Privacy-Preserving Dijkstra

Benjamin Ostrovsky

California Institute of Technology
Pasadena CA, USA

June 19, 2024

Abstract

Given a graph $G(V, E)$, represented as a secret-sharing of an adjacency list, we show how to obliviously convert it into an alternative, MPC-friendly secret-shared representation, so-called d-NORMALIZED REPLICATED ADJACENCY LIST (which we abbreviate to *d-normalized*), where the size of our new data-structure is only 4x larger – compared to the original (secret-shared adjacency list) representation of G. Yet, this new data structure enables us to execute oblivious graph algorithms that simultaneously improve underlying graph algorithms' round, computation, and communication complexity. Our *d-normalization* proceeds in two steps:

- First, we show how for any graph G, given a secret-shared adjacency list, where vertices are arbitrary alphanumeric strings that fit into a single RAM memory word, we can efficiently and securely rename vertices to integers from 1 to V that will appear in increasing order in the resulting secret-shared adjacency list. The secure renaming takes $O(\log V)$ rounds and $O((V + E) \log V)$ secure operations. Our algorithm also outputs in a secret-shared form two arrays: a mapping from integers to alphanumeric names and its sorted inverse. Of course, if the adjacency list is already in such an integer format, this step could be skipped.

- Second, given a secret-shared adjacency list for any graph G, where vertices are integers from 1 to V and are sorted, there exists a *d-normalization* algorithm that takes $O(1)$ rounds and $O((V + E))$ secure operations.

We believe that both conversions are of independent interest. We demonstrate the power of our data structures by designing a privacy-preserving Dijkstra's single-source shortest-path algorithm that simultaneously achieves $O\left((V + E) \cdot \log V\right)$ secure operations and $O(V \cdot \log V \cdot \log \log \log V)$ rounds. Our secure Dijkstra algorithm works for any adjacency list representation as long as all vertex labels and weights can individually fit into RAM memory word. Our algorithms work for two or a constant number of servers in the honest but curious setting. The limitation of our result (to only a constant number of servers) is due to our reliance on linear work and constant-round secure shuffle.

Keywords: Oblivious Graph Algorithms, MPC, Oblivious RAM, Distributed ORAM, Garbled RAM, Single-Source Shortest Path, Secure Dijkstra.

*Preliminary version will appear at CRYPTO 2024 conference, to be held in Santa Barbara, CA, August 18-22, 2024.
E-mail: *ben.ostrovsky@caltech.edu*. Work done while at Santa Monica High School, Santa Monica, CA

Contents

1 Introduction

1.1 Secure Multi-Party Computation

Secure multi-party computation (MPC) allows two or more servers to jointly compute an arbitrary polynomial-time computable function on private data while learning only the size of the inputs and the output of the function and nothing else. These notions were invented in the 1980s both in the computational [49, 83] and in the information-theoretic [13, 31, 73] settings. In the last decade, implementations of MPC have attracted considerable attention [80, 84, 61, 76, 74, 36, 14, 34, 67, 53, 20, 85, 65, 57, 78, 41].

While initial work from the 1980s considered secure computation protocols solely for circuits (either Boolean or Arithmetic), it was recognized in 1997 that Random Access Memory (RAM) in the MPC setting could be realized to make MPC protocols more efficient [71]. Follow-up works, adopting Oblivious RAM [50] and GRAM [63] to MPC support of RAM has gained much attention over the last decade. Integrating ORAM with MPC gave rise to distributed ORAM (DORAM) [71, 62, 36, 24, 41]. Applications of GRAM to MPC to achieve RAM access for MPC were considered in [63, 45, 64, 54, 82, 55].

1.2 Secure Graph Processing

The ability to perform random access is especially relevant for secure graph processing algorithms. Not surprisingly, graph processing has received considerable attention in the MPC literature [6, 60, 61, 5, 17, 7, 21, 82]. Privacy-preserving graph processing has followed two common approaches:

(a) **Adjacency Matrix Approach.** Instead of a (secret-shared) adjacency list representation, convert graph representation to the adjacency matrix representation, which hides the degree of any node by scanning the entire row of the adjacency matrix[1].

(b) **ORAM/GRAM Compilation Approach.** Choose an insecure graph algorithm and compile it into a secure version using Distributed ORAM and/or Garbled RAM inside MPC.

We discuss both approaches separately. Jumping ahead, we will show a third alternative approach: in near linear work, obliviously convert any graph into a new representation that we define, the so-called d-NORMALIZED REPLICATED ADJACENCY LIST and then modify insecure graph algorithms to run on this new representation in an oblivious fashion. The conversion works for *any* graph, yet the resulting representation "acts" as if any graph is d-regular. As a side remark, showing (by a counting argument) that this data structure exists is easy; what is hard is to show how to convert any secret-shared adjacency list representation into such a format securely and efficiently. That is the main contribution of our paper. We demonstrate the use of the new representation on Dijkstra's shortest path algorithm. As part of the new privacy-preserving Dijkstra, we design a privacy-preserving parallel distributed priority queue algorithm (in support of Dijkstra).

