm, requiring that the real-world algorithm's output and access pattern distribution should be indistinguishable from an ideal world in which the output is generated by a trusted ideal functionality, and the access patterns are generated by a simulator that is not aware of the inputs to the computation. Such a simulation-based definition is standard in this body of work, and is inspired by the multi-party computation literature [Can00]. Since both the output and the access patterns are captured by the joint distribution in the definition, the same definition simultaneously captures correctness and security.

Definition 2.1 (Oblivious simulation)**.** We say that a possibly randomized algorithm Alg *obliviously realizes* a (possibly randomized) functionality $\mathcal{F}$ with leakage function $\mathcal{L}(\cdot)$ if there exists a poly-time simulator Sim and a negligible function $\mathsf{negl}(\cdot)$ such that for any input x,

$$\mathbf{Exec}^{\mathsf{Alg}}(1^\lambda, x) \overset{\mathsf{negl}(\lambda)}{\approx} \Big(\mathcal{F}(x), \quad \mathsf{Sim}(1^\lambda, |x|, \mathcal{L}(x))\Big).$$

In the above, the notation $\mathbf{Exec}^{\mathsf{Alg}}(1^\lambda, x)$ means to execute $\mathsf{Alg}(1^\lambda, x)$ and return the program's output as well as access patterns; the notation $\mathcal{L}(x)$ represents any information assumed to be public (other than the size) about an input x; and the notation $\overset{\mathsf{negl}(\lambda)}{\approx}$ means that the left-hand- and right-hand-sides are computationally indistinguishable, i.e., the distinguishing advantage of any probabilistic polynomial-time (PPT) distinguisher is upper bounded by $\mathsf{negl}(\lambda)$. We will sometimes specify that the left-hand and right-hand-sides have negligible statistical distance, i.e. that they are statistically indistinguishable, in which case we obtain statistical security instead of computational security. Whenever the leakage function $\mathcal{L}(\cdot)$ is not specified, we assume that it is null, i.e., nothing other than the size $|x|$ of the input is leaked.

As mentioned, we consider a strong adversary that observes not only block-level accesses, but also word-level accesses, and instruction types. Obliviousness against such a strong adversary is also called *strongly oblivious* [LSX19, RS21] or *doubly oblivious* [MPC+18] in the literature.

Observe that Definition 2.1 captures both correctness and obliviousness by requiring that the *joint distribution* of the output and access patterns be indistinguishable in the ideal and real worlds.

2.3 Definitions: Oblivious Priority Queue

An oblivious priority queue, parametrized by the maximum capacity N, implements the following *reactive* functionality denoted $\mathcal{F}_{\text{PQ}}^{N}$, which maintains a data structure that stores up to N elements each carrying a *distinct key* along with a *priority*. The functionality supports the following operations:

- **ExtractMin**(): Remove and return the element with the smallest priority. We assume that ties on the priority are broken by key comparisons.
- **Delete**(k): Remove the element with key k, if one exists.
- **DecrKey**($k; p$): If an element with key k and priority p' exists, update its priority to $\min\{p, p'\}$. Else, insert an element with key k and priority p.
- **FindMin**(): Return (without removing) the element with the smallest priority.
- **IsEmpty**(): Return `True` if the data structure contains no elements, and `False` otherwise.

Note that insertion can be implemented with **DecrKey**, and so we do not write a separate **Insert** algorithm. We now formally define oblivious simulation of the reactive functionality $\mathcal{F}_{\text{PQ}}$.

Definition 2.2 (Oblivious Simulation of $\mathcal{F}_{\text{PQ}}$, adapted from [Shi20])**.** Let N be upper bounded by a fixed polynomial in λ. We say that a stateful algorithm $\mathsf{PQ}(1^\lambda, N)$ *obliviously realizes the priority queue functionality* $\mathcal{F}_{\text{PQ}}^{N}$ if there exists a polynomial-time stateful simulator $\mathsf{Sim}(1^\lambda, N, \cdot)$ such that for any $T = \mathsf{poly}(N)$ and for any *admissible* adversary $\mathcal{A}$, its views in the following experiments are computationally indistinguishable:

- $\mathsf{Ideal}(1^\lambda, N)$: For $j \in [T]$:
  - $\mathcal{A}(1^\lambda, N)$ adaptively issues the next query Q_j whose type is among {**ExtractMin**, **DecrKey**, **Delete**, **FindMin**, **IsEmpty**}.
  - $\mathcal{A}$ receives the correct output of the query (if any) as well as the simulated access patterns output by $\mathsf{Sim}(1^\lambda, N, j)$.
- $\mathsf{Real}(1^\lambda, N)$: Initialize an instance of $\mathsf{PQ}(1^\lambda, N)$. Now, for $j \in [T]$, do the following:
  - $\mathcal{A}(1^\lambda, N)$ adaptively issues the next query Q_j whose type is among {**ExtractMin**, **DecrKey**, **Delete**, **FindMin**, **IsEmpty**}.
  - Invoke the corresponding operation of PQ, and return its output (if any) as well as the access patterns to $\mathcal{A}$.

In the above experiments, the adversary $\mathcal{A}(1^\lambda, N)$ is said to be *admissible* if its T queries are such that the maximum number of elements stored in the (ideal-world) priority queue at any time is at most N.

Observe that in Definition 2.2 the simulator Sim knows only the maximum capacity N, and does not know the type of the operations or the operands. Therefore, the above security definition implies that the access patterns must also hide the type of the operation.

2.4 Definitions: Oblivious Single-Source Shortest Path

We next formally define single-source shortest path (SSSP) and introduce the relevant notations and input-output format. We use $G = (V, E)$ to denote an undirected, weighted graph where V denotes the vertex set and E denotes the edge set. We assume that each edge is of the form (u, v, w) where $u, v \in V$ and w is a non-negative weight. Given a source vertex $s \in V$, our goal is to compute the *shortest path distance* from s to every vertex $u \in V$. Besides outputting the distance, the algorithms described in our paper (Algorithm 1 and Algorithm 2) can also easily be augmented to additionally output each vertex's predecessor on the shortest path tree.

Input and output format. We assume that the input graph G is expressed as an adjacency list denoted $\mathsf{AdjArr} = \{(u, v, w), (v, u, w)\}_{(u,v)\in E}$, where each edge $(u, v, w) \in E$ appears twice, once as (u, v, w), and once as (v, u, w). Without loss of generality, we may assume that the AdjArr is sorted according to lexicographical ordering, i.e., all edges incident to the same vertex appear together — if the array is not sorted to start with, we can always obliviously sort it with $O(\frac{E}{B} \log_{\frac{M}{B}} \frac{E}{B})$ I/O cost and $O(E \log E)$ work [RS21], which are asymptotically absorbed by the rest of the algorithms. We further assume that AdjArr comes with a separate list $\{u\}_{u\in V}$, as this is necessary to express the vertex set V (since some $u \in V$ may not appear in the edges, if $\deg(u) = 0$). Besides the graph G and source vertex s, the algorithm may additionally take as input the security parameter 1^λ.

We assume that the output is an *unordered* array of the form $\{(v, d)\}_{v\in V}$ where each pair (v, d) gives the correct distance d corresponding to the shortest path from s to v.

We say that an SSSP algorithm on input x is oblivious if it obliviously simulates $\mathcal{F}_{\text{SSSP}}$ on x with leakage $\mathcal{L}(x) = (|V|, |E|)$, where $\mathcal{F}_{\text{SSSP}}$ is the ideal functionality that correctly outputs an unordered array $\mathsf{ans} = \{(v, p)\}_{v\in V}$ of the form described above.

Oblivious SSSP. Oblivious SSSP is simply an oblivious algorithm that realizes the an ideal functionality which solves the SSSP problem, where the allowed leakage includes the number of vertices $|V|$ and the number of edges $|E|$.

3 Existing Oblivious Building Blocks

3.1 Building Blocks Equivalent to Sorting

Prior work has shown how to obliviously implement various algorithmic building blocks in the external-memory model. Below, we review some primitives that we rely on and their asymptotical bounds. All oblivious building blocks below take the security parameter 1^λ as an additional input, but for simplicity we avoid writing 1^λ explicitly in the notation:

- **Oblivious sort**, denoted $\mathbf{Sort}(\mathsf{Arr})$, obliviously realizes the following ideal functionality. Sort the input array Arr and output the result. Without loss of generality, we shall assume that all array elements are distinct — if not, we can always tag all elements by their indices in the input array and break ties using the indices.
- **Oblivious bin placement**, denoted $\mathbf{BinPlace}^{\beta,Z}(\mathsf{Arr})$, obliviously realizes the following ideal functionality. Let β denote the number of bins and Z denote each bin's capacity. Let Arr be an input array containing real elements and filler elements, and suppose that every real element is tagged with a number from $[\beta]$ indicating the destination bin it wants to go to. It is promised that no more than Z real elements want to go to the same bin. The functionality outputs a list

of β bins where each bin contains the elements destined for it, padded with filler elements to a load of Z. Within the same bin, the elements are ordered based on their indices in the input.

- **Oblivious routing**, denoted **Route**($\mathsf{Src}, \mathsf{Dst}$), obliviously realizes the following functionality. Let Src be a source array where each element is tagged with some key (possibly $\perp$) and a value, and it is promised that all non-$\perp$ keys in Src are distinct. Let Dst be a destination array where each element is tagged with a key. The keys of elements in Dst are not necessarily distinct. The functionality sends each element in Dst the value of the element in Src with the same key, or $\perp$ if the key was $\perp$ or not found in Src.
- **Oblivious random permutation**, denoted **RandPerm**(Arr), obliviously realizes the following randomized functionality. Permute the elements of Arr accordingly to a uniformly random permutation and output the results.
- **Oblivious deduplication**, denoted **Dedup**($\mathsf{Arr}_0, \mathsf{Arr}_1, n'$), obliviously realizes the following functionality. Let Arr_0 and Arr_1 be two input arrays of items, where each item is either a filler $\perp$ or a real item in the form of a key-value pair (k, v). It is promised that the total number of distinct keys in the union of the two input arrays is at most n', and we assume there is some comparison order on both keys and values. The functionality outputs an array of length n' containing only the smallest-value item of each key that appears in the union of the input arrays, padded with $\perp$ up to the given length.

Prior works [BCP15,CCS17,ACN+20,RS21,GWT+23] have shown how to realize all these primitives through (possibly randomized) algorithms that incur $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ I/O cost and $O(n \log n)$ work, where n is an upper bound on the the number of elements in the input/output array(s), and assuming that each element can be stored in $O(1)$ words. We note that the realizations of these primitives are statistically secure.

3.2 Building Blocks Equivalent to Compaction

The following oblivious primitives can be accomplished in linear work. We will not care about their I/O efficiency because they will only be applied to small sub-problems (that fit in cache).

Oblivious compaction. Given an input array containing items where each item is marked with a boolean label that is either 0 or 1, a compaction algorithm (denoted **Compact**) moves all the 0-items to the front of the array, and all the 1-items to the end of the array. In the output array, the relative ordering among items with the same key may be arbitrary.

Asharov et al. [AKL+20] constructed a *deterministic* algorithm that compacts an array of length n in $O(n)$ work. Moreover, the algorithm's access patterns are also deterministic and depend only on the input length but not the contents of the input array.

Oblivious intersperse. We will rely on two variants of an intersperse procedure:

- **Intersperse**($\mathsf{Arr}_0, \mathsf{Arr}_1$): let Arr_0 and Arr_1 be two randomly shuffled input arrays of arbitrary lengths, the algorithm outputs the concatenation of the two arrays shuffled with a fresh permutation.
- **IntersperseRF**(Arr): let Arr be an input array that contains real and filler items. Further, the relative ordering among the real items is random. The algorithm outputs a random permutation of Arr shuffled with fresh random coins.

We say that **IntersperseRF** is perfectly oblivious iff there exists a simulator Sim such that the following two experiments are identically distributed for any array Arr:

- Real: randomly shuffle the real items in Arr using freshly sampled coins (without moving any filler item). Run the **IntersperseRF** algorithm on the resulting array, output the algorithm's outcome as well as the access patterns.
- Ideal: output $\mathcal{F}_{\mathsf{shuffle}}(\mathsf{Arr})$ and simulated access patterns $\mathsf{Sim}(|\mathsf{Arr}|)$.

The (perfect) obliviousness of **Intersperse** variant can also be similarly defined except that in this case, the simulator knows the two input lengths Arr_0 and Arr_1.

Asharov et al. [AKL+20] showed how to construct perfectly oblivious **Intersperse** and **IntersperseRF** achieving $O(n)$ work where n is the total input length. The algorithms are randomized, but their access patterns are deterministic.

Oblivious selection. The input includes a rank k and an Arr containing real and filler items. A selection algorithm denoted **Select** outputs the k-th smallest real item in Arr, and if there are fewer than k real items in Arr, simply output the maximum item.

Oblivious selection can be accomplished using oblivious compaction [AKL+20] and the classical selection algorithm by Blum et al. [BFP+73]. The resulting algorithm is deterministic, and its access patterns depend only on the input length but not the contents of the input array. The algorithm achieves $O(n)$ work on an input array of length n.

3.3 Oblivious Hash Table for Non-Recurrent Lookups and Shuffled Inputs

A hash table HT supports the following operations:

- $\mathsf{Stash} \leftarrow \textbf{Build}(1^\lambda, \mathsf{Arr})$: The input array Arr contains n items where each item is either a filler or a real key-value pair of the form (k, v). It is promised that all real items in Arr have distinct keys. Let Stash be an array of length $\log \lambda$ containing a randomly sampled subset of Arr, along with an appropriate number of fillers. All real items in Arr that are not in Stash are then stored in the main data structure. Output Stash in a randomly shuffled order.
- $v \leftarrow \textbf{Lookup}(k)$: upon receiving a key k, if some (k, v) exists in the main data structure with a matching key k, then return v; else, return $\perp$.
- $\mathsf{Unvisited} \leftarrow \textbf{Extr}()$: output an array $\mathsf{Unvisited}$ which contains all remaining items in the main data structure that have not been looked up, possibly padded with fillers to a fixed length of n. All real and filler items in $\mathsf{Unvisited}$ are randomly shuffled before the array is output.

We say that some implementation HT obliviously realizes an ideal hash table for non-recurrent lookups and shuffled inputs, iff there exists some PPT simulator Sim, such that for any *admissible* non-uniform PPT adversary $\mathcal{A}$, its views in the following two experiments are computationally indistinguishable, where $\mathcal{A}$ is said to be admissible iff with probability 1, the same real k is looked up at most once:

Experiment $\mathsf{HTReal}^{\mathcal{A}}(1^\lambda)$**.** Consider the following experiment where each call to $\mathcal{F}_{\mathsf{shuffle}}$ permutes the input array using fresh random coins:

- *Initialization.* The adversary $\mathcal{A}(1^\lambda)$ submits an array Arr of length $n = \mathsf{poly}(\lambda)$ in which real items have distinct keys. The challenger calls $\mathsf{Stash} \leftarrow \mathsf{HT}.\textbf{Build}(\mathcal{F}_{\mathsf{shuffle}}(\mathsf{Arr}))$, and returns to $\mathcal{A}$ the Stash, and the access patterns of the **Build** algorithm followed by some canonical delimiter (e.g., ";").

- *Queries.* In each iteration $t = 1, 2, \ldots$: $\mathcal{A}$ submits a **Lookup** query for some specified key k; the challenger calls $v \leftarrow \mathsf{HT}.\mathbf{Lookup}(k)$, and returns v to $\mathcal{A}$, as well as the access patterns of the algorithm along with some canonical delimiter.
- *Extract.* Finally, $\mathcal{A}$ submits "`Extr`", and the challenger calls $\mathsf{Unvisited} \leftarrow \mathsf{HT}.\mathbf{Extr}()$. Return to $\mathcal{A}$ the output $\mathsf{Unvisited}$, as well as the access patterns of the algorithm.

Experiment $\mathsf{HTIdeal}^{\mathcal{A},\mathsf{Sim}}(1^\lambda)$**.**

- *Initialization.* The adversary $\mathcal{A}(1^\lambda)$ submits an array Arr of length $n = \mathsf{poly}(\lambda)$ in which real items have distinct keys. Let Stash be an array of fixed length $\log \lambda$, containing a subset sampled from Arr according to some polynomially samplable distribution $\mathcal{D}$ (which depends on the scheme), as well as an appropriate number of fillers. Return to $\mathcal{A}$ the randomly shuffled stash $\mathcal{F}_{\text{shuffle}}(\mathsf{Stash})$ as well as the simulated access patterns $\mathsf{Sim}(1^\lambda, n, 0)$.
- *Queries.* In each iteration $t = 1, 2, \ldots$: $\mathcal{A}$ submits a **Lookup** query for some specified key k; return to $\mathcal{A}$ the correct answer of the query (by looking up the main data structure), as well as the simulated access patterns $\mathsf{Sim}(1^\lambda, n, t)$.
- *Extract.* Finally, $\mathcal{A}$ submits "`Extr`". Let $\mathsf{Unvisited}$ be an array of length n containing the items in the main data structure that have not been looked up, and an appropriate number of fillers. Return to $\mathcal{A}$ the randomly shuffled array $\mathcal{F}_{\text{shuffle}}(\mathsf{Unvisited})$, as well as the simulated access patterns $\mathsf{Sim}(1^\lambda, n, \text{“}\infty\text{”})$.

Theorem 3.1 (Oblivious hash table [AKL+20])**.** *Assume the existence of a pseudorandom function, and suppose that $n \geq \log^9 \lambda$. Then, there exists an oblivious hash table (for non-recurrent lookups and shuffled inputs), where*

- *the* **Build** *and* **Extr** *procedures incur $O(n)$ work; and*
- *each* **Lookup** *operation incurs $O(1)$ work.*

4 Optimal Strongly Oblivious Priority Queue

In this section we present our optimal strongly oblivious priority queue construction. The main result of this section is stated in Corollary 4.5, which we will build up using the components outlined in the following subsections.

4.1 Oblivious Deduplication for Randomly Shuffled Inputs

We start by introducing a modified deduplication building block that can be realized in linear work.

Syntax. The syntax is the same as the one defined in Section 3.1 except that we now additionally require the input arrays to be randomly shuffled. A deduplication algorithm, denoted $\mathsf{Dedup}(1^\lambda, \mathsf{Inp}_0, \mathsf{Inp}_1, n')$, has the following syntax:

- **Input.** The input contains randomly shuffled input arrays Inp_0 and Inp_1 of lengths n_0 and n_1, respectively, and the output length n'. Each item in the input arrays is either a filler or a real item in the form of a key-value pair denoted (k, v).

  It is promised that within each individual input array, all real items' keys are distinct; however, the same key can appear in both input arrays. Further, it is promised that the total number of distinct keys in the union of the two input arrays is at most n'.

- **Output.** Suppose there is a publicly known tie-breaking function that outputs a preferred item when given two items of the same key. We want to output a combined array such that if the same key appears in both input arrays, preserve the one that wins the tie-breaking. Moreover, the output array is possibly padded with fillers to a fixed length of n', and randomly shuffled.

Obliviousness. We say that some $\mathbf{Dedup}(1^\lambda, \cdot, \cdot)$ algorithm obliviously realizes ideal deduplication under randomly shuffled inputs, iff there exists a PPT simulator Sim such that for any n_0, n_1 that are polynomially bounded in the security parameter λ, for any input arrays Inp_0 and Inp_1 of lengths n_0 and n_1 respectively satisfying distinctness of the real keys within each individual array, the outputs of the following two experiments are computationally indistinguishable:

- $\mathsf{DedupReal}(1^\lambda, \mathsf{Inp}_0, \mathsf{Inp}_1)$: call $\mathsf{Outp} \leftarrow \mathbf{Dedup}(1^\lambda, \mathcal{F}_{\text{shuffle}}(\mathsf{Inp}_0), \mathcal{F}_{\text{shuffle}}(\mathsf{Inp}_1))$, output Outp as well as the **Dedup** algorithm's access patterns.
- $\mathsf{DedupIdeal}(1^\lambda, \mathsf{Inp}_0, \mathsf{Inp}_1)$: let Comb be an array of length $n_0 + n_1$ that correctly combines and deduplicates the elements from Inp_0, Inp_1. Output $\mathcal{F}_{\text{shuffle}}(\mathsf{Comb})$ along with the simulated access patterns $\mathsf{Sim}(1^\lambda, n_0, n_1, n')$.

Construction. We now describe our oblivious Dedup algorithm.

1. Create a hash table by calling $\mathsf{HT.Stash} \leftarrow \mathsf{HT}.\mathbf{Build}(1^\lambda, \mathsf{Inp}_0)$.
2. Let $\mathsf{New} = \emptyset$. For each real or filler item $(k, v_1) \in \mathsf{Inp}_1$, call $v_0 \leftarrow \mathsf{HT}.\mathbf{Lookup}(k)$. If $v_0 = \perp$, append (k, v_1) to New; else append the preferred item among (k, v_0) and (k, v_1) to New.
3. Let $\mathsf{Old} \leftarrow \mathsf{HT}.\mathbf{Extr}()$, and let $\mathsf{Comb} \leftarrow \mathbf{Intersperse}(\mathsf{Old}, \mathsf{New})$ be a combined array of size $n_0 + n_1$. Rebuild a hash table of size $n_0 + n_1$ by calling $\mathsf{HT'.Stash} \leftarrow \mathsf{HT'}.\mathbf{Build}(1^\lambda, \mathsf{Comb})$.
4. Use oblivious sort to perform deduplication on the combined stash $\mathsf{HT.Stash}||\mathsf{HT'.Stash}$, and use oblivious random permutation to shuffle the deduplicatd array — let $\mathsf{CombStash}$ be the outcome which has length $2\log\lambda$.
5. Let $\mathsf{New} = \emptyset$. For each real or filler item $(k, v) \in \mathsf{CombStash}$, call $v' \leftarrow \mathsf{HT'}.\mathbf{Lookup}(k)$. If $v' = \perp$, append (k, v) to New; else append the preferred item among (k, v) and (k, v') to New.
6. Let $\mathsf{Old} \leftarrow \mathsf{HT'}.\mathbf{Extr}()$. Let $\mathsf{Comb'} \leftarrow \mathbf{Intersperse}(\mathsf{Old}, \mathsf{New})$, and then call **Compact** to shrink $\mathsf{Comb'}$ to length n' without losing any real items. In the resulting array $\mathsf{Comb'}$, all filler items appear at the end.
7. Output $\mathbf{IntersperseRF}(\mathsf{Comb'})$.

Theorem 4.1 (Oblivious deduplication for randomly shuffled inputs)**.** *The above algorithm obliviously realizes ideal deduplication under randomly shuffled inputs. Moreover, if* $n_0+n_1 \geq \log\lambda\log\log\lambda$, *the algorithm incurs* $O(n_0 + n_1)$ *work.*

Proof. It is not hard to see that the above algorithm incurs $O(n_0 + n_1 + \log\lambda\log\log\lambda)$ work. The $\log\lambda\log\log\lambda$ part is asymptotically dominated by $O(n_0 + n_1)$ assuming $n_0 + n_1 \geq \log\lambda\log\log\lambda$. The obliviousness proof is deferred to Section D. □

4.2 Overview of External-Memory Oblivious Priority Queue

We now describe a cache-agnostic variant of the oblivious priority queue of Jafargholi et al. [JLS21].

Data structure. Let $Z = O(\log^c \lambda)$ be the bucket size,[4] set such that Z is a multiple of 8 and where $c > 1$ is an appropriate constant. The oblivious priority queue's data structure is a binary tree of buckets each of size Z. The tree has N/Z leaves where N is the (logical) capacity of the priority queue. Each item is of the form (p, k, v) where p is the priority, k is the key, and v is the value.[5] An item with the key k must reside on some randomly chosen tree path computed by applying a pseudorandom function $\mathsf{PRF}(\mathsf{sk}, k)$ to its key k, where the PRF's secret key sk is unknown to the adversary. Although Jafargholi et al. [JLS21] showed that this PRF can actually be instantiated with a k-wise independent hash, our description simply assumes the existence of a PRF since we will need it in our new, strongly oblivious instantiation.

We number the tree's levels as $0, 1, \ldots, L = O(\log(N/Z))$, where the root is at level 0, and level i of the tree has 2^i buckets.

Operations. We now explain how to support the queries. Our overview below omits the details of each bucket's internal data structure, since Jafargholi et al. [JLS21]'s realization achieves only weak obliviousness. Later in Section 4.3, we present a new approach for realizing the bucket's internal data structure that yields a strongly oblivious algorithm.

Every time there is a **FindMin**, **ExtractMin**, **Delete**(k) or **DecrKey**(p, k, v) operation, do the following:

1. *Add to root bucket.* If the operation is **FindMin** or **ExtractMin**, add a filler item $\perp$ to the root bucket; else, add either $(\text{“}\texttt{delete}\text{”}, k)$ or $(\text{“}\texttt{deckey}\text{”}, (p, k, v))$ to the root bucket depending on the type of the operation.
2. *Periodic maintainence.* For $i \in \{0, 1, \ldots, L-1\}$, every $2^i \cdot (Z/8)$ operations, we rebuild the (non-leaf) levels $0, 1, \ldots, i$. The rebuild works as follow:
   (a) First, for levels $j = 0$ to i, each bucket at level j performs a **PushDown** operation which distributes its contents to its two children. The bucket is emptied after the **PushDown** operation.
   (b) Next, for levels $j = i$ down to 0, each bucket at level j performs a **PullUp** to obtain the $Z/2$ smallest items (ranked by priority) of the form $(\text{“}\texttt{deckey}\text{”}, (_, _, _))$ from the union of the two children — if there are not enough such items, simply pad with fillers. These items are then removed from the corresponding child bucket.
3. *Output.* If the operation is **ExtractMin** or **FindMin**, then discover the answer using the root bucket, and remove the smallest item from the root bucket in the case of **ExtractMin**. Else, perform a fake operation on the root bucket to emulate access patterns without causing any side effect.

Details of PushDown. We now elaborate on the **PushDown** operation. Recall that each real item belongs to some tree path. This tree path determines whether each item in the parent bucket should go left or right. When the left (or right, respectively) child receives the portion that should go left (or right, respectively), it must perform the following deduplication procedure to ensure the distinctness of each key k in this bucket:

- $(\text{“}\texttt{delete}\text{”}, k) + (\text{“}\texttt{delete}\text{”}, k)$: preserve one of them;

[4] Jafargholi et al. [JLS21] sets the bucket size to be $\Theta(M)$, we change the parameter to make it cache-agnostic.

[5] The value component is an additional payload added for generality, and can be considered to always be $\perp$ in our earlier priority queue syntax (Section 2.3).

- ("`delete`", k) + ("`deckey`", (p, k, v)): preserve the one that is fresher — we may assume that every item carries a timestamp which records the time it is added to the root bucket;
- ("`deckey`", (p, k, v)) + ("`deckey`", (p', k, v')): preserve the one with the smaller priority.

Theorem 4.2 (External-memory oblivious priority queue [JLS21])**.** *Suppose that we store the buckets level by level contiguously in memory. Then, the above priority queue is weakly oblivious, and achieves amortized $O(\frac{1}{B} \cdot \log(N/M))$ I/O cost per operation. Further, assuming we use the appropriate data structure to implement each bucket, then each operation can be supported using amortized $O(\log N)$ work.*

One crucial insight in the proof of the above theorem [JLS21] is to argue the following:

Claim 4.3 (Bucket size bound [JLS21])**.** *Suppose that at most $\mathsf{poly}(\lambda)$ queries are made. Then, except with negligible probability, no bucket ever contains more than Z items.*

Later in Section 4.3, our oblivious implementation of the bucket's internal data structure implicitly relies Claim 4.3 since we always truncate the bucket size to Z at the end of each bucket-level operation.

4.3 Oblivious Implementation Bucket-Level Operations

Our result assumes a lower bound on cache size of $M = \Omega(\log^c \lambda)$, similar to [JLS21], such that bucket-level operations can be done entirely from cache and thus incur only additional work, not I/O cost. All buckets need to support the **PushDown** and **PullUp** operations. Moreover, the root bucket alone must additionally support an oblivious priority queue for polylogarithmically-sized instances. For the root's priority queue, we can simply rely on a perfectly secure ORAM [DMN11, CNS18] to compile the ordinary binary heap, incurring $\mathsf{poly} \log \log \lambda$ work per query. This overhead is asymptotically absorbed in our final priority queue construction.

Below, we mainly focus on how to implement the **PushDown** and **PullUp** operations.

Main invariant: randomly shuffled buckets. We describe how to obliviously implement the bucket-level operations **PushDown** and **PullUp**. The main invariant we maintain is that each bucket is always randomly shuffled (with coins unknown to the adversary) before and after each **PushDown** or **PullUp** operation.

Implementing the PushDown operation. In **PushDown**, a parent bucket divides its items into two (approximate) halves, sent to its left and right children, respectively. Whether an item belongs to the left or right half depends on the hash of its key $\mathsf{PRF}(\mathsf{sk}, k)$. The **PushDown** operation can be implemented as follows where we use $\mathsf{Parent}, \mathsf{Left}, \mathsf{Right}$ to denote the buckets of the parent, and the left and right children, respectively.

1. Call **Compact** to divide the parent bucket Parent into two arrays denoted Left' and Right', respectively, each padded with fillers to a fixed length of Z. Call $\mathsf{Left}' \leftarrow \textbf{IntersperseRF}(\mathsf{Left}')$ and $\mathsf{Right}' \leftarrow \textbf{IntersperseRF}(\mathsf{Right}')$.
2. Merge Left' and Right' into the left and right children, respectively. Specifically, let $\mathsf{Left} \leftarrow \textbf{Dedup}(\mathsf{Left}, \mathsf{Left}', Z)$, and $\mathsf{Right} \leftarrow \textbf{Dedup}(\mathsf{Right}, \mathsf{Right}', Z)$.