1.2.1 Review of Adjacency Matrix Approaches for Graph Processing:

Adjacency matrix approaches were taken in [6, 4, 5, 17]. We give a brief overview of these approaches here. Blanton et al. [17] consider an adjacency matrix representation to solve several graph algorithms, namely breadth-firth search (BFS), Single Source Shortest Path (SSSP) for an unweighted graph using BFS,

[1] A variant of this approach is to reveal the graph's maximum degree and truncate rows of the adjacency matrix representation to the graph's maximum degree. Since we are unwilling to reveal any information about the graph (other than the total size of all vertices and all edges combined), we do not consider such a leakage acceptable.

minimum spanning tree (MST), and maximum flow. They achieve $O(V^2)$ work complexity for BFS, SSSP, and MST, as well as $O(V^3 \cdot E \log V)$ work for maximum flow. Anagreh et al. [7] presented a privacy-preserving implementation of Prim's algorithm to solve MST, with $O(V \log V)$ rounds and $O(V^2)$ work. They also generalize their MST algorithm to work for the minimum spanning forests. Aly et al. [5, 4, 6] designed a secure SSSP, where in order not to disclose the degree of any vertex, they upper-bound it to be the maximum $n-1$ degree. By permuting the adjacency matrix, they achieve $O(V^3)$ secure operations [6]. Aly et al. revisited the SSSP problem several times in [4, 5]. Specifically, they achieve secure Dijkstra with $O(V^2 \log V)$ secure operation and $O(V^2)$ rounds.

1.2.2 Review of ORAM/GRAM Compilers for Graph Processing:

Approach (b) using Distributed ORAM to support random access inside MPC was introduced in [71]. ORAM compilation for graph processing was explored in [60, 61, 82]. Specifically, Keller et al. [60] applied ORAM compilers to insecure Dijkstra, building all the necessary data structures to support it. Since the ORAM Compilers (of various building blocks) were not as developed as they are now, in 2014 [60] required $O(V \log^4 V + E \log^5 V)$ secure operations and rounds. Liu et al. [61] extended the ideas of Keller at al. using their ObliVM ORAM framework, as well as two modifications: (1) loop coalescing and (2) avoiding weight updating. Loop coalescing made Dijkstra run in one loop, with a secret shared value telling it whether it was processing an edge or a vertex, as opposed to an inner and outer loop for vertex and edges, respectively. This allowed them to avoid padding the vertices to the maximum degree while keeping the topology. (i.e., the degree of each vertex) of the graph secret. Their second change, avoiding decrease-key weight updating, was replacing the decrease-key step in the priority queue with an insert of a new item into the priority queue with a smaller weight. This increases the number of vertices in the PRIORITY QUEUE. With these changes, they achieve $O((V+E)\log^2 V)$ secure operations. (Note that our bound is $O((V+E)\log V)$).

It is important to examine *generic transformations* approach from Distributed Oblivious RAM (DORAM) [62, 84, 38, 28, 24, 41, 56, 69] and Garbled RAM (GRAM) [63, 45, 44, 54, 55] for RAM style algorithms. Note that the latest DORAM does provide an addressable memory with private read/write capabilities with logarithmic overhead in the running time and logarithmic round complexity [41, 56] and even sub-logarithmic overhead for large blocks [69]. However, a general compiler from arbitrary code to addressable memory adds another level of inefficiencies, such as implementing and supporting pointers and recursion and hiding which operation is performed at any particular step. For example, hiding which operation the CPU executes requires multiplexing the general-purpose CPU for all its instructions and doing it for each computation step. This alone results in considerable additional overhead [61]. Furthermore, handling pointers and recursive program stack (if used) has to be explicitly programmed – for general MPC compilers, this leads to additional difficulties.

Recent progress on GRAM application was achieved in Yang et al. [82]. They present a variable instruction set architecture (VISA), a method of handling programs inside MPC, where all strait-line fragments of the code are unrolled into individual "custom" CPU instructions that are executed as garbled circuits, and the latest Garbled RAM [55] is used for obvious random access. Since [55] incurs $O(\log^3 n \cdot \log\log n)$ overhead, and [82] use $O((V+E)\log V)$ insecure Dijkstra's with the HEAP for Priority Queue, the result is $O((V+E)\log^4 V \log\log V)$ secure operations and constant rounds.

Currently, ORAM overhead is far are more efficient than GRAM overhead; thus, if we want the smallest overhead possible at the expense of non-constant rounds, we should examine ORAM compilers' application to various insecure SSSP solutions. For example, if we consider asymptotically the most efficient insecure Dijkstra algorithm, which has $O(V \log V + E)$ running time [43, 23, 22], and compile it into secure DORAM

we get a logarithmic additional overhead for such a compilation (which is currently the best-case scenario [41, 56] for small blocks). That is, the naive compilation solution gets $O((V \log V + E) \log V)$ secure operations and the same number of rounds. Even assuming $O(\log n / \log \log n)$ ORAM overhead for larger blocks [1, 69], we get $O((V \log V + E) \log V) / \log \log V)$ secure operations.

In this paper, we achieve $O((V + E) \log V)$ secure operations for Dijkstra, which is an improvement even compared to the ORAM generic compilers in terms of the overhead. We also improve on the naive approach's *round complexity* of $O((V \log V + E) \log V / \log \log V)$, for ORAM compliers, to $O(V \log V \log \log \log V)$ rounds.