Implementing the PullUp operation. In a **PullUp**, a parent (currently empty) receives $Z/2$ items with the smallest priorities from the union of the two children, and these items are removed from the children. **PullUp** can be implemented as follows:

1. Let $p \leftarrow$ **Select**(Left||Right, $Z/2$) be the $(Z/2)$-th priority (or the maximum priority if there are fewer than $Z/2$ items). Here the **Select** algorithm ignores all "`delete`" items and looks at only "`deckey`" items.
2. Call **Compact** to construct a new parent bucket Parent which contains all "`deckey`" items whose priority is p or less from Left||Right, padded with fillers to a fixed length of Z. Now, output **IntersperseRF**(Parent) as the new parent bucket.
3. Similarly, call **Compact** to remove the "`deckey`" items whose priority is p or less from Left, and Right, respectively, and call **IntersperseRF** to ensure that the resulting arrays are randomly shuffled. Output the resulting buckets at the left and right children, both of length n.

Theorem 4.4 (Oblivious implementation of buckets)**.** *For $Z \geq \log^c \lambda$ where $c > 1$ is an appropriate constant, the above* **PushDown** *and* **PullUp** *operations incur $O(Z)$ work where Z is the bucket's capacity. Further, the above implementations of* **PushDown** *and* **PullUp** *obliviously realize $\mathcal{F}_{\text{pushdown}}$ and $\mathcal{F}_{\text{pullup}}$, respectively, for randomly shuffled inputs, where $\mathcal{F}_{\text{pushdown}}$ and $\mathcal{F}_{\text{pullup}}$ are the natural ideal functionalities that correctly realize the syntax of* **PushDown** *and* **PullUp***, and moreover shuffle the output buckets using fresh random coins.*

Proof. The performance bounds are easy to see based on the performance bounds of the building blocks used. The obliviousness proof is somewhat mechanical and similar in spirit to the obliviousness proof of the deduplication algorithm described in Section D. Essentially, the proof can be shown through a sequence of hybrids based on the chronological order in which the primitives are invoked: if some **IntersperseRF** or **Dedup** instance's input satisfies the corresponding random shuffled requirement (either on the entire input or the real items only), we can replace the primitive with an ideal counterpart, which then guarantees that its output (i.e., input to the next primitive) satisfies the shuffled requirement. Like in the proof in Section D, the proof additionally relies on the following observation: since **Compact** is a deterministic algorithm with deterministic access patterns, running **Compact** on a randomly shuffled array will produce an output whose real items are randomly shuffled. □

Theorem 4.4, combined with Jafargholi et al. [JLS21], gives rise to the following corollary.

Corollary 4.5 (Strongly oblivious priority queue)**.** *Assume the existence of a pseudorandom function, and suppose that $n = \Omega(\log^c \lambda)$ and $M = \Omega(\log^c \lambda)$ for a suitably large constant $c > 1$. Then, there exists a cache-agnostic, strongly oblivious priority queue with capacity n that supports each* **FindMin**, **ExtractMin**, **Insert**, **DecrKey**, *and* **Delete** *operation with an amortized I/O cost of $O(\frac{1}{B} \log \frac{n}{M})$, and amortized $O(\log n)$ work.*

5 Oblivious Adjacency List Iterator

We now turn toward our goal of obliviously emulating the (non-oblivious) meta-algorithm for SSSP. In the meta-algorithm, whenever we expand a vertex, we need to visit all its neighbors. During this process, we need to hide which vertex is being expanded and how many neighbors it has. To accomplish this goal, we define and construct an oblivious *adjacency list iterator*. At a high level, our oblivious adjacency list is inspired by the work of Ostrovsky [Ost24], who proposed a secure

computation protocol for Dijkstra's algorithm. Specifically, we will have a table where each row stores up to D neighbors of some vertex, possibly padded with fillers to a fixed (public) length of D. If a vertex has more than D neighbors, it will occupy multiple rows in the table. Our oblivious SSSP algorithm will load one row at a time, and then scan all of the neighbors contained in the row. The set of rows is randomly shuffled such that the adversary cannot infer when we switch to expanding a new vertex. A pointer to a row in the table is henceforth called a *reference*. Looking ahead, in our oblivious SSSP algorithm, the oblivious priority queue will also store the the first row reference for each vertex it stores, such that when the vertex is popped out of the priority queue, we immediately learn which row in the table to access. Moreover, if a vertex's neighbors is split into multiple rows, then each row will store a reference to the next row of that vertex.

Formalizing the above intuition requires care. Below, we formally define the syntax and present the construction of our oblivious adjacency list iterator.

5.1 Syntax

We introduce a building block called an oblivious adjacency list iterator, denoted $\mathsf{Adj}^D = (\textbf{Init}, \textbf{Load}, \textbf{NextNb})$, parametrized by a positive integer D whose meaning will become clear as we describe the construction. Specifically, in a graph algorithm like Dijkstra's algorithm, we often need to iterate through all neighbors of some vertex that is currently being processed. Adj^D provides such an iterator object through three methods:

- $\mathsf{ref}_s, \mathsf{refs} \leftarrow \textbf{Init}(1^\lambda, \mathsf{AdjArr}, s)$: Upon receiving adjacency list AdjArr of a graph $G = (V, E)$, initialize the iterator data structure and output a list $\mathsf{refs} = \{(u, \mathsf{ref}_u)\}_{u \in V}$ containing a *reference* for each vertex; moreover, explicitly return the source vertex reference ref_s (so that the caller need not make a memory access to get it).
- $\textbf{Load}(\mathsf{ref})$:
  - If $\mathsf{ref} = \mathsf{ref}_u$ for some $u \in V$, then set the iterator's current context to u.
  - Else if $\mathsf{ref} = \perp$, set the current context to $\perp$.
- $(v, w, \mathsf{ref}_v, b) \leftarrow \textbf{NextNb}()$:
  - If the iterator's current context is some $u \in V$: if u has run out of neighbors, simply return $(\perp, \perp, \perp, 1)$; else, for the next neighbor v of u, return $(v, w, \mathsf{ref}_v, b)$ where w is the weight of the edge (u, v), ref_v is the reference for v, and $b \in \{0, 1\}$ indicates whether u has more remaining neighbors.
  - If the current context is $\perp$: return $(\perp, \perp, \perp, 1)$.

Intuitively, the caller (i.e., our oblivious SSSP algorithm) will call $\mathsf{Adj}^D.\textbf{Init}$ to initialize a graph $G = (V, E)$ once upfront. Afterwards, whenever the caller wants to iterate through the neighbors of some vertex u, it needs to first call $\mathsf{Adj}^D.\textbf{Load}(\mathsf{ref}_u)$, feeding it the correct reference of u. At this moment, subsequent invocations of $\mathsf{Adj}^D.\textbf{NextNb}$ will return the neighbors of u in the order that they appear in the original adjacency list AdjArr. As we will see in the Adj^D construction, this functionality is realized by essentially creating a regularized version of the graph (with a larger vertex set) where each vertex has exactly D neighbors, which will enable oblivious iteration through the adjacency list.[6]

[6] Assuming D is set to a fixed function of the public sizes $|V|$ and $|E|$, as will be the case in our caller (oblivious SSSP).

5.2 Security Definition

We want to make sure that the access patterns of Adj^D leak no information about the underlying graph structure besides the sizes $|V|$ and $|E|$. To achieve this, we expect that the caller (i.e., our oblivious SSSP algorithm) will respect a couple constraints which will be expressed as admissibility rules on the environment $\mathcal{Z}$ that interacts with Adj^D, described later in the definition. We also reserve a special input `nil` to **Load**, such that Adj^D.**Load**(`nil`) performs the access pattern of a normal **Load** call without changing any logical state. As we will see, this allows $\mathcal{Z}$ to satisfy admissibility without necessarily setting a new context every D **NextNb** calls.

Formally, to define security, consider the following experiment where an environment $\mathcal{Z}$ interacts with a challenger:

Experiment $\mathsf{Expt}^{\mathcal{Z}}(1^\lambda)$**:**

- **Initialization.** $\mathcal{Z}$ submits AdjArr, s, D to the challenger, where AdjArr is the adjacency list of some graph $G = (V, E)$ and $s \in V$. The challenger calls $\mathsf{ref}_s, \mathsf{refs} \leftarrow \mathsf{Adj}^D$.**Init**$(1^\lambda, \mathsf{AdjArr}, s)$.
- **Queries.** $\mathcal{Z}$ can adaptively make the following queries, where its first query must be "`load`" with $u \in V \cup \{\perp\}$:
  - **Load**: $\mathcal{Z}$ submits "`load`" along with $u \in V \cup \{\perp, \texttt{nil}\}$.
    If $u \in V$ and there exists a unique entry (u, ref_u) of the form $(u, \cdot)$ in refs,[7] the challenger calls Adj^D.**Load**(ref_u). Else if $u = \perp$, it calls Adj^D.**Load**$(\perp)$; else, it calls Adj^D.**Load**(`nil`).
  - **NextNb**: $\mathcal{Z}$ submits "`nextnb`."
    The challenger calls $v, w, \mathsf{ref}_v, b \leftarrow \mathsf{Adj}^D$.**NextNb**() and returns (v, w, b) to $\mathcal{Z}$.
- **Output access patterns.** Output the access patterns of the execution, defined as the access patterns[8] emitted by all calls to Adj^D.**Init**, Adj^D.**Load**, and Adj^D.**NextNb**, including a canonical delimiter that marks the end of every Adj^D operation.

Admissibility. Formally, we say that $\mathcal{Z}$ is admissible iff the following holds.

(A1) $\mathcal{Z}$ must call **NextNb** *exactly* D times in between any two (possibly `nil`) **Load** queries.

(A2) After a **Load** query with $u \in V$, $\mathcal{Z}$ must call **NextNb** *at most* $D \cdot \max\{1, \lceil \deg(u)/D \rceil\}$ before the next *non-`nil`* **Load** query. After a load query with $u = \perp$, $\mathcal{Z}$ must call **NextNb** *exactly* D times before the next non-`nil` **Load** query.

(A3) Over the entire experiment, $\mathcal{Z}$ must call **NextNb** exactly $T \cdot D$ times, where $T := 2|V| + 2\left\lceil \frac{|E|}{D} \right\rceil$.

(A4) The vertices in V submitted in **Load** queries over the entire experiment must be distinct.

We can now define the correctness and security guarantees desired from this iterator specification.[9]

[7] Of course, the intended functionality of **Init** is that refs should satisfy this requirement. The clause added here ensures the definition of the experiment is robust to the event that the **Init** algorithm fails and the output is incorrect, which we will show happens with negligible probability.

[8] Including block-level accesses, word-level accesses, and instruction type, to achieve strong obliviousness.

[9] Note that, relative to the definition of oblivious simulation in Section 2.2, we can split up the correctness and security notions here because the desired output is deterministic.

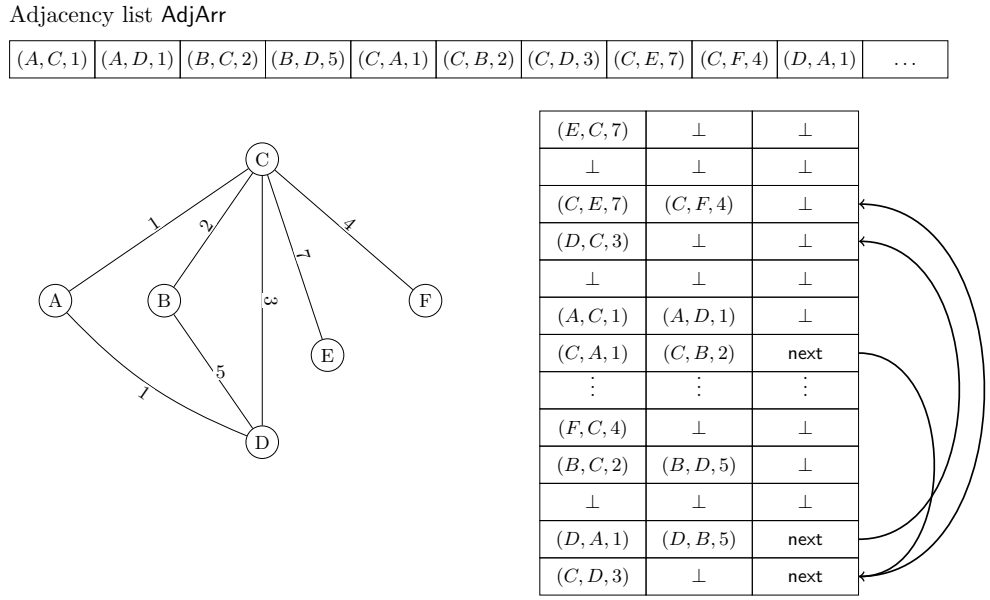


Figure 2: Storing the graph's adjacency list using an adjacency list iterator. The rows are randomly permuted. The (non-nil) pointers point to the continuation of the vertex's neighbor list.

Definition 5.1 (Oblivious adjacency list iterator)**.** We say that Adj^D is an oblivious realization of an adjacency list iterator for a graph $G = (V, E)$, iff there exists some negligible function $\mathsf{negl}(\cdot)$ such that the following holds for the experiment $\mathsf{Expt}^{\mathcal{Z}}(1^\lambda)$:

1. For all (even non-admissible) $\mathcal{Z}$, all λ, with probability at least $1 - \mathsf{negl}(\lambda)$, the outputs of Adj^D calls are correct in the following sense:

   (a) **Init** *validity.* The output $(\mathsf{ref}_s, \mathsf{refs})$ of **Init** is *valid*: (i) for every $u \in V$, there exists a unique entry (u, ref_u) of the form $(u, \cdot)$ in refs—we then call ref_u the *canonical* reference for u; (ii) the separately returned reference ref_s is equal to the canonical reference for s.

   (b) **NextNb** *correctness*: The output of each **NextNb** call is *correct*, as defined in the most natural manner: return the next neighbor v of the most recently loaded vertex, along with the correct edge weight w, canonical reference ref_v, and done-bit b; or return $(\perp, \perp, \perp, 1)$ if either all neighbors have been exhausted or the most recently loaded vertex is $\perp$.[10]

2. There exists a PPT simulator Sim^D such that for all *admissible* $\mathcal{Z}$, we have

$$\textsc{Acc}(\mathsf{Expt}^{\mathcal{Z}}(1^\lambda)) \overset{\mathsf{negl}(\lambda)}{\approx} \mathsf{Sim}^D(1^\lambda, |V|, |E|)$$

   where $\textsc{Acc}(\mathsf{Expt}^{\mathcal{Z}}(1^\lambda))$ denotes the *access pattern* of $\mathsf{Expt}^{\mathcal{Z}}(1^\lambda)$ as defined previously.

We will show in Section C.1 that our Adj^D construction, presented next, satisfies Definition 5.1.

[10] All **Load** calls with `nil` input are ignored when considering the most recently loaded vertex.

5.3 Construction

5.3.1 Internal Data Structure

Our construction is inspired by the work of Ostrovsky [Ost24] who constructed a secure computation protocol for Dijkstra's algorithm. We tailor his idea to best fit an external-memory model.

Table representation of adjacency list. During initialization, we build a table with D columns and $R := 3|V| + 3\left\lceil \frac{|E|}{D} \right\rceil$ rows to store the given adjacency list AdjArr. Every row is either a *real* row or a *filler* row, and there are at least $T := 2|V| + 2\left\lceil \frac{|E|}{D} \right\rceil$ filler rows in total (which are needed for obliviousness).

- *Real rows.* Each real row stores neighbors of the same vertex $u \in V$. In a row responsible for $u \in V$, each entry is of the form $(u, v, w, \mathsf{ref}_v)$. The adjacency list of a vertex $u \in V$ with $\deg(u)$ neighbors is split into $\max\{1, \lceil \deg(u)/D \rceil\}$ rows, with neighbors stored in the same order as in AdjArr. Further, if $\deg(u)$ is not a positive multiple of D, the last row responsible for u may not be fully utilized — in this case, the row is padded with an appropriate number of filler entries marked $\perp$ until the end.

  In particular, note that any vertex u with $\deg(u) = 0$ is still assigned a real row, though all entries in that row will be marked $\perp$.

- *Filler rows.* Each filler row stores filler entries marked $\perp$.[11]

The set of rows will eventually be randomly shuffled and stored in a random order.

Reference pointers and continuation pointers. Each row's location is indicated by a pointer ref. For some vertex u, its reference pointer ref_u points to the first row responsible for u. Each real row responsible for some vertex u stores a continuation pointer denoted $\mathsf{ref}_{\mathsf{next}}$, that points to the next row also responsible for u. The continuation pointer $\mathsf{ref}_{\mathsf{next}}$ of the last row responsible for u is set to $\perp$, indicating that u has no more neighbors stored in other rows.

Each filler row's continuation pointer $\mathsf{ref}_{\mathsf{next}}$ points to the next filler row. Finally, we use a global variable $\mathsf{ref}_\perp$ to store the pointer to the next unconsumed filler row. Initially, $\mathsf{ref}_\perp$ points to the first filler row, and when this filler row is consumed, $\mathsf{ref}_\perp$ is updated to its continuation pointer.

5.3.2 Algorithms

Conventions. In all of our algorithms below, if there is an `if`-condition on a secret variable, we assume that both branches will imitate the access patterns of the other branch such that the adversary cannot infer which branch is taken. Also, arrays/tables are 1-indexed.

The initialization algorithm $\mathbf{Init}(1^\lambda, \mathsf{AdjArr}, s)$. The initialization algorithm obliviously prepares the table using the following steps.

1. *Preparation.* Create an empty table $\mathcal{T}$ with $R = 3|V| + 3\left\lceil \frac{|E|}{D} \right\rceil$ rows and D columns.

[11] Notably, a row corresponding to a degree-0 vertex is not classified as a filler row. Though both types of rows contain only filler entries, they will be utilized differently and will follow separate requirements.

2. *Create entries for degree-0 vertices.* We add an entry $(u, \perp, \perp)$ to AdjArr for each vertex $u \in V$ with $\deg(u) = 0$. This can easily be accomplished in $O(1)$ **Sort**s and scans of AdjArr and the list of vertices it comes with.[12] The resulting AdjArr contains all edges and $(u, \perp, \perp)$ entries for any degree-0 u, and is padded with fillers $\perp$ up to length $|V| + |E|$.

3. *Initialize unshuffled table.* Make a linear scan over AdjArr which groups all neighbors of the same originating vertex u together, and assign every entry a row number[13] — specifically, the row number is incremented every time we encounter a new originating vertex or we have encountered D neighbors of the same vertex since last incrementing. Call $\mathbf{BinPlace}^{R,D}(1^\lambda, \cdot)$ to obliviously place each entry of AdjArr into the appropriate row of $\mathcal{T}$.

   At this moment, rows 1 to $\mathsf{numreal}$ for some $\mathsf{numreal} \le |V| + \frac{|E|}{D}$ are the real rows, and all remaining rows are filler.

4. *Determine row permutation and assign pointers.* Call $\mathbf{RandPerm}(1^\lambda, \cdot)$ on the index array $[R]$ to obliviously generate a random permutation denoted π, meaning that eventually row i will be stored at location $\pi(i)$. Every row $i \in [R]$ learns about its own location $\pi(i)$ and sets its continuation pointer $\mathcal{T}[i].\mathsf{ref}_{\text{next}}$: if $i < R$ and rows $i, i+1$ both belong to the same vertex or are both filler rows, then $\mathcal{T}[i].\mathsf{ref}_{\text{next}} \leftarrow \pi(i+1)$; else, $\mathcal{T}[i].\mathsf{ref}_{\text{next}} \leftarrow \perp$. Moreover, in a single scan over π, record the global next filler row pointer $\mathsf{ref}_\perp = \pi(\mathsf{numreal} + 1)$.

5. *Route correct references.* In a linear scan of all rows, create an array refs such that if i is the first row responsible for u, then $\mathsf{refs}[i]$ is set to $(u, \pi(i))$; else $\mathsf{refs}[i]$ is set to $\perp$. Further set $\mathsf{ref}_s \leftarrow \pi(i_s)$ during this scan, where i_s is the first row responsible for s. Treating $\mathsf{refs}[i]$ as the source array, and all entries in the table as the destination array, call $\mathbf{Route}(1^\lambda, \cdot)$ such that each table entry (u, v, w) with $v \neq \perp$ receives the correct ref_v.

6. *Shuffle.* Permute all rows of $\mathcal{T}$ based on their desired destination determined by π, via a call to **Sort**.

7. *Return references.* Return ref_s, and $(\mathsf{ref}_u)_{u \in V}$ as the first V elements of $\mathbf{Sort}(\mathsf{refs})$.

The load algorithm Load(ref). We now describe how to set the iterator's current context to a vertex whose neighbors we want to iterate through.

1. If $\mathsf{ref} = \texttt{nil}$, do nothing (simulate other branches' accesses).
2. Else if $\mathsf{ref} = \perp$, let $\mathsf{col} \leftarrow 0$, let $\mathsf{row} \leftarrow \mathsf{ref}_\perp$, and update $\mathsf{ref}_\perp \leftarrow \mathcal{T}[\mathsf{ref}_\perp].\mathsf{ref}_{\text{next}}$.
3. Else, let $\mathsf{col} \leftarrow 0$ and let $\mathsf{row} \leftarrow \mathsf{ref}$ (interpreted as a row number).

The iterator algorithm NextNb. The **NextNb** algorithm allows one to iterate through the neighbors of the most recently loaded vertex, and is realized as follows.

1. If $\mathsf{row} = \perp$, return $(\perp, \perp, \perp, 1)$.
2. Let $\mathsf{col} \leftarrow \mathsf{col} + 1$, and let $\mathsf{ans} \leftarrow \mathcal{T}[\mathsf{row}][\mathsf{col}]$ (ignoring the originating vertex field).
3. If $\mathsf{col} = D$, update $\mathsf{row} \leftarrow \mathcal{T}[\mathsf{row}].\mathsf{ref}_{\text{next}}$, $\mathsf{col} \leftarrow 0$. If now $\mathsf{row} \in \{\perp, \mathsf{ref}_\perp\}$, let $b \leftarrow 1$, else $b \leftarrow 0$.
4. Else (if $\mathsf{col} \neq D$): if $\mathcal{T}[\mathsf{row}][\mathsf{col}+1] = \perp$, let $b \leftarrow 1$, else $b \leftarrow 0$.
5. Return ans along with b.

[12] Recall the input format of AdjArr given in Section 2.4.

[13] This includes the entry $(u, \perp, \perp)$ of any degree-0 vertex u, so that any such u is still assigned its own row, which should be considered the first (and only) row responsible for u in Step 5. However, after **Init** completes (i.e., in **NextNb**), the entry $(u, \perp, \perp)$ is treated as a filler entry.

5.4 Efficiency

The work and I/O cost of Adj^D operations are stated below. The proof is deferred to Section C.2.

Lemma 5.2 (Adj^D: Efficiency)**.** *Consider a sequence of calls to* Adj^D.**Init**, Adj^D.**Load**, *and* Adj^D.**NextNb** *for graph* $G = (V, E)$*, such that any calls to* Adj^D.**NextNb** *are made consecutively in batches of length at least* k.[14] *Then each* Adj^D *operation incurs the following total work and I/O cost:*

- Adj^D.**Init**$(1^\lambda, \mathsf{AdjArr}, s)$*:* $O((R \cdot D)\log(R \cdot D))$ *work and* $O(\frac{R \cdot D}{B} \log_{M/B} \frac{R \cdot D}{B})$ *I/O cost, where* $R = 3|V| + 3\left\lceil \frac{|E|}{D} \right\rceil$.
- Adj^D.**Load**(ref)*:* $O(1)$ *work and no I/O cost.*
- Adj^D.**NextNb**$()$*:* $O(1)$ *work and amortized* $O\left(\frac{1}{\min\{k, D, B\}}\right)$ *I/O cost.*

6 Oblivious Single-Source Shortest Path

6.1 Additional Preliminaries: Meta-Algorithm for SSSP

Kumar and Schwabe [KS96] proposed a non-oblivious external-memory SSSP algorithm (which we refer to as the *meta-algorithm*) that achieves $O(V + E \cdot \mathsf{IO}_{pq}(E))$ I/O cost, where $\mathsf{IO}_{\mathrm{pq}}(E)$ denotes the per-query I/O cost of a priority queue of capacity E. Our oblivious external-memory SSSP algorithm is based on this meta-algorithm. We now review the necessary background on the meta-algorithm.

Recall that the ordinary Dijkstra's algorithm [Dij59] works as follows:

- Initially, set $\mathsf{visited}[u] = \mathsf{False}$ for all vertices u. Initialize a priority queue PQ containing the entry $(s, 0)$ where s denotes the source vertex, and 0 means that the distance from s to the present vertex (here, s itself) is 0.
- Every iteration, we call $(u, d) \leftarrow \mathsf{PQ}$.**ExtractMin**$()$ to obtain the next unexpanded vertex u closest to the source, and register the distance d_u from s to u as $d_u \leftarrow d$. If $\mathsf{visited}[u]$ is `False`, we expand u by calling PQ.**DecrKey**$(v; d + w(u, v))$ for every v that is a neighbor of u, where $w(u, v)$ denotes the weight of the edge (u, v). At the end of the iteration, set $\mathsf{visited}[u] = \mathsf{True}$.

Using the external-memory prority queue of Chowdhury and Ramachandran [CR04] which achieves amortized $O(\frac{1}{B} \cdot \log \frac{E}{M})$ I/O cost per query, the above algorithm can be realized (non-obliviously) with $O(E + \frac{E}{B} \cdot \log \frac{E}{M})$ I/O cost, where the $O(E)$ part — which we want to get rid of — comes from $O(E)$ accesses to the $\mathsf{visited}$ array. Kumar and Schwabe [KS96] propose a two-priority-queue trick which gets rid of the $\mathsf{visited}$ array, thus improving the I/O cost to $O(V + \frac{E}{B} \log \frac{E}{M})$. Henceforth we call Kumar and Schwabe's algorithm the *meta-algorithm* (shown in Algorithm 1), and our goal is to obliviously emulate the semantic behavior of this meta-algorithm. Since the meta-algorithm is not the final oblivious implementation, we only care about its semantic behavior, and need not worry how to instantiate its internal data structures at this point.

[14] That is, the number of calls to **NextNb** between any calls to **Init** or **Load** is either 0 or at least k.

Algorithm 1: Meta-algorithm for SSSP [KS96, KM03, CR04]

// The time variable is incremented every time it is accessed.

1. PQ_1.**DecrKey**$(s; (0, \mathsf{time}))$ where s is the source vertex.
2. While !PQ_1.**IsEmpty**():
   (a) Let $(u; (d, t)) \leftarrow \mathsf{PQ}_1$.**FindMin**(), $((u', v'); (d', t')) \leftarrow \mathsf{PQ}_2$.**FindMin**().
   (b) If PQ_2.**IsEmpty**() or $(d, t) \leq (d', t')$: Record d as the distance from s to u. Call PQ_1.**ExtractMin**(), and for each neighbor v of u:
   - PQ_1.**DecrKey**$(v; (d + w(u, v), \mathsf{time}))$,
   - PQ_2.**DecrKey**$((u, v); (d + w(u, v), \mathsf{time}))$.

   (c) Else, call PQ_1.**Delete**(u'), and PQ_2.**ExtractMin**().
3. Output the recorded distances for vertices, together with a distance of ∞ for any vertex whose distance was not recorded in the main loop (disconnected from the source).

Intuition: use a second priority-queue as a safeguard. We now explain the intuition of the meta-algorithm. As we expand a vertex u, for every neighbor v of u, we will call **DecrKey**$(v; _)$ — if v does not exist in the priority queue, then a new entry will be created for v. Some neighbors of u (say, v) may already have been expanded earlier; in this case, expanding u may cause a spurious entry for v to be added to PQ_1 (if one does not exist already).

The idea is to maintain a second priority queue PQ_2. PQ_2 serves as a *safeguard*: it ensures that all spurious entries in PQ_1 are deleted before the corresponding vertices (which have already been expanded) get expanded again. To make this work, every time we iterate over the neighbors v of a vertex u, we do the following:

- Call PQ_1.**DecrKey**$(v; (d_u + w(u, v), \mathsf{time}))$, as before, but add the current timestamp time as part of the priority field — the timestamp will be used to break ties when the first term is equal.
- Next, call PQ_2.**DecrKey**$((u, v); (d_u + w(u, v), \mathsf{time}))$, where u is the vertex currently being expanded, d_u is its distance to the source, and time is the current time.

Every time we are ready to expand a new vertex, we will examine the top item in both PQ_1 and PQ_2. We will show that if the top item $((u, v); _)$ in PQ_2 is smaller in priority, it means that v must already have been expanded, which might have caused a spurious entry for u to be added to PQ_1. Thus, we call PQ_1.**Delete**(u) to suppress possible spurious entries for u. If the top item $(u; _)$ in PQ_1 is smaller in priority, we proceed to expand u as normal.

This two-priority-queue implementation still incurs $O(E)$ calls to the priority queues as before, but it avoids the need to dynamically access a visited array.

Lemma 6.1 (Correctness of the meta-algorithm)**.** *Algorithm 1 correctly realizes the SSSP functionality.*

We prove Lemma 6.1 below. In particular, our presentation clarifies the prior literature by explicitly using the timestamp for tie-breaking, which is important for ensuring correctness. This subtlety is often omitted in the description of prior work [KS96, CR04].

Proof of Lemma 6.1 (Correctness). Correctness follows from that of Dijkstra's algorithm if we can show the following two claims about the **Delete** calls issued in Line 2c:

1. For any PQ_1.**Delete**(u') call, u' is a vertex that has already been expanded.
2. Any *spurious entry* $x = (u; (\tilde{d}, \tilde{t}))$ inserted into PQ_1 is deleted before it can be extracted and incorrectly re-expanded.

Claim 1. By inspection, any call to PQ_1.**Delete**(u') is triggered by a *guard entry* $((u', _); _)$ returned by PQ_2.**FindMin**. Such an entry can only have been inserted when u' was being expanded in Line 2b. Hence, any call to PQ_1.**Delete**(u') must correspond to a vertex that has already been expanded.