As a side remark, to improve round-complexity, in addition to [82] GRAM option, one can also consider running a secure variant of the parallel version of Bellman-Ford algorithm, similar to [7]. Specifically, recall that Bellman-Ford relaxes all E edges in parallel V times. There are implementations of Parallel ORAM read/write compilers of [29, 11] which incur logarithmic overhead. Thus, one can use parallel Bellman-Ford with $O(V \log V)$ round complexity, but at the expense of $O(V \cdot E \cdot \log V)$ secure operations. This paper aims to minimize the overhead and the round complexity *jointly*.

1.2.3 Graph Processing Algorithms that leak Partial Information:

Brickell et al. [21] present a secure SSSP algorithm. However, they use a modified definition of security, in which their algorithm immediately reveals the identity of a start node, and, during the execution of the algorithm, gradually reveals the identity and distance for each "explored" vertex in G. As already noted by Aly et al. [6], Brickell et al.'s work is unusable as a building block for a larger secure computation, as it leaks all the distances vector and additional structural information about the graph during the algorithm execution. In contrast, all inputs and outputs are secret-shared in our work, allowing our protocol to be used as a secure subroutine of a larger protocol.

Another solution that leaks partial information about the graph is that of [8]. Specifically, [8] considers the so-called Radius-Stepping algorithm, where there is some graph structure leakage. The authors argue that such leakage can be further masked by running an algorithm for a longer number of iterations that could be determined experimentally. However, the bounds on the amount of masking to achieve provable guarantees that do not leak any information about the graph are not analyzed. In this paper, we aim only at algorithms that have provable guarantees on the running time and round complexity and no leakage whatsoever, including not leaking the identity of the start node.

Solutions for graphs with fixed bounded degrees are also irrelevant to our goals. For example, [81] considers only planar graphs (e.g., Manhattan distances), and as such, their methods are not applicable to graphs with no restrictions on the individual vertex degrees. Observe that for general graphs, some vertices could have small degrees or even be isolated vertices, and some could have huge degrees. In our work, we make no assumption on the vertex degree distribution.

1.3 Organization of the rest of the paper

In section 2, we provide a high-level overview of our results and introduce the notion of d-NORMALIZED REPLICATED ADJACENCY LIST. Section 3 covers preliminaries and reviews and introduces various building blocks that we use. Section 4 presents two new oblivious graph conversion algorithms. Specifically, it shows an oblivious way to rename all vertices to be sorted integer values 4.3. Section 4 also shows how to convert an adjacency list into a *d-normalized* form 4.1, which is only four times bigger than the original adjacency list size. Section 5 presents our secure algorithm for the Parallel Priority Queue. Section 6 presents our secure Dijkstra algorithm and its analysis. Finally, section 7 presents conclusions and further work.

2 Overview of Our Results

Our algorithms apply to two-party, three-party, or constant-party MPC protocols. We consider honest-but-curious MPC, and consider the following primitives, all used in a block-box fashion:

- Arithmetic Black Box (ABB) secure operations. Consistent with previous works, our complexity ignores the difference between different types of ABB operations. ABB operations include addition, multiplication, exponentiation, comparison, and bit-decomposition on secret shared values, where both inputs and outputs are secret-shared and assumed to fit into RAM word size.
- We use a secure shuffle protocol with constant rounds and linear work. These are only known for two, three, or a constant number of servers [28, 30].
- We use linear work secure merge algorithms [39, 19]. Again, these are known only for two, three, and a constant number of servers.
- We use DORAM and Parellel DORAM. One option to deploy DORAM inside MPC for any number of servers by using client-server ORAM [70, 72, 10, 37] and Parallel DORAM of [64, 11], and run ORAM client within MPC. However, the constants involved in [10, 11] and even [37] are enormous. For two-party and three-party servers, custom solutions for DORAM that do not incur these enormous constants [62, 64, 69] have been developed.
- In our priority queue and in vertex renaming, we use an oblivious secure sort algorithm. For instantiation, for three-party cases, one can use the honest-but-curious secret-shared SECURE SORT algorithm of [9], which is based on radix sort. It takes $O(n \log n)$ work and the number of rounds proportional to the length of the key. Since we will be dealing with keys that are vertex names from 1 to V and are $O(\log V)$ in length, that gives us logarithmic round complexity[2]. For two or more servers, one can also use the shuffle-then-sort paradigm of [52, 51], which gets $O(\log n)$ overhead oblivious sort protocol with a small constant, unlike AKS.

As far as the collusion threshold goes, as long as the underlying building blocks listed above are resilient to such collusions, our overall protocols can support the same type of collusion as well – by relying on the stand-alone composition framework of Canetti [25] (we remark that we don't need more general UC composition [26] since we only consider honest-but-curious setting.).

2.1 Changing Graph Representations

We start our exposition with a simple counting argument. Specifically, given any graph G, represented as an adjacency list, we show that by replicating vertices and padding adjacency lists with $\perp$, any graph G can be converted into a graph on $2n$ (potentially repeating) vertices, such that each potentially copied vertex has (partial) adjacency list that is exactly twice the average degree of the original graph, padded with $\perp$'s as needed. We call such representation of the graph d-NORMALIZED REPLICATED ADJACENCY LIST. The main contribution of this paper is a secure algorithm that converts any secret-shared adjacency list into a secret-shared d-normalized adjacency list, which is only four times larger than the initial secret-shared adjacency list representation. We now get into the details.

[2]Of course, the same asymptotic result can be achieved using AKS [3] sorting network, with evaluating each comparison gate bit by bit under MPC, but at the cost of an enormous constant in the big-O notation. For additional discussion on secure sort implementations, see [40].

We pick a parameter d, which is about twice the average degree of the graph, rounding up. Observe that we know the average degree since we know the overall number of vertices and the total number of edges, and this information is not private. For vertices with fewer than d edges, we will obliviously pad the adjacency list of each such vertex with $\perp$ symbols so that the linked list will be of length exactly d. For vertices with more than d outgoing edges, we will allow the same vertex name to appear (consecutively) multiple times on our new adjacency list representation. Each copy of the vertex will then have exactly d edges on its adjacency list, again padding as needed.

Definition 1. d-Normalized Replicated Adjacency List. *For any graph G, we define a* d-NORMALIZED REPLICATED ADJACENCY LIST *to be an adjacency list representation of G where any vertex u may appear multiple times in the adjacency list with the following properties:*

1. *For any edge $(u, v) \in G$, v appears in at least one replicated adjacency list of u.*
2. *Each adjacency list of u is of length exactly d, containing either vertices of G or $\perp$.*
3. *If there is more than one copy of v's edges in the replicated adjacency list, all copies of v's appear consecutively to each other.*
4. *Multiple $\perp$ entries with d-length $\perp$'s for its adjacency lists are allowed.*

For example, and ignoring edge weights for now, if a vertex u has outgoing edges to vertices (b, c, d, e, g), then a standard adjacency list representation will include a linked list: $[u, (b, c, d, e, g)]$. The 2 -REPLICATED ADJACENCY LIST for u could be: $[u, (b, c)]$, $[u, (d, e)]$, $[u, (g, \perp)]$, $[u, (\perp, \perp)]$, where all copies of u must appear consecutively. As another example, the 3-NORMALIZED REPLICATED ADJACENCY LIST for u could be: $[u, (b, c, e)]$, $[u, (d, g, \perp)]$. Alternatively, it could even be: $[u, (b, c, e)]$, $[u, (d, \perp, \perp)]$, $[u, (g, \perp, \perp)]$. $[u, (\perp, \perp, \perp)]$. We remark that condition 4 allows $\perp$ entries in the replicated adjacency lists. For example, in the 2-normalized adjacency list, entries of the form $(\perp, (\perp, \perp))$ are allowed.

We consider two cases for the adjacency list representation of any graph $G(V, E)$:

- **Sorted Integer Representation:** vertices are integers from 1 to V and appear in increasing order in the adjacency list.
- **Alphanumeric Vertex Representation:** vertices are arbitrary alphanumeric strings, with the restriction that each vertex label fits into a single RAM memory word.

We first show how to convert alphanumeric vertex representation to sorted integer representation:

Theorem 1. (Oblivious Graph Renaming) *For c servers, assume the existence of honest-but-curious* SECURE SHUFFLE *protocol with linear work and $O(1)$ rounds, resilient against any collusion of at most $u < c$ servers. Further, assume that a c-server MPC protocol exists for ABB operations, tolerating at most $u < c$ colluding servers. Given an adjacency list $\mathcal{A}$ for any graph $G(V, E)$, where vertices are arbitrary alphanumeric labels that fit into a single memory word, there exists a secure algorithm for c servers tolerating $u < c$ collusions to convert $[\![\mathcal{A}]\!]$ into $[\![\hat{\mathcal{A}}]\!]$, where $\hat{\mathcal{A}}$ is an adjencecy list of G with vertices that are ordered integers from 1 to V. The conversion algorithm takes $O(\log V)$ rounds and $O((V + E) \log V)$ secure operations. The algorithm also outputs a secret-sharing of the mapping from ordered integers to original alphanumeric labels and from sorted alphanumeric labels to integers.*

Our Oblivious Graph Renaming algorithm may be of independent interest in multiple applications. For example, knowledge graphs and Privacy-Enhancing Technologies (PETs) often work on graphs where alphanumeric labels represent data. In information science, ontology graphs are used with alphanumeric vertex labels. Our algorithm allows us to efficiently and privately convert such graphs to integers from 1 to V. Since we also provide a secret-sharing of a mapping from integers back to alphanumeric labels, after whatever secure computation is computed on integer-labeled graph representation, one can convert it back to its original format.

d-normalization: Our main result (that relies on a counting argument in lemma 6) shows how to obliviously convert the adjacency list of G, which is represented as integers from 1 to V into *d-normalized* secret-shared representation, which is only four times bigger than its original adjacency list. We again assume as our building block a secure shuffle protocol. As of this writing, secure shuffles exist only for two, three, or a constant number of honest-but-curious non-colluding servers. Our protocols rely on secure shuffle in a black-box manner. If, in the future, secure shuffles with linear work and constant round complexity are developed for a larger number of servers, our theorems will apply automatically in these settings as well (see formal definition in section 3.2). We also assume Arithmetic Black Box operations (ABB for short – see section 3.1) with the same collusion threshold. We state of main result on d-normalization :