Claim 2. Let $x = (u; (\tilde{d}, \tilde{t}))$ be a spurious entry inserted into PQ_1 when a neighbor v of u is expanded along edge (v, u, w), where $\tilde{d} = \mathsf{dist}[v] + w$. When u was expanded, the guard entry $((u, v); (d, t^G_{u,v}))$ was inserted into PQ_2, where $d = \mathsf{dist}[u] + w$ and $t^G_{u,v}$ is the value of time at this operation. We show that this guard entry is extracted from PQ_2 after x is inserted into PQ_1 but before x is extracted; this implies the claim due to the PQ_1.**Delete**(u) that occurs alongside the guard entry's extraction (Line 2c).

Consider the call to PQ_1.**DecrKey**$(v; (d, t_{u,v}))$ that occurs when u is being expanded, where $t_{u,v}$ denotes the value of time at this operation. Observe that $(d, t_{u,v})$ is less than the guard entry's priority of $(d, t^G_{u,v})$ by the order of operations in Line 2b, since time is incremented each time it is accessed. After this call, the priority of v in PQ_1 can be *at most* $(d, t_{u,v})$. The comparison rule in Line 2b now implies that v must be expanded—and x inserted into PQ_1—before the guard entry is extracted from PQ_2. Next, since u is finalized before v, $\mathsf{dist}[u] \le \mathsf{dist}[v] \Rightarrow d \le \tilde{d}$ and $t^G_{u,v} < \tilde{t}$. This implies that the guard entry's priority is less than the spurious update x's priority. It follows that the guard entry will get extracted from PQ_2 before x can be extracted from PQ_1. □

We are now ready to present our main algorithm: an I/O-efficient, cache-agnostic, strongly oblivious SSSP algorithm.

6.2 Overview of Our Oblivious SSSP Algorithm

Our starting point is the meta-algorithm [KS96] described in Section 6.1. We design an oblivious version of the meta-algorithm as follows.

First, the meta-algorithm needs to iterate through each vertex's neighbors when exploring the graph; to accomplish this, we use the oblivious adjacency list iterator Adj presented in Section 5. The two priority queues PQ_1 and PQ_2 in the meta-algorithm are instantiated using our new external-memory oblivious priority queue from Section 4. However, replacing the data structures with oblivious counterparts alone is not sufficient for making the meta-algorithm oblivious. The main challenge is that the adversary can still observe which data structure is being called during each time step, and this leaks information about the underlying graph structure.

To remedy this, in our algorithm we devise a fixed schedule for calling the internal data structures Adj, PQ_1, and PQ_2, while preserving correctness of the overall algorithm. Concretely, the schedule S of operation can be summarized as follows, where we simply write PQ_1 and PQ_2 to refer to some operation of each priority queue (as different types of operations are indistinguishable, per Definition 2.2).

$$S = \mathsf{Adj}^D.\textbf{Init}, \mathsf{PQ}_1, \underbrace{\mathsf{PQ}_1, \mathsf{PQ}_2, \mathsf{PQ}_1, \mathsf{PQ}_2, \mathsf{PQ}_1, \mathsf{Adj}^D.\textbf{Load}, X}_{T \text{ times}},$$

where $X = \underbrace{\mathsf{Adj}^D.\textbf{NextNb}, \mathsf{PQ}_1, \mathsf{PQ}_2, \mathsf{PQ}_1, \mathsf{PQ}_2, \mathsf{PQ}_1, \mathsf{PQ}_2}_{D \text{ times}}$.

The number of iterations T and D are fixed functions of $|V|$ and $|E|$: $D = \left\lceil \frac{|E|}{|V|} \right\rceil$ and $T = 2|V| + 2\lceil |E|/D \rceil$. Thus the full schedule depends only on the public values $|V|$ and $|E|$, as per the definition of oblivious SSSP in Section 2.4, and hence reveals no private graph information (e.g., vertex degrees).

To make the fixed schedule idea work, our data structures need to support fake operations that merely imitate the desired access patterns but register no side effect. For the PQs, we fix a special key $\texttt{nil} \in \{0,1\}^w$ such that it is guaranteed no real elements are inserted with key $\texttt{nil}$. We then define the *phantom* priority queue operation **Phantom** as an alias for **Delete**($\texttt{nil}$). The purpose of this phantom operation is to imitate (any) real PQ operations, while leaving the logical state unchanged. Meanwhile, we use the adjacency list iterator's ability to load a filler vertex via Adj^D.**Load**($\perp$) to imitate the access pattern of iterating through a vertex's neighbors via subsequent Adj^D.**NextNb**() calls. Together, these operations ensure the final access pattern leaks no data-dependent information about when different data structures are being used (for real logical operations).

6.3 Detailed Construction of Oblivious SSSP

The full pseudocode of our algorithm is given in Algorithm 2. As in the Adj^D construction, we omit explicit imitation of trivial access patterns (e.g., CPU instructions, cache accesses) across branches of secret conditionals.

Notation. For convenience in presentation of the pseudocode, we define the alias **GuardNext**() for the following helper function, which returns $(\perp, \perp)$ unless the top element of PQ_2 (the "guard") should be processed next to delete some element from PQ_1, in which case the relevant key $(u', \mathsf{ref}_{u'})$ is returned:

- If PQ_1.**IsEmpty** or PQ_2.**IsEmpty**: Return $(\perp, \perp)$.
- Else: $(_; (d, t)) \leftarrow \mathsf{PQ}_1$.**FindMin**(), $((u', \mathsf{ref}_{u'}, _); (d', t')) \leftarrow \mathsf{PQ}_2$.**FindMin**(). If $(d', t') < (d, t)$, return $(u', \mathsf{ref}_{u'})$, else return $(\perp, \perp)$.

Execution overview. We explain Algorithm 2 in words. After initializing the PQ and Adj^D building blocks, the algorithm loops a fixed number of times T. In each iteration:

1. *Vertex loading.* If not currently scanning a vertex's adjacency list (indicated by done), we load the next vertex. If a guard should be processed next (indicated by **GuardNext**()), we load $\perp$ into Adj^D; otherwise, we extract a vertex from PQ_1 and load it.
2. *Neighbor/guard processing.* We call **NextNb** D times. If filler $\perp$ was loaded in this iteration, we process a guard (while **GuardNext** does not output $(\perp, \perp)$) after each **NextNb** call. Otherwise, we relax any real edges outputted by **NextNb**. We pad the PQ operations with **Phantom** if needed, to mimic PQ operations after each of the D **NextNb** calls to maintain the fixed schedule.
3. *Distance setting.* If on the last iteration of a real vertex u (indicated by done), we place u and its priority d in dist; otherwise, we place filler $\perp$.

We will show in the proof of correctness that $T := 2|V| + 2\lceil |E|/D \rceil$ loop iterations is sufficient for all real vertices to be processed. Finally, the algorithm uses the **Dedup** building block to extract

Algorithm 2: Oblivious SSSP Algorithm

Input: Adjacency list AdjArr for $G = (V, E)$. Source vertex $s \in V$. Security parameter λ.

Output: Array $\mathsf{ans} = \{(u, d_u)\}_{u \in V}$ where $\forall (u, d_u)$, d_u is the shortest-path distance from s to u.

// The timestamp time is incremented every time every time it is referenced.

1. Set $D \leftarrow \left\lceil \frac{|E|}{|V|} \right\rceil$, $T \leftarrow 2|V| + 2\left\lceil \frac{|E|}{D} \right\rceil$. Set $u \leftarrow \perp$, $d \leftarrow 0$, $\mathsf{done} \leftarrow \texttt{True}$.
2. Initialize $\mathsf{PQ}_1, \mathsf{PQ}_2$ each as a $\mathsf{PQ}(1^\lambda, |E| + 1)$ and dist as a length-T array.
3. ref_s, $\mathsf{refs} \leftarrow \mathsf{Adj}^D.\mathbf{Init}(1^\lambda, \mathsf{AdjArr}, s)$.[a]
4. $\mathsf{PQ}_1.\mathbf{DecrKey}((s, \mathsf{ref}_s); (0, \mathsf{time}))$.
5. For $i = 1 \to T$:
   (a) If done and $!\mathsf{PQ}_1.\mathbf{IsEmpty}()$ and $\mathbf{GuardNext}() = (\perp, \perp)$:
   Set $((u, \mathsf{ref}_u); (d, _)) \leftarrow \mathsf{PQ}_1.\mathbf{ExtractMin}()$. $\mathsf{Adj}^D.\mathbf{Load}(\mathsf{ref}_u)$.
   Else if done: Set $u, d \leftarrow \perp$. $\mathsf{PQ}_1.\mathbf{Phantom}()$. $\mathsf{Adj}^D.\mathbf{Load}(\perp)$.
   Else: $\mathsf{PQ}_1.\mathbf{Phantom}()$, $\mathsf{Adj}^D.\mathbf{Load}(\texttt{nil})$.
   (b) For $j = 1 \to D$: set $(v, w, \mathsf{ref}_v, \mathsf{done}) \leftarrow \mathsf{Adj}^D.\mathbf{NextNb}()$ and $(u', \mathsf{ref}_{u'}) \leftarrow \mathbf{GuardNext}()$. Then do the following:
   If $u = \perp$ and $u' \neq \perp$:
   $\mathsf{PQ}_1.\mathbf{Delete}((u', \mathsf{ref}_{u'}))$, $\mathsf{PQ}_2.\mathbf{ExtractMin}()$.
   Else if $(v, w) \neq (\perp, \perp)$:
   $\mathsf{PQ}_1.\mathbf{DecrKey}((v, \mathsf{ref}_v); (d + w, \mathsf{time}))$, $\mathsf{PQ}_2.\mathbf{DecrKey}((u, \mathsf{ref}_u, v); (d + w, \mathsf{time}))$.
   Else: $\mathsf{PQ}_1.\mathbf{Phantom}()$, $\mathsf{PQ}_2.\mathbf{Phantom}()$.
   (c) If done and $u \neq \perp$: $\mathsf{dist}[i] \leftarrow (u, d)$, else $\mathsf{dist}[i] \leftarrow \perp$.
6. Return $\mathsf{ans} = \mathbf{Dedup}(\mathsf{dist}, \mathsf{unreach}, |V|)$, where $\mathsf{unreach}$ is the $|V|$-length array $[(u, \infty)]_{u \in V}$.

[a] The return value refs is not used in the algorithm; it is specified only for reference in analysis.

real vertices' distances from dist while assigning distance ∞ to any vertices unreachable from the source, which would not have appeared in the main loop.

Our main result shows that Algorithm 2 is a cache-agnostic and oblivious realization of the SSSP functionality, with total work and I/O cost nearly matching that of the non-private baseline (i.e. the meta-algorithm instantiated with the state-of-the-art non-oblivious priority queue [JL19]).[15]

Theorem 6.2 (External-memory oblivious SSSP)**.** *Consider an undirected graph $G = (V, E)$ with non-negative weights, represented as adjacency list* AdjArr*. Let s denote a source vertex in V. Assume the existence of a pseudorandom function and the standard tall cache assumption that $M \geq B^2$ and $M \geq \log^c \lambda$ for some suitable constant $c > 1$.*

[15] More specifically, up to a $\log\log(E)$ factor, and in the standard regime of $E = \Omega(V)$.

Algorithm 2 is a cache-agnostic, strongly oblivious realization of the SSSP functionality on input $x := (\mathsf{AdjArr}, s)$ *with leakage* $\mathcal{L}(x) = (|V|, |E|)$*, running in* $O((E+V)\log(E+V))$ *work and* $O\left(V + \frac{E+V}{B} \cdot \left(\log \frac{E}{M} + \log_{M/B} \frac{E+V}{B}\right)\right)$ *I/O cost. When* $E = \Omega(V)$*, this is* $O(E \log E)$ *work and* $O\left(V + \frac{E}{B} \log \frac{E}{M}\right)$ *I/O cost.*

The proof of Theorem 6.2 is deferred to Section B. We will separately prove the oblivious simulation and efficiency components of the theorem in Lemma B.1 and Lemma B.11, respectively.

Acknowledgements

Arya was supported by an NSF Graduate Research Fellowship (NSF-DGE2140739). Elaine was supported by an ONR grant and a research grant from 0xPARC. We relied on AI tools, including Gemini 3 and ChatGPT 5.2, only to polish the text and draw figures. We did not use any AI tool for devising the scheme or conceiving the proofs. We gratefully acknowledge Michael Bender for helpful discussions about external-memory algorithms.

References

[ABD+02] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 268–276, New York, NY, USA, 2002. Association for Computing Machinery.

[ACN+20] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *SOSA*, pages 8–14, 2020.

[AEKL25] Gilad Asharov, Eliran Eiluz, Ilan Komargodski, and Wei-Kai Lin. MegaBlocks: Breaking the logarithmic i/o-overhead barrier for oblivious RAM. In *ACM CCS*, 2025.

[AHR24] Ananya Appan, David Heath, and Ling Ren. Oblivious single access machines - a new model for oblivious computation. In *ACM CCS (2024)*, page 3080–3094, New York, NY, USA, 2024. Association for Computing Machinery.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2020*, 2020. To appear. See also: https://eprint.iacr.org/2018/892.

[AKL+22] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. *Optimal Oblivious Parallel RAM*, pages 2459–2521. 2022.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram. In *Theory of Cryptography Conference (TCC)*, 2015.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *STOC'03*, page 307–315, New York, NY, USA, 2003. Association for Computing Machinery.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, August 1973.

[BSA13] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIA CCS*, 2013.

[But] Vitalik Buterin. A maximally simple l1 privacy roadmap. https://ethereum-magicians.org/t/a-maximally-simple-l1-privacy-roadmap/23459.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 2000.

[CCS17] T.-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel ram. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 567–597. Springer, 2017.

[CNS18] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In *TCC (2018)*, volume 11240 of *Lecture Notes in Computer Science*, pages 636–668. Springer, 2018.

[CR04] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *SPAA '04*, page 245–254, New York, NY, USA, 2004. Association for Computing Machinery.

[Dem02] 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.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[DMSY23] Ran Duan, Jiayi Mao, Xinkai Shu, and Longhui Yin. A randomized algorithm for single-source shortest path on undirected real-weighted graphs. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 484–492, 2023.

[DMSY26] Ran Duan, Xiao Mao, Xinkai Shu, and Longhui Yin. A faster directed single-source shortest path algorithm. *arXiv preprint arXiv:2602.07868*, 2026.

[DMWS12] 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.

[FHLS19] Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. In *STOC*, 2019.

[FLPR99] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS'99*, pages 285–297, 1999.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[GMOT12a] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, page 13–24, New York, NY, USA, 2012. Association for Computing Machinery.

[GMOT12b] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA '12*, pages 157–167. SIAM, 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[GWT+23] Tianyao Gu, Yilei Wang, Afonso Tinoco, Bingnan Chen, Ke Yi, and Elaine Shi. Flexway o-sort: Enclave-friendly and optimal oblivious sorting. Cryptology ePrint Archive, Paper 2023/1258, 2023.