Theorem 2. (Secure d-Normalization) *For c servers, assume the existence of honest-but-curious* SECURE SHUFFLE *protocol with linear work and $O(1)$ rounds, resilient against any collusion of at most $u < c$ servers. Further, assume that there exists a c-server MPC protocol for ABB operations, tolerating at most $u < c$ colluding servers. Assume that c servers are given a secret-shared adjacency list $[\![\mathcal{A}]\!]$ for any graph $G(V, E)$, where vertex labels are integers from 1 to V and appear in $\mathcal{A}$ in increasing order. Then, there exists an honest-but-curious secure algorithm, tolerating at most u collusions, to obliviously convert $[\![\mathcal{A}]\!]$ into $\frac{2E}{V}$-*normalized *replicated adjacency list of size $4[\![\mathcal{A}]\!]$. The conversion takes $O(1)$ rounds and $O(V+E)$ secure operations.*

We remark that in the theorems 1 and 2, we do not rely on SISO-PRF as one of the ABB (see 3.1). Therefore, the above results for three or a constant number of parties are unconditional.

2.2 Secure Dijkstra

Given a secret-sharing of the start vertex s, a secret-sharing of a directed weighted graph G with non-negative weights, our next objective is to compute Dijkstra's shortest path algorithm securely. More specifically, the goal is to compute the Single Source Shortest Path (SSSP). That is, we wish to compute a secret-sharing of a vector containing the numerical value for the shortest path from the source to each vertex in G, called the SSSP distance vector, without learning anything about G other than the total number of vertices and edges of G. Just like regular Dijkstra, our algorithm can also securely compute a secret-sharing of a *predecessor* for each vertex. We stress an important requirement in our problem statement: in order for our SSSP protocol to be useful as a subroutine for other secure protocols, the distance and predecessor vectors, which is the output of Dijkstra, must be computed in a secret-shared form and should remain hidden[3].

Our secure Dijkstra algorithm *does* requires evaluation of SISO-PRF with a constant number of rounds. Naturally, using constant-round MPC, one can achieve it assuming only the existence of one-way functions

[3]This is in contrast to the work of [21], where the entire vector of distances is revealed in the clear (in fact, in the specific order in which Dijkstra computes distances) for their algorithm to work.

and running PRF [48] under MPC[4]. But are one-way functions strictly necessary? It turns out that we do not even need the full power of SISO-PRF for our application. Instead, what we actually need is very limited SISO-PRF, where inputs are restricted to be integers from 1 to V; such limited SISO-PRF does exist without any additional cryptographic assumptions in the three (or more) server settings. The catch, however, is that these functions are based on recursive position-map ideas from [77] and require a logarithmic number of rounds instead of a constant. For that reason, these SISO-PRFs are not applicable if we care about round complexity (and we do). We remark, however, that it could still be the case that for $c \geq 3$ there exists another constant-round small-domain SISO-PRF without any cryptographic assumptions, but this is wide open. We are now ready to state our theorem on the Privacy-Preserving Single-Source Shortest Path (SSSP):

Theorem 3. (Secure Dijkstra) *Let k be a security parameter and $G(V, E)$ be a directed weighted graph with non-negative edges, where all weights and vertex names fit into a single RAM memory word. Assume the existence of honest-but-curious* SECURE SHUFFLE *protocol with linear work and $O(1)$ rounds for c servers, resilient against any collusion of at most $u < c$ servers. Further, assume that a c-server MPC protocol exists for ABB operations, including SISO-PRF, tolerating at most $u < c$ colluding servers. Then, given as input a secret-sharing among c servers of a start vertex and secret-sharing of G among c servers, there exists c-server honest-but-curious* SSSP *protocol tolerating at most u collisions with $O((V + E) \cdot \log V)$ secure operations and $O(V \cdot \log V \cdot \log\log\log V)$ rounds, where all secure operations are bounded by a fixed polynomial in k number of steps.*

We compare theorem 3 on privacy-preserving Dijkstra to the previous work in table **1** below:

Secure Single Source Shortest Path Schemes

Secure SSSP	Secure Operations	Round Complexity
Aly et al. (2013) [6]	$O(V^3)$	$O(V^2 \log V)$
Keller et al. (2014) [60]	$O(V \log^3 V + E \log^4 V)$	$O(V \log^3 V + E \log^4 V)$
Liu et al. (2015) [61]	$O((V + E) \log^2 V)$	$O((V + E) \log^2 V)$
Aly et al. (2022) [5]	$O(V^2 \log V)$	$O(V^2)$
Yang et al. (2023) [82]	$O((V + E)\, log^4 V \, \log\log V)$	$O(1)$
Naive Solution using DORAM	$O\left(\frac{(V \log V + E) \log V}{\log\log V}\right)$	$O((V \log V + E) \log V)$
This paper	$O((V + E) \log V)$	$O(V \cdot \log V \cdot \log\log\log V)$

Table 1: *Comparison of our work to the previous works, counting the number of Arithmetic Black Box (ABB) operations. By ABB, we mean secure addition, multiplication, and comparison, and for two-server case, secure public-key operations, such as encryption, decryption, and homomorphic addition.*

To develop a secure Dijkstra algorithm, we must address the MPC variant of the priority queue algorithm, which is an integral part of Dijkstra. There are several works addressing secure priority queues in the literature in the client-server ORAM model, where the client is trusted. There is no trusted client in the MPC setting, and MPC support to simulate the client presents new challenges. Our starting point is Jafargholi et al. [58, 59] client-server priority queue, which we modify for MPC purposes. We choose Jafargholi et al. [58, 59] paper which achieves $O(\log n)$ overhead in the client-server setting over a follow-up paper of [75], since [75] requires $O(\log n + \log(1/\delta))$ client's memory, where δ denotes the failure probability per request, and therefore $\log(1/\delta)$ has to be asymptotically greater than $O(\log n)$ for the probability of failure

[4]For 3-server case, an efficient implementation of this was shown in [41].

to be negligible. For MPC, we needed to keep the client's memory as small as possible since we support it inside MPC. We prove the following theorem for Oblivious Priority Queues (OPQ)[5]:

Theorem 4. *(***Secure OPQ)** *Assume the existence of honest-but-curious* SECURE SHUFFLE *protocol with linear work and $O(1)$ rounds for c servers, resilient against any collusion of at most $u < c$ servers. Further, assume that there exists a c-server MPC protocol for ABB operations, including SISO-PRF, tolerating at most $u < c$ colluding servers. Further, assume that all elements and priorities fit into a single RAM memory word of OPQ. Then, there exists c-server honest-but-curious OPQ protocol tolerating at most $u < c$ collisions, supporting n elements with the amortized cost of $O(\log n)$ secure operations and $O(\log n \cdot \log\log\log n)$ rounds for each of* EXTRACT-MIN*,* INSERT*, and* PARALLEL-DECREASE-KEY *OPQ-procedures, where all secure operations are bounded by a fixed polynomial in k number of steps.*

As a side remark, an alternative for Secure Priority Queue is to use pointer-based worst-case Fibonacci Heap of Brodal et al. [23] together with MPC ORAM compilers. However, this is still worth more than our solution: it gets us $O((V \log V + E) \log V)$, which is clearly inferior to our version of secure Dijkstra, which takes $O((V + E) \log V))$ secure operations. In addition, general compilation loses round efficiency. On a related note, it is important to point out that standard Fibonacci Heap [43] is not secure even if decrease-key operations are implemented within ORAM and, therefore, hides memory access if the time when the call to Fibonacci Heap and when it terminates is revealed to the adversary. The main reason [43] is not secure, even if run as a subroutine inside ORAM, is that it has only *expected* $O(1)$ run-time for each decrease-key operation. Therefore, the run-time of each decrease key varies depending on the graph structure and may leak information about the graph. In contrast, in our solution, we do not need to hide when different subroutines begin and end since each subroutine has a fixed amount of work independent of the graph topology.

2.3 Applications of secure SSSP:

Data-oblivious SSSP has multiple applications that we list below. In addition, we believe that our graph representation will be applicable to other graph algorithms since it hides the degree of the vertices yet maintains the efficiency of adjacency-graph representation.

- **Privacy-Preserving Navigation Systems**: [81, 79, 35]: In a navigation system, the shortest path information between locations might be sensitive for some users. Data-oblivious algorithms can ensure that the shortest path computation does not reveal any private information about the user's starting point, destination, or frequently visited locations.
- **Secure Graph Processing**: [6]: In distributed graph processing, where multiple servers collaboratively analyze graphs without revealing sensitive information, data-oblivious SSSP algorithms can ensure that no party can infer the private attributes or structure of the graph.
- **Computational Biology**: Computing edit distance between two DNA sequences (without revealing DNA individual inputs) can be implemented using the privacy-preserving SSSPalgorithm as a building block [12, 15]. As another example, sequence alignment problems in computational biology can be solved securely using secure shortest path algorithms [42].

[5]We remark that we call EXTRACT-MIN, INSERT, and PARALLEL-DECREASE-KEY as *procedures* instead of operators in order not to confuse these with secure operations that we use to implement these primitives.

- **Social network analysis**: Social network graphs, such as Facebook friends or Twitter graphs [66], often require secure shortest-path calculations.

Beyond applications listed above, the SSSP secure computation is explored in outsourcing mobile computations to the cloud [27]. In addition, Privacy-Preserving Knowledge Graphs [32] must hide information about these graphs, including keeping various sensitive ontologies private, yet being able to compute on these siloed datasets.

3 Preliminaries

Recall that two samplable families of distributions $\{X_n\}$ and $\{Y_n\}$ are *computationally indistinguishable* $\{X_n\} \stackrel{c}{\approx} \{Y_n\}$ [83] if for all constants c, and for all probabilistic poly-time adversaries $\mathcal{A}$, and for all sufficiently large n, $|\Pr[x \leftarrow X_n : \mathcal{A}(x) = 1] - \Pr[y \leftarrow Y_n : \mathcal{A}(y) = 1]| < \frac{1}{n^c}$. We call Random Access Memory (RAM) algorithm ALG *oblivious* if for any two inputs of the same length, ALG takes the same number of steps, and the distribution of Random Access Memory (RAM) locations accessed is computationally indistinguishable.