[JL19] Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with decreasekeys. In *SODA'19*, pages 1331–1343, 2019.

[JLN19] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *SODA*, 2019.

[JLS21] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues. In *SODA'21*, page 2366–2383, USA, 2021. Society for Industrial and Applied Mathematics.

[KL21] Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious ram (for all parameters). In *Advances in Cryptology – CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV*, page 579–609, Berlin, Heidelberg, 2021. Springer-Verlag.

[KM03] Irit Katriel and Ulrich Meyer. *Elementary Graph Algorithms in External Memory*, pages 62–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[KS96] V. Kumar and E.J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–176, 1996.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *CRYPTO*, 2018.

[LSX19] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, page 359–376, USA, 2015. IEEE Computer Society.

[met] Meta's oram implementation. https://github.com/facebook/oram.

[MPC+18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *IEEE S&P*, pages 279–296, 2018.

[ost] Oblivious stl library. https://github.com/obliviouslabs/rostl.

[Ost24] Benjamin Ostrovsky. Privacy-preserving dijkstra. In *Advances in Cryptology – CRYPTO 2024: 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2024, Proceedings, Part IX*, page 74–110, Berlin, Heidelberg, 2024. Springer-Verlag.

[RFK+15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.

[RS21] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA '21*, page 373–384, New York, NY, USA, 2021. Association for Computing Machinery.

[RTSS09] 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 CCS*, pages 199–212, New York, NY, USA, 2009. ACM.

[Shi20] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *IEEE S&P '20*, pages 842–858, 2020.

[sig] Technology deep dive: Building a faster oram layer for enclaves. https://signal.org/blog/building-faster-oram/.

[SS13] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *S & P*, 2013.

[TGS23] Afonso Tinoco, Sixiang Gao, and Elaine Shi. Enigmap: External-memory oblivious map for secure enclaves. In *USENIX Security*, pages 4033–4050, 2023.

[vEBKZ76] 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.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. CCS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P'15*, pages 640–656, 2015.

[Yan25] Shuyi Yan. Lossless derandomization for undirected single-source shortest paths and approximate distance oracles. *arXiv preprint arXiv:2510.12598*, 2025.

A Obliviously Simulating an External-Memory Machine

In this section, we present an approach for obliviously simulating $\mathsf{RAM}'[N, M, B]$ on a target machine $\mathsf{RAM}[O(N), O(M), B]$. The high-level idea is to rely on the optimal ORAM of Asharov et al. [AKL+22] to implement two ORAMs, called the *external ORAM* and *internal ORAM*, respectively. The external ORAM serves requests to external memory at the block granularity, and the internal ORAM serves requests to words that reside in internal memory (i.e., the CPU's cache). One problem is that the original external-memory machine may leak information through the timing of the external-memory accesses. However, this technicality can be resolved by first compiling the original machine into one that operates on a fixed schedule, henceforth called a *fixed-schedule* RAM. We then obliviously simulate the resulting fixed-schedule RAM. We now elaborate on the details.

Fixed-schedule simulation of the original RAM. On a standard external-memory machine, every instruction is accompanied with an access to internal memory (i.e., the CPU cache). Suppose that executing every instruction and serving a request to an internal ORAM of size M takes takes τ_1 amortized time, and every request to an external ORAM of size N takes $\tau_2 \geq \tau_1$ amortized time.

We will modify the original RAM to have the following schedule by inserting some fake instructions (including accesses to internal memory) or fake requests to external memory:

- Every time we serve $s = \lceil \frac{\tau_2}{\tau_1} \rceil$ instructions including accesses to internal memory, we perform an external-memory read and write — henceforth a group of $\lceil \frac{\tau_2}{\tau_1} \rceil$ instructions plus an external memory access is called a *segment*.
- Pad the program to have exactly $T + \lceil W/s \rceil$ segments.

We will analyze the overhead of this fixed-schedule compilation after we describe the oblivious simulation.

Oblivious simulation of a fixed-schedule RAM. We maintain two ORAM data structures [AKL+22], an external ORAM that stores N/B blocks of size B each, and an internal ORAM that stores M memory words. The internal ORAM is stored inside the CPU cache, so it consumes $O(M)$ amount of CPU cache. The external ORAM is stored in external memory, so it requires N/B blocks of external memory. Additionally, we can spend $O(M)$ amount of CPU cache to act as a private cache for the external-memory ORAM.

Every time the original RAM wants to make a block request to external memory, it is served by the external ORAM. Every time the original RAM wants to access some word residing in internal memory, it is served by the internal ORAM. We use Asharov et al. [AKL+22] to realize both ORAMs. For the external ORAM, we perform an additional optimization to take advantage of the CPU cache. We store the top $O(\log(M/B))$ levels of the hierarchical ORAM in the CPU cache. This way, each external-memory request will incur $O(\log \frac{N}{B})$ instructions and $O(\log \frac{N}{M})$ physical block reads and writes. Similarly, as long as $M \geq \log^c \lambda$ for some suitable constant $c > 1$, then each internal-memory request will incur $O(\log M)$ instructions and $O(\log M)$ word-level accesses.

Thus we have the following theorem.

Theorem A.1 (Oblivious simulation of an external-memory machine)**.** *Assume the existence of pseudorandom functions. Let* $\mathsf{RAM}'[N, M, B]$ *be a fixed-schedule RAM that consumes* W *work and* T *amount of I/O, and suppose that* $M \geq \log^c \lambda$ *for some appropriate constant* $c > 1$*. Then, there exists machine* $\mathsf{RAM}[O(N), O(M), B]$ *that obliviously simulates* $\mathsf{RAM}'[N, M, B]$ *with the leakage* (T, W)*, satisfying the following performance bounds:*

- **I/O cost:** $O(T \cdot \log \frac{N}{M})$.
- **Work:** $O(T \cdot \log \frac{N}{B} + W \cdot \log M)$.

Overhead of the fixed-schedule compilation. If we applied the above oblivious simulation directly to the original RAM that is not necessarily fixed-schedule, then the performance bounds in Theorem A.1 still hold, but the resulting RAM henceforth denoted RAM^* may leak information through the timing of memory accesses. For this reason, we introduced a fixed-schedule compilation step prior to the oblivious simulation. It is not hard to see that this fixed-schedule compilation introduces only a constant factor slowdown relative to RAM^* in terms of the *wallclock running time*. In this sense, *the fixed-schedule assumption can be made without loss of generality.*

B Proofs for Oblivious SSSP

B.1 Security Proofs

Lemma B.1. *Under the assumptions on* M, B *specified in Theorem 6.2, Algorithm 2 obliviously realizes the SSSP functionality* $\mathcal{F}_{\text{SSSP}}$ *with leakage function* $\mathcal{L}((\mathsf{AdjArr}, s)) = (|V|, |E|)$.

For the proof below, let $\mathsf{Sim}_{\text{DEDUP}}(1^\lambda, |\mathsf{Arr}_0|, |\mathsf{Arr}_1|, n')$ denote the simulator that witnesses the obliviousness of the **Dedup** building block. Let $\mathsf{Sim}_{\text{PQ}}(1^\lambda, N, \cdot)$ and $\mathsf{Sim}^D_{\mathsf{Adj}}(1^\lambda, |V|, |E|)$ denote the stateful simulators corresponding to the PQ (Definition 2.2) and Adj^D (Definition 5.1(2)) building blocks respectively. We will use the values $D = \left\lceil \frac{|E|}{|V|} \right\rceil, T = 2|V| + 2\left\lceil \frac{|E|}{D} \right\rceil$ from Algorithm 2 below. Line numbers will refer to line numbers in the pseudocode of Algorithm 2.

Proof of Lemma B.1. Let Alg denote Algorithm 2. We must prove that there exists a poly-time simulator Sim such that for any input $x = (\mathsf{AdjArr}, s)$, the joint distribution $\mathsf{Exec}^{\mathsf{Alg}}(1^\lambda, x)$ is computationally indistinguishable from $\big(\mathcal{F}(x),\ \ \mathsf{Sim}(1^\lambda, |x|, \mathcal{L}(x))\big)$, where $\mathcal{L}(x) := |V|, |E|$ for the graph $G = (V, E)$ represented by AdjArr.

We proceed via a sequence of hybrids. In each hybrid we consider an experiment that produces two random variables comprising the adversary's view: Out (output) and Acc (access pattern).

Experiment SSSPReal**.** This is the real-world execution $\mathsf{Exec}^{\mathsf{Alg}}(1^\lambda, x)$: we run Alg and let Out be the final output ans of the algorithm and Acc be the concatenation of access patterns generated by all instructions in the program.

In subsequent hybrids, we define Out and Acc by describing how the experiment is modified relative to the previous hybrid. Note that changes to the output of an instruction affect the program's future control flow, so such modifications affect not only the final output Out but *also* the final access pattern Acc.

Experiment Hyb_1: Replace **Dedup** with ideal counterpart.

Replace the output of the **Dedup** call with that of the ideal deduplication functionality, and replace the access pattern of **Dedup** in Acc with the access pattern generated by running the simulator $\mathsf{Sim}_{\textsc{Dedup}}(1^\lambda, T, |V|, |V|)$.

Claim B.2. Hyb_1 *is statistically indistinguishable from* SSSPReal.

Proof. Follows directly from the obliviousness of oblivious deduplication **Dedup**. □

Experiment Hyb_2: Replace PQ operations with ideal counterpart.

Replace the output of each PQ operation Q_i with that of the ideal $\mathcal{F}_{\text{PQ}}^N$ operation Q_i, and replace each operation's access pattern in Acc with that of the appropriate simulator call $\mathsf{Sim}_{\text{PQ}}(1^\lambda, N, \cdot)$, initializing one $\mathcal{F}_{\text{PQ}}$ and Sim_{PQ} per PQ ($\mathcal{F}_{PQ1}$ for PQ_1, $\mathcal{F}_{PQ2}$ for PQ_2).

Claim B.3. Hyb_2 *is computationally indistinguishable from* Hyb_1.

Proof. Follows by Corollary 4.5 and the definition of obliviously simulating $\mathcal{F}_{\text{PQ}}$ (Definition 2.2). □

Experiment Hyb_3: Abort if **Init** fails.

Almost the same as Hyb_2, except we now *abort* immediately after Adj^D.**Init** if its output is not *valid* (Definition 5.1(1a)): that is, abort unless **Init** outputs $(\mathsf{refs}, \mathsf{ref}_s)$ where refs contains exactly one reference for each $u \in V$, called the *canonical* reference for u, and ref_s is equal to the canonical reference for s.[16]

Claim B.4. Hyb_3 *is statistically indistinguishable from* Hyb_2.

Proof. Follows by definition of the oblivious adjacency list iterator (Definition 5.1), which requires that the probability that the outputs of **Init** are not valid be negligible in λ, and Lemma C.2. □

We henceforth assume in all future hybrids that **Init** outputs are valid (i.e., the execution aborts otherwise). Further, for clarity we henceforth denote the canonical reference for $u \in V$ by ref_u^*.

Experiment Hyb_4: Enforce canonical ref values and **NextNb** correctness.

Call an invocation of Adj^D.**Load**(ref) in the experiment *valid* if its input ref is $\perp$, `nil`, or the canonical reference ref_z^* for some $z \in V$. In this hybrid, do the following upon each call to Adj^D.**NextNb**: if all calls to Adj^D.**Load** in the experiment so far have been valid, replace the output of this **NextNb** call with the *correct next neighbor output* $(v, w, \mathsf{ref}_v^*, \mathsf{done})$—as defined in Definition 5.1(1b)—for the most recently loaded $u \in V \cup \{\perp\}$.[17]

Note that each Adj^D.**NextNb** call still incurs an access pattern that is appended to Acc, regardless of whether its output is replaced; further, the replacement action has no access pattern itself, but it potentially affects future program flow (and thereby Out and Acc).

Claim B.5. Hyb_4 *is statistically indistinguishable from* Hyb_3.

[16] Formally, we think of an abort as stopping the execution of the program and returning both Out and Acc equal to the string "`abort`."

[17] That is, the vertex $u \in V \cup \{\perp\}$ whose canonical reference was most recently loaded, where we ignore **Load**(`nil`) calls and consider $\perp$ to be the canonical reference for $\perp$.

Proof. Let G_i be the event that the new (possibly replaced) output of the ith **NextNb** call is the same as the old. Observe that conditioned on the event $\cap_{i=1}^{T\cdot D} G_i$, (Out, Acc) are identical in Hyb_3 and Hyb_4, so it suffices to show that $\Pr[\neg(\cap_{i=1}^{T\cdot D} G_i)]$ is negligible in λ.

Consider the ith call to **NextNb**, $i \in [T \cdot D]$. Condition on $\cap_{j=1}^{i-1} G_i$ (if $i > 1$). Observe that if all **Load** calls so far were valid, then the algorithm acts as an environment $\mathcal{Z}$ such that the execution so far conforms to experiment $\mathsf{Expt}^{\mathcal{Z}}$ (Section 5.2). Then, it follows by Definition 5.1(1) that w.p. $1 - \mathsf{negl}(\lambda)$, the old output of **NextNb** is correct. It follows that $\Pr[\neg G_i | G_1, \ldots, G_{i-1}] \leq \mathsf{negl}(\lambda)$, which suffices by union bound since $T \cdot D \leq \mathsf{poly}(\lambda)$. □

We now show the following claims about Hyb_4 to set up the remainder of the proof.

Claim B.6 (Valid **Load**s)**.** *All* Adj^D.**Load** *calls in the* Hyb_4 *experiment are valid.*

Proof. This is seen by induction. The first call is **Load**(ref_s^*), by **Init** validity. Then, if the first $i-1$ **Load**s are valid, all **NextNb** queries before the ith **Load** are correct and return neighbor and reference of the form (v, ref_v^*) or $(\perp, \perp)$. Thus the contents of $\mathcal{F}_{PQ1}$ are all of the form $((z, \mathsf{ref}_z^*); _)$ for some $z \in V$, which implies that the ith **Load** is valid. □

Claim B.7 (SSSP correctness)**.** *The sequence of values* $\{(u, _); (d, _)\}$ *extracted from* $\mathcal{F}_{PQ1}$ *in the* Hyb_4 *experiment matches the values* $\{(u, d)\}$ *extracted in the meta-algorithm (Algorithm 1). In particular, the output* Out *matches the output of the meta-algorithm.*

Proof. Note that *all* **NextNb** outputs are correct in Hyb_4 by Claim B.6, and $\mathcal{F}_{\mathrm{PQ}}$.**Phantom**() = $\mathcal{F}_{\mathrm{PQ}}$.**Delete**(`nil`) calls have no affect on Out. The proof is then straightforward by comparing the executions of Hyb_4 and the meta-algorithm. We highlight the key points to observe:

1. Compared to the meta-algorithm's `while` loop, the Hyb_4 execution relaxes edges and processes guards in batches of size D per iteration, but it continues correctly as needed in the next iteration via the done bit.