We denote by k the computational security parameter (e.g., the size of the key for block-cipher, such as AES.) As standard for RAM architectures [2], each server is modeled as a RAM Machine that can read/write a single word of RAM in one step and perform CPU operations in one step. Each word of RAM is of size $O(k)$, and thus, cryptographic keys can fit into one (or a constant number of) word(s) of RAM memory. We assume that for graphs that servers are working with, vertex labels and edge weights also fit into a single (or a constant number of) words of RAM.

We use lower-case letters to represent variables and parameters, except in the case of V and E, by which denote the vertices and edges of a graph G, respectively. We abuse the notation and refer to the *number* of vertices and the number of edges of a graph as V and E as well, instead of $|V|$ and $|E|$. Arrays are denoted with capital letters. The i'th element of array Y is denoted Y_i. We use the following definition [68]:

Definition 2 (Secret-Sharing)**.** *Let there be s servers, and let U represent the maximal corruptible subsets of $\{0, \ldots, s-1\}$. Let Share be a randomized function that maps $X \rightarrow X_s$ and let Reconstruct be a deterministic function that maps $X_s \rightarrow X$. Share and Reconstruct define a U-secure secret-sharing scheme if $\forall x \in X$, Reconstruct(Share(x)) = x with probability 1 and $\forall u \in U$, $\forall x_a, x_b \in X$, Share$(x_a)_u$ and Share$(x_b)_u$ have the same distribution, where Share$(\cdot)_u$ is the view of any U servers.*

We use $[\![x]\!]$ to denote a secret-sharing of x. $[\![x]\!]$ is a *fresh* secret-sharing of x if its random distribution which is independent of all previous distributions. When we write $[\![A]\!]$ for an array A, it denotes secret-sharing every of every *entry* of A. Note that what this does not hide is the dimension of A, which is public.

We investigate a scenario in which several (two or three or a constant number of) servers have a secret sharing of an adjacency list of a graph and a secret sharing of a start vertex and wish to compute Dijkstra's Single Source Shortest Path (SSSP). Only the total number of edges E and vertices V is publicly revealed. Servers do not know the graph's topology (meaning the degree of any vertex or maximum degree of the graph). At the end of the protocol, the distance function and shortest path should be secret-shared.

We use MPC ideal functionality as an arithmetic black box $\mathcal{F}_{ABB}$, which was first introduced by Damgård and Nielsen [33], and generalized in [56] to define primitive secure operations.

3.1 Arithmetic Black Box functionalities

Concretely, we invoke $\mathcal{F}_{ABB}$ with the following functionalities where all values are in the publicly known finite field $\mathbb{F}$:

- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.add($[\![x]\!], [\![y]\!]$), which adds secret shared values x, y to obtain a fresh secret shared value z s.t. $z = x + y$. Note that we can trivially add more than two values by adding a third to the sum of the first two. We assume that addition can be done non-interactively, sometimes referred to as "silent" addition.
- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.multiply($[\![x]\!], [\![y]\!]$), which takes secret shared values x, y to obtain a fresh secret shared value z s.t. $z = x * y$.
- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.exp($[\![x]\!], [\![y]\!]$), which takes secret shared values x, y to obtain a fresh secret shared value z s.t. $z = x^y$.
- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.min($[\![x]\!], [\![y]\!]$), which takes the minimum between values x, y to obtains a fresh secret shared value of $z \leftarrow \min(x, y)$.
- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.max($[\![x]\!], [\![y]\!]$), which takes the minimum between values x, y to obtains a fresh secret shared value of $z \leftarrow \max(x, y)$.
- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.compare($[\![x]\!], [\![y]\!]$), which takes either numeric or alphabetic secret shared values x, y to obtain fresh secret shared boolean z s.t. $z \leftarrow$ True iff $x < y$, while $z \leftarrow$ False when $x \geq y$. Note that when comparing two alphabetic values (e.g., names), $z \leftarrow$ True iff x is lexicographically smaller than y.
- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.equality($[\![x]\!], [\![y]\!]$), which takes either numeric or alphabetic secret shared values x, y to obtain fresh secret shared boolean z s.t. $z \leftarrow$ True iff $x = y$.
- $[\![z]\!] \leftarrow \mathcal{F}_{ABB}$.bit-decomposition($[\![x]\!]$), which takes a secret sharing of a field element $[\![x]\!]$ and converts it into secret-sharing of individual bits of the bit decomposition of x.
- $[\![z]\!] \leftarrow$ SISO-PRF($[\![s]\!], [\![x]\!]$). Shared-Input, Shared-Output Pseudo-Random Function[6] (SISO-PRF) takes secret shared key $[\![s]\!]$, a secret shared input $[\![x]\!]$, and computes a fresh secret shared output $[\![z]\!]$, where $z \leftarrow \text{PRF}_s(x)$.

Since we use ABB and show that our algorithms are oblivious based on ABB algorithms, the stand-along composition theorem of [47] applies.