   In particular, the number of iterations $T := 2|V| + 2\lceil |E|/D \rceil$ is sufficient to ensure all vertices are processed: letting $\deg_i$ be the degree of the ith vertex processed and ℓ_i be the number of guards to process between the $(i-1)$-th and ith vertices, the number of iterations needed is $\sum_{i=1}^{|V|} \lceil \ell_i / D \rceil + \lceil \deg_i / D \rceil \leq 2(|V| + |E|/D)$, since the total number of guards inserted is $|E|$.

2. The final deduplication step adjoins distances $\{(u, \infty)\}$ for precisely the vertices u that do not already have an entry $(u, _)$ in dist. In particular, this is the set of vertices unreachable from s, which are not processed in the main loop.

□

Claim B.8 (Admissible environment)**.** *The algorithm in* Hyb_4 *defines an* admissible *environment* $\mathcal{Z}$ *(Section 5.2) such that the full* Hyb_4 *experiment conforms to* $\mathsf{Expt}^{\mathcal{Z}}$*.*

Proof. First, the Hyb_4 execution conforms to $\mathsf{Expt}^{\mathcal{Z}}$ for some environment $\mathcal{Z}$, precisely because all **Load** calls are valid (Claim B.6). We now observe why the environment $\mathcal{Z}$ is admissible.

By inspection of the algorithm, exactly D **NextNb** calls are made in between **Load** calls, and $T \cdot D$ **NextNb** calls are made in total. Since outputs of **NextNb** are always replaced by the *correct* outputs, the done bit is always set correctly, such that **NextNb** is called the right number of times after **Load**(ref_u^*) or **Load**($\perp$). Finally, the vertices u for which **Load**(ref_u^*) is called (Line 5a, first branch) are *distinct*: this follows by Claim B.7, since the vertices extracted by the meta-algorithm are distinct by its correctness (Lemma 6.1). □

Experiment Hyb_5: Replace Adj^D access patterns with simulator.

Almost the same as Hyb_4, except we modify (only) Acc as follows. Let `str` denote the result of running the Adj^D simulator $\mathsf{Sim}^D_{\mathsf{Adj}}(1^\lambda, |V|, |E|)$ (Definition 5.1(2)). Define $\mathsf{Sim}^D_{\mathsf{Adj}}.\mathsf{Next}()$ to output the next unconsumed segment of `str` up to the next delimiter, as specified in the access pattern of $\mathsf{Expt}^{\mathcal{Z}}$ (Section 5.2). At each call to an Adj^D operation, replace the operation's access pattern in Acc with the simulated access pattern $\mathsf{Sim}^D_{\mathsf{Adj}}.\mathsf{Next}()$.

Claim B.9. Hyb_5 *is statistically indistinguishable from* Hyb_4.

Proof. This follows by the security of the oblivious adjacency list iterator, stated in Definition 5.1(2), which can be invoked because the Hyb_4 experiment conforms to $\mathsf{Expt}^{\mathcal{Z}}$ for an *admissible* environment $\mathcal{Z}$ (Claim B.8).[18] □

Experiment SSSPIdeal. This is the ideal-world execution: Out is given by $\mathcal{F}_{\mathrm{SSSP}}(x)$, and Acc is given by $\mathsf{Sim}(1^\lambda, |V|, |E|)$, where the simulator Sim is given by $\mathsf{Sim}_{\mathrm{SSSP}}$ defined below.

Claim B.10. SSSPIdeal *is identically distributed to* Hyb_5.

Proof. The output of SSSPIdeal is deterministic, so it suffices to separately consider Out and Acc. Out is identical between Hyb_5 and Hyb_6 by Claim B.7. Further, observe that the calls that generate the access pattern Acc of the Hyb_5 experiment do not depend on secret data,[19] so they can be simulated knowing only $|V|$ and $|E|$, as specified in simulator $\mathsf{Sim}_{\mathrm{SSSP}}$ below. □

Simulator $\mathsf{Sim}_{\mathrm{SSSP}}(1^\lambda, |V|, |E|)$. We omit simulation of trivial access patterns (e.g., CPU instructions; simple cache accesses). We assume that $\mathsf{Sim}_{\mathrm{SSSP}}$ maintains a counter $j \leftarrow 0$ for each of the two $\mathsf{Sim}_{\mathrm{PQ}}$ instances, and we then use the notation $\mathsf{Sim}_{\mathrm{PQ}}.\mathsf{Next}()$ to mean that first the corresponding counter j is incremented, and then $\mathsf{Sim}_{\mathrm{PQ}}(1^\lambda, N, j)$ is called. We also use the shorthand $\mathsf{Sim}_{\mathrm{Guard}}()$ as an alias for the sequence of calls $\mathsf{Sim}_{\mathrm{PQ1}}.\mathsf{Next}()$, $\mathsf{Sim}_{\mathrm{PQ2}}.\mathsf{Next}()$, $\mathsf{Sim}_{\mathrm{PQ1}}.\mathsf{Next}()$, $\mathsf{Sim}_{\mathrm{PQ2}}.\mathsf{Next}()$, corresponding to the PQ operations in the **GuardNext** alias.

□

B.2 Efficiency

The proof of Theorem 6.2 is now completed with a straightforward analysis of Algorithm 2's efficiency. Below, we use $\mathrm{sort}(n)$ to denote the optimal sorting bounds of $O(n \log n)$ work and $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/O cost, as is standard in external-memory literature, and we abuse the notation V, E to denote the sizes $|V|, |E|$. Note that the standard tall cache assumption $M \geq B^2$ is necessary to achieve optimal $\mathrm{sort}(n)$ I/Os cache-agnostically [BF03].

Lemma B.11 (Algorithm 2: Efficiency)**.** *Under the assumptions on M, B specified in Theorem 6.2, Algorithm 2 completes in $O\left(V + \frac{E+V}{B} \log \frac{E}{M}\right) + \mathrm{sort}(E+V)$ I/Os and $\mathrm{sort}(E+V)$ work.*

Proof. We analyze the cost of each component of the algorithm. Below, note that $T = 2V + 2\lceil E/D \rceil = O(V)$, so $T \cdot D = O(E+V)$; similarly $R = 3V + 3\lceil E/D \rceil = O(V)$ and $R \cdot D = O(E+V)$.

[18] Note that it is important that the access pattern of $\mathsf{Expt}^{\mathcal{Z}}$ is defined to include delimiters, as the boundaries between Adj^D operation are observable in our algorithm (due to other types of instructions, e.g. priority queue operations, being interleaved between Adj^D operations).

[19] In particular, all calls that generate access patterns for priority queue and Adj^D operations can be factored out of conditional branches on secret data at this point.

Simulator $\mathsf{Sim}_{\mathrm{SSSP}}$

Inputs: $|V|, |E|$; security parameter λ.

1. Define $D \leftarrow \left\lceil \frac{|E|}{|V|} \right\rceil$, $T \leftarrow 2|V| + 2\left\lceil \frac{|E|}{D} \right\rceil$. Initialize dist array of length T.
2. Instantiate $\mathsf{Sim}^D_{\mathsf{Adj}}(1^\lambda, |V|, |E|)$ and two $\mathsf{Sim}_{\mathrm{PQ}}(1^\lambda, E+1, \cdot)$ simulators $\mathsf{Sim}_{\mathrm{PQ1}}, \mathsf{Sim}_{\mathrm{PQ2}}$.
3. Run $\mathsf{Sim}^D_{\mathsf{Adj}}.\mathsf{Next}()$. `// for` Adj^D**.Init**
4. Run $\mathsf{Sim}_{\mathrm{PQ1}}.\mathsf{Next}()$. `// for` PQ_1**.DecrKey**
5. For $i = 1 \to T$:
   (a) Run $\mathsf{Sim}_{\mathrm{GUARD}}()$. `//  for` **GuardNext**
   (b) Run $\mathsf{Sim}_{\mathrm{PQ1}}.\mathsf{Next}()$. `// for` PQ_1**.ExtractMin**
   (c) Run $\mathsf{Sim}^D_{\mathsf{Adj}}.\mathsf{Next}()$. `// for` Adj^D**.Load**
   (d) For $j = 1 \to D$:
      i. Run $\mathsf{Sim}_{\mathsf{Adj}}.\mathsf{Next}()$. `// for` Adj^D**.NextNb**
      ii. Run $\mathsf{Sim}_{\mathrm{GUARD}}$. `// for` **GuardNext**
      iii. Run $\mathsf{Sim}_{\mathrm{PQ1}}.\mathsf{Next}()$, then $\mathsf{Sim}_{\mathrm{PQ2}}.\mathsf{Next}()$. `// for` PQ_1, PQ_2 `ops in Line 5b`
   (e) Access $\mathsf{dist}[i]$.
6. Run $\mathsf{Sim}_{\mathrm{DEDUP}}(1^\lambda, T, |V|, |V|)$.

- dist *operations*: The cost of these array operations (Lines 5c, 6) is dominated by $\mathrm{sort}(R+V) = \mathrm{sort}(V)$ work and I/Os.
- Adj^D *operations*: By inspection of the loop in Line 5b, Algorithm 2 satisfies Lemma 5.2 with $k = D$. In total, Adj^D**.Init** is called once; Adj^D**.Load** is called T times; and Adj^D**.NextNb**() is called $T \cdot D$ times. Lemma 5.2 then implies that the resulting total cost is $\mathrm{sort}(E+V) + O(T) + O(T \cdot D)$ work and $\mathrm{sort}(E+V) + O(T \cdot D \cdot \frac{1}{\min\{D,B\}})$ I/Os.

  This is $\mathrm{sort}(E+V)$ work and $O(V) + \mathrm{sort}(E+V)$ I/Os, observing that $O\left(T \cdot D \cdot \frac{1}{\min\{D,B\}}\right) = O(V + \frac{E}{B})$ and that $\frac{E}{B}$ is dominated by $\mathrm{sort}(E+V)$.
- PQ *operations*: In each of the T iterations of the outermost loop, we execute $O(D)$ PQ operations (from PQ_1 and PQ_2), so we have $O(E+V)$ PQ operations in total. By the PQ work and I/O bounds (Corollary 4.5) with $N := E + 1$,[20] in total this is

$$O\left(\frac{E+V}{B} \log \frac{E}{M}\right) \text{ I/Os and } (E+V)\log E \text{ work.}$$

Summing these components yields the claimed bounds. □

[20] Note that this is a valid upper bound as at most $E+1$ keys are ever inserted into either PQ: E real keys for PQ_2 and one phantom key `nil`.

C Proofs for Oblivious Adjacency List Iterator

C.1 Correctness and Security Proofs

Here we prove that our Adj^D construction is an oblivious adjacency list iterator. We start by showing that, if the output of each oblivious building block is replaced with the ideal output, Adj^D.**Init** constructs a table $\mathcal{T}$ satisfying the desired properties (Section 5.3.1) and returns the correct reference pointers. Throughout, we set $R := 3|V| + 3\lceil |E|/D \rceil$, as in the **Init** algorithm.

Lemma C.1 (Adj^D.**Init** outputs)**.** *Suppose the outputs of all calls to* **BinPlace**, **RandPerm**, **Route**, **Sort** *are replaced with their ideal functionalities. Let $\pi : [R] \to [R]$ denote the output of the random permutation call. Then* Adj^D.**Init** *constructs a $(R \times D)$ table $\mathcal{T}$, with set of rows shuffled according to π, whose unshuffled version satisfies the following properties (T1) – (T6):*

(T1) Each row stores edges of a single vertex $u \in V$ or is a filler *row, which consists entirely filler entries $\perp$.*[21]

(T2) At least $T := 2|V| + 2\left\lceil \frac{|E|}{D} \right\rceil$ rows are filler rows.

(T3) For each vertex $u \in V$, its edges appear in $n_u := \max\{1, \lceil \deg(u)/D \rceil\}$ consecutive rows $\{r_u, \ldots, r_u + n_u - 1\}$, with neighbors stored in the same order as in AdjArr *and the last row padded with filler elements $\perp$ if needed.*

(T4) Each non-$\perp$ entry (u, v, w) in $\mathcal{T}$ carries reference pointer $\mathsf{ref}_v = \pi(r_v)$, i.e. the shuffled location of the first row responsible for v.

(T5) For each $u \in V$, for each row $r_u + k$ $(0 \le k < n_u)$, the continuation pointer ref_{next} equals $\pi(r_u + k + 1)$ if $k < n_u - 1$ and $\perp$ otherwise.

(T6) Letting $y_1 < \cdots < y_Y$ denote the pre-shuffle filler rows, for each y_k, $\mathsf{ref}_{next} = \pi(y_{k+1})$ if $k < Y$ and $\perp$ otherwise.

Moreover, the returned list refs *is $\{(u, \pi(r_u))\}_{u \in V}$; the returned value ref_s is $\pi(r_s)$; and the initial value of $\mathsf{ref}_\perp$ is $\pi(y_1)$.*

Proof of Lemma C.1. We refer to the steps in the **Init** algorithm (Section 5.3.2). Step 6 shuffles the table according to π, so it suffices to examine the unshuffled version from Steps 1–5 for (T1)–(T6).

Properties (T1), (T3) follow directly from Step 3, noting that Step 2 ensures that a row will be created for any vertex of degree 0. Observe that the highest assigned row number (denoted numreal) in Step 2 is at most $|V| + \lceil |E|/D \rceil$, so there are at least $R - \mathsf{numreal} = 2|V| + 2\lceil |E|/D \rceil$ filler rows, proving (T2). Properties (T5), (T6) follow from Step 4. Property (T4) follows from Step 5. Finally, the claim on the returned list refs follows by Step 7; $\mathsf{ref}_s = \pi(r_s)$ by Step 5; and $\mathsf{ref}_\perp = \pi(y_1) = \pi(\mathsf{numreal} + 1)$ by Step 4. □

We now prove the main lemma of this section, stated below.

Lemma C.2. *The Adj^D construction specified in Section 5.3.2 is an oblivious realization of an adjacency list iterator (Definition 5.1).*

[21] Recall that the row for a vertex u with $\deg(u) = 0$ is treated as a real row, not a filler row, though it does not contain any real edges.

We separately prove the correctness (Definition 5.1(1)) and security (Definition 5.1(2)) claims. In both arguments, in place of the experiment $\mathsf{Expt}^{\mathcal{Z}}$ (Section 5.2), we work in the hybrid experiment $\mathsf{Hyb}^{\mathcal{Z}}$ where outputs of all oblivious building blocks ($\mathbf{BinPlace}, \mathbf{RandPerm}, \mathbf{Route}, \mathbf{Sort}$) are replaced by the outputs of their ideal functionalities, and their access patterns are replaced by those of the corresponding simulators (denoted $\mathsf{Sim}_{\mathrm{BP}}, \mathsf{Sim}_{\mathrm{RP}}, \mathsf{Sim}_{\mathrm{Route}}, \mathsf{Sim}_{\mathrm{Sort}}$).

By definition of each building block's obliviousness, the outputs of all calls and the overall access pattern in $\mathsf{Hyb}^{\mathcal{Z}}$ have negligible statistical distance from those in the original experiment $\mathsf{Expt}^{\mathcal{Z}}$. Thus it suffices to prove the desired correctness and security claims for $\mathsf{Hyb}^{\mathcal{Z}}$, in which Lemma C.1 can be invoked as its hypothesis is now satisfied.

Proof of Lemma C.2, Correctness (Definition 5.1(1)). Consider experiment $\mathsf{Expt}^{\mathcal{Z}}(1^\lambda)$ for some $\mathcal{Z}$ and consider the corresponding hybrid experiment $\mathsf{Hyb}^{\mathcal{Z}} := \mathsf{Hyb}^{\mathcal{Z}}(1^\lambda)$, as defined above. We show that the **Init** validity and **NextNb** correctness claims hold (with probability 1) in this hybrid.

1. **Init** *validity.* Immediate from Lemma C.1: refs is the length-$|V|$ list $(u, \pi(r_u))_{u \in V}$ and $\mathsf{ref}_s = \pi(r_s)$ (for π as defined in Lemma C.1).

2. **NextNb** *correctness.* Ignoring all Adj^D.**Load**(`nil`) calls, which do not affect the logical state, consider any **Load** query with some input $z \in V \cup \{\perp\}$. We have two cases.

   (i) If $z = \perp$, then Adj^D.**Load**($\perp$) sets $\mathsf{row} = \mathsf{ref}_\perp$ and updates $\mathsf{ref}_\perp \leftarrow \mathcal{T}[\mathsf{ref}_\perp].\mathsf{ref}_{\mathrm{next}}$. This means row is either set to a filler row or $\perp$, by Lemma C.1(T6). It follows by inspection that all subsequent **NextNb** calls before the next **Load** correctly return $(\perp, \perp, \perp, 1)$.[22]

   (ii) If $z \in V$, then Adj^D.**Load**(ref_z) sets $\mathsf{row} = \pi(r_z) \in [R]$, the (shuffled) location of the first row in $\mathcal{T}$ for z. By Lemma C.1 (T5), continuation pointers $\mathsf{ref}_{\mathrm{next}}$ are set such that subsequent **NextNb** calls will scan $\mathcal{T}[\pi(r_z)], \mathcal{T}[\pi(r_z + 1)], \ldots$, one entry at a time.

   By Lemma C.1 (T1), (T3), and (T4), each scanned entry $(z, v, w, \mathsf{ref}_v)$ corresponds to the correct next edge of z with correct reference $\mathsf{ref}_v = \pi(r_v)$, and the done-bit b is set correctly by inspection of **NextNb**.[23] Note that the last row of z has $\mathsf{ref}_{\mathrm{next}} = \perp$ (T5), so any **NextNb** queries after exhausting all rows of z return $(\perp, \perp, \perp, 1)$. □

Proof of Lemma C.2, Security (Definition 5.1(2)). Consider experiment $\mathsf{Expt}^{\mathcal{Z}}$ for *admissible* environment $\mathcal{Z}$ (Section 5.2), and consider the corresponding hybrid $\mathsf{Hyb}^{\mathcal{Z}}$. Recall that the access pattern of the experiment is defined by the access patterns generated by all calls to Adj^D.**Init**, Adj^D.**Load**, and Adj^D.**NextNb**, with delimiters between each operation. We show that the access pattern can be simulated by a simulator $\mathsf{Sim}^D(1^\lambda, |V|, |E|)$ that depends only on $|V|, |E|$ and D.

1. *Initialization.* It is straightforward to see that the access pattern of the singular Adj^D.**Init** call can be simulated: aside from the calls to $\mathsf{Sim}_{\mathrm{BP}}, \mathsf{Sim}_{\mathrm{RP}}, \mathsf{Sim}_{\mathrm{Route}}$ and $\mathsf{Sim}_{\mathrm{Sort}}$, the access patterns of each step are deterministic, and all access patterns depend only on $|V|, |E|, D$.

2. **Load** *queries.* By admissibility (A1), **Load** is invoked (possibly with input `nil`) according to a fixed schedule—always after exactly D calls to **NextNb**. This implies that the overall access pattern due to Adj^D.**Load** calls can be simulated, as the access pattern of each Adj^D.**Load**($\cdot$) is trivial.

[22] Reading a filler entry $\perp$ in $\mathcal{T}$ is interpreted as producing $\mathsf{ans} = (\perp, \perp, \perp)$.

[23] Indeed, for a degree-0 vertex u, the entries scanned in row $\pi(r_u)$ will all have $v, w, \mathsf{ref}_v = \perp, \perp, \perp$ as desired.

3. **NextNb** *queries.* Admissibility conditions (A1) and (A3) imply that **NextNb** is invoked in $T := 2|V| + 2\lceil |E|/D \rceil$ batches of D queries each. We claim that the corresponding access pattern is identical to scanning a uniformly random T-subset of the rows of $\mathcal{T}$, one row per batch and one entry per query.

   To see this, note first that at the start of each batch of D queries, $\mathsf{ref}_\perp$ points to an *unconsumed* (i.e., not yet scanned) filler row, by inspection of the **Load** algorithm and Lemma C.1(T6). Call this property ($\star$).

   Now consider any **Load** query with input $u \in V \cup \{\perp\}$.

   - If $u = \perp$, there is exactly one batch of D **NextNb** calls until the next such **Load** by admissibility (A2); these scan the row $\mathsf{ref}_\perp = \pi(y_k)$ for some filler row y_k as defined in Lemma C.1.
   - If $u \in V$, then there are at most $n_u := \max\{1, \lceil \deg(u)/D \rceil\}$ batches of **NextNb** calls until the next such **Load** by (A2). As in the correctness proof, it follows by Lemma C.1 that the ith such batch ($i \le n_u$) scans row $\pi(r_u + i - 1)$ of $\mathcal{T}$.[24]

   Crucially, admissibility (A4) guarantees that the queried vertices $u \in V$ are *distinct*. Thus it follows from the above, via (A4) and ($\star$), that the T rows scanned over the T batches correspond to *distinct* pre-shuffle indices $r^{(1)}, \ldots, r^{(T)}$. Since π is a uniform random permutation sampled independently in **Init**, the shuffled indices $\pi(r^{(1)}), \ldots, \pi(r^{(T)})$ form a uniformly random T-subset of $[R]$.[25]

   Thus, the access pattern of all **NextNb** calls depends only on $|V|$, $|E|$, and D: the simulator can sample a uniform permutation $\sigma : [R] \to [R]$ and emit the access pattern corresponding to scanning rows $\sigma(1), \ldots, \sigma(T)$, one row per batch of D queries, interleaving the fixed per-query accesses and delimiters. □

C.2 Efficiency

Proof of Lemma 5.2. The work and I/O cost of each operation is analyzed below. We use the notation $\mathrm{sort}(n)$ as before, and the notation $\mathrm{scan}(n)$ to denote the asymptotic cost of scanning an array of length n, i.e. $O(n)$ work and $O(\lceil \frac{n}{B} \rceil)$ I/Os, as is standard in the external-memory literature.

- Adj^D.**Load**: The work and I/O bounds are both immediate.
- Adj^D.**NextNb**: The work bound is immediate. To see the I/O bound, recall the hypothesis that **NextNb** calls are made in batches of size at least k. In any batch of $k' \ge k$ consecutive calls, entries are scanned linearly by increasing col in a given row. A new block is only possibly fetched when jumping to a new row (which occurs only when $\mathsf{col} = D$) or after crossing a block boundary (if $D > B$). Thus every $\min\{k', D, B\}$ calls incur only $O(1)$ I/Os, implying an amortized cost of $O\left(\frac{1}{\min\{k', D, B\}}\right) = O\left(\frac{1}{\min\{k, D, B\}}\right)$ I/Os.

[24] In particular, the variable row in the Adj^D construction never becomes $\perp$ in these batches by admissibility (A2) and the properties of $\mathcal{T}$ (Lemma C.1), so each batch of **NextNb** queries indeed scans a valid (unconsumed) row in $\mathcal{T}$, whether real or filler.

[25] Note that π is sampled once at initialization and remains independent of all environment queries. In particular, the environment's queries may reveal the location of some rows (i.e., images under π), but the remaining, unrevealed parts of π stay uniformly random and independent of the environment's choices. Hence the access pattern distribution does not depend on the specific sequence of queried vertices.

- Adj^D.**Init**: Each building block **BinPlace**, **RandPerm**, **Route**, and **Sort** can be realized in $\text{sort}(n)$ work and I/Os on input arrays of size n (Section 3.1). We can then bound each step of Adj^D.**Init** as follows, viewing the table $\mathcal{T}$ as a contiguous 1-dimensional array stored in row-major order (rows laid out sequentially in memory).

  Step 1: Incurs $\text{scan}(R \cdot D)$ work and I/Os.

  Step 2: Dominated by **Sort** on (augmented) AdjArr, hence $\text{sort}(|V| + |E|)$ work and I/Os.

  Step 3: Dominated by the $\mathbf{BinPlace}^{R,D}$ call, which is $\text{sort}(R \cdot D)$ work and I/Os.

  Step 4: Calls **RandPerm** on $[R]$, which is $\text{sort}(R)$ work and I/Os. The rest of the step can be accomplished with $O(1)$ scans through $\mathcal{T}$, i.e. $\text{scan}(R \cdot D)$ work and I/Os.

  Step 5: Calls **Route** on arrays of size R and $R \cdot D$, hence $\text{sort}(R \cdot (D+1))$ work and I/Os.

  Step 6: Calls **Sort** on input array of size $R \cdot D$, hence $\text{sort}(R \cdot D)$ work and I/Os.

  Step 7: Dominated by $\text{sort}(R)$ work and I/Os.

  Note that $R \cdot D$ is always $\Omega(|V| + |E|)$. Thus the overall cost of **Init** is asymptotically $\text{sort}(R \cdot D)$ work and I/Os. □

D Proofs for Oblivious Deduplication

We now prove the obliviousness of our oblivious deduplication for randomly shuffled inputs.

Experiment DedupReal**.** Run the real experiment as in the obliviousness definition above. The real experiment faithfully runs the real-world Dedup algorithm. The output of the experiment is the final deduplicated array along with the access patterns observable by the adversary.

Experiment Hyb_0**.** Run the Dedup algorithm as in DedupReal; however, every time the algorithm makes a call to HT.**Build**, HT.**Lookup** or HT.**Extr**,

- replace the outcomes of the queries (referenced later in the algorithm) with the ideal answers generated according to the HTIdeal experiment; and
- replace the access patterns of HT with the simulated access patterns generated according to the HTIdeal experiment, where the simulator knows the input length n_0.

Claim D.1. Hyb_0 *is computationally indistinguishable from* DedupReal*.*

Proof. Observe that the experiments take a shuffled input array $\mathcal{F}_{\text{shuffle}}(\mathsf{Inp}_0)$ as input. Now, by the obliviousness of the hash table HT, Hyb_0 is computationally indistinguishable from DedupReal. □

Experiment Hyb_1**.** Almost the same as Hyb_0, except that we now replace the outcome of $\mathsf{Intersperse}(\mathsf{Old}, \mathsf{New})$ in Step 3 with the ideal outcome $\mathcal{F}_{\text{shuffle}}(\mathsf{Old}||\mathsf{New})$, and replace the access patterns of Intersperse with simulated access patterns, where the simulator knows n_0 and n_1.

Claim D.2. Hyb_1 *is computationally indistinguishable from* Hyb_0*.*

Proof. Observe that in Hyb_1, since HT's outputs and access patterns have been replaced with the ideal counterparts, the Old array which contains the unvisited items of HT must be randomly shuffled. Further, New is randomly shuffled since Inp_1 is randomly shuffled. Therefore, the indistinguishability of Hyb_1 and Hyb_0 directly follows from the obliviousness of Intersperse. □

Experiment Hyb_2**.** Almost the same as Hyb_1 except that we now replace the answers to HT'.**Build**, HT'.**Lookup** and HT'.**Extr** with the ideal answers generated according to the $\mathsf{HTIdeal}$ experiment. Moreover, replace the access patterns of HT' with the simulated access patterns generated according to the $\mathsf{HTIdeal}$ experiment, where the simulator knows the input length $n_0 + n_1$.

Claim D.3. Hyb_2 *is computationally indistinguishable from* Hyb_1.

Proof. Recall that in Hyb_1, we replaced the **Intersperse** in Step 3 with an ideal counterpart. At this moment, its output Comb must be randomly shuffled. Therefore, the indistinguishability of Hyb_2 and Hyb_1 follows directly from the obliviousness definition of the hash table HT'. □

Experiment Hyb_4**.** Almost the same as Hyb_3 except that we now replace the outcome of the oblivious sort and the oblivious random permutation with an ideal outcome, and replace their access patterns with simulated ones.

Claim D.4. Hyb_3 *is computationally indistinguishable from* Hyb_2.

Proof. Follows directly from the obliviousness of the oblivious sort and oblivious random permutation. □

Experiment Hyb_4**.** Almost the same as Hyb_3 except that we replace the outcome of $\mathsf{Intersperse}(\mathsf{Old}, \mathsf{New})$ in Step 6 with the ideal outcome $\mathcal{F}_{\mathsf{shuffle}}(\mathsf{Old}||\mathsf{New})$, and replace the access patterns of $\mathsf{Intersperse}$ with simulated access patterns, where the simulator knows the input lengths $n_0 + n_1$ and $2\log\lambda$.

Claim D.5. Hyb_4 *is computationally indistinguishable from* Hyb_3.

Proof. Recall that in Hyb_2, we replaced HT' with an ideal counterpart, its output Old in Step 6 must be randomly shuffled. Further, the New array in Step 6 is also randomly shuffled because New has the same ordering as $\mathsf{CombStash}$, which is the output of an ideal shuffle (recall that we replaced the oblivious random permutation with an ideal shuffle in Hyb_3. Because both the input arrays to **Intersperse** are randomly shuffled, the indistinguishability of Hyb_4 and Hyb_3 now follows directly from the obliviousness of the primitive. □

Experiment Hyb_5**.** Finally, replace the outcome of $\mathsf{IntersperseRF}(\mathsf{Comb}')$ in the last step with $\mathcal{F}_{\mathsf{shuffle}}(\mathsf{Comb}')$, and replace the access patterns of $\mathsf{IntersperseRF}$ with simulated access patterns where the simulator knows the input length n'.

Claim D.6. Hyb_5 *is computationally indistinguishable from* Hyb_4.

Proof. Recall that in Hyb_4 we replaced the **Intersperse** in Step 6 with an ideal counterpart. Therefore, its output Comb' is randomly shuffled. Recall that **Compact** is a deterministic algorithm with deterministic access patterns. Therefore, in the compacted result, all the real items must be in a random order. Now, the indistinguishability of Hyb_5 and Hyb_4 follows directly from the obliviousness definition of **IntersperseRF**. □

Experiment $\mathsf{DedupIdeal}$**.** It is not hard to see that Hyb_5 can be equivalently viewed as $\mathsf{DedupIdeal}$. First, since all building blocks are now replaced with ideal counterparts, it is not hard to verify that the deduplicated outcome must be correct and randomly shuffled. Second, observe that all building blocks' access patterns are now simulated; moreover, the remaining access patterns are deterministic and can be trivially simulated. In other words, Hyb_5 implicitly defines a simulator Sim that simulates the access patterns knowing only λ, n_0, n_1, and n'.