3.2 Building Blocks

Silent PrefixSum We will repeatedly use (as a building block) the SILENT PREFIXSUM algorithm. Given a secret-sharing of an array of n field elements in $GF(q)$, $[x_1, \ldots, x_n]$ we compute

$$[[\![y_1]\!], \ldots [\![y_1]\!]] \leftarrow \text{SILENT PREFIXSUM}([[\![x_1]\!], \ldots, [\![x_n]\!]])$$

where each $[\![y_i]\!]$ is a fresh sharing of $\sum_1^i x_i$. Specifically, if $x_1, \ldots, x_q$ are secret-shared, and the underlying secret-sharing scheme supports non-interactive addition of the secret-shared values, the servers can compute

[6] We remark that SISO-PRF is the most expensive ABB function, and should be used sparingly. We are not using it inside secure graph conversion at all. For Dijkstra, we use it once for every d edges, where d is twice the average degree.

the secret-sharing of an array $y_1, \ldots, y_q$ *silently* – without any interaction with each other. We note that we do not need the full power of [18] for SILENT PREFIXSUM, since every server can locally compute a fresh share of each y_i.

Compositional Secret-sharing of Permutations We define a permutation $\pi \in S_n$ on n elements to be *secret-shared* among $c \geq 2$ servers $P_1, P_2, \ldots, P_c$ if every server holds permutation π_i on n elements, such that $\pi = \pi_1 \circ \pi_2 \circ \cdots \circ \pi_c$, where $\circ$ is *composition* operator. By $[\![\pi]\!]$ we denote such a secret-sharing. A uniform random permutation can be easily sampled *silently:*

$$[\![\pi]\!] \leftarrow \text{SAMPLE-FRESH-PERMUTATION}$$

by all c servers, with each server P_i just picking a fresh uniform random permutation π_i. Looking ahead, [28, 30, 41, 40, 39, 19] show how to not only store such compositional permutations and apply them with linear work and a constant number of rounds on any n-size secret-shared array but also be able to evaluate π^{-1} on secret-shared arrays of size n by applying $\pi_c^{-1} \circ \pi_{(c-1)}^{-1} \circ, \ldots, \circ \pi_1^{-1}$.

Secure Shuffle Given a secret-shared array $[\![A]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$ where each item has d bits, and a secret-sharing of a permutation $[\![\pi]\!]$ securely compute a secret-shared array $[\![\hat{A}]\!]$ of n elements:

$$[\![\hat{A}]\!] \leftarrow \text{SECURE-SHUFFLE}([\![A]\!], [\![\pi]\!])$$

where $\forall i, 1 \leq i \leq n$, $[\![\hat{A}_{\pi(i)}]\!]$ is a fresh copy of $[\![A_i]\!]$. Given $[\![\pi]\!]$, these protocols also support securely computing $[\![\pi^{-1}]\!]$ *silently* (i.e. without any communication). Two-party secure shuffle with linear work and constant rounds is known based on additively homomorphic encryption [30, 40, 39, 19], while three-party (or a constant-party) secure shuffle can be constructed unconditionally [28].

Oblivious k-Compaction: Given a public parameter $k < n$, and two secret-shared arrays, a boolean array of n bits $[\![B]\!] = [[\![b_0]\!], \ldots, [\![b_{(n-1)}]\!]]$ where exactly k bits are 1, and an array of length n: $[\![A]\!] = [[\![a_0]\!], \ldots, [\![a_{(n-1)}]\!]]$ where each a_i is d bits, we want to compute a fresh secret-sharing of array $[\![C]\!] = [[\![c_0]\!], \ldots, [\![c_{(k-1)}]\!]]$ which extracts k elements from $[\![A]\!]$ for which $b_i = 1$, not necessarily in the same order:

$$[\![C]\!] \leftarrow \text{OBLIVIOUS } k\text{-COMPACT}(k, [\![B]\!], [\![A]\!])$$

That is, $\forall i$ for which $b_i = 1$, $\exists! j$ such that $C[j]$ is a fresh copy of $A[i]$. Let array $[\![BA]\!]$ denote pairing of entries of $[\![A]\!]$ with entries of $[\![B]\!]$, where i'th entry of $[\![BA]\!]$ is $[\![(b_i, a_i)]\!]$. Oblivious k-compaction is done as follows:

1. $[\![\pi]\!] \leftarrow$ SAMPLE-FRESH-PERMUTATION
2. $[\![D]\!] = [([\![b'_0]\!], [\![a'_0]\!]), \ldots, ([\![b'_{(n-1)}]\!], [\![a'_{(n-1)}]\!])] \leftarrow$ SECURE-SHUFFLE$([\![BA]\!], [\![\pi]\!])$
3. Open all b'_i in $[\![D]\!]$ and compute $[\![C]\!]$ as subset of all a'_i s.t. $b'_i = 1$.

Oblivious Use-Once-KVS In our secure implementation of Dijkstra, we rely on a new primitive, called USE-ONCE KEY-VALUE STORE (ONCE-KVS for short), an oblivious data structure to retrieve (once for each key) a large payload indexed by short distinct keys. Specifically, we define a special-purpose Key-Value store that is more efficient than [46, 16] at the expense of a restriction on how it can be used.

Specifically, we are given a list of distinct keys, each associated with a large payload. We build a new data structure where the build time takes only a constant number of rounds and is linear time and linear communication. However, retrieval time is faster than prior works: it is just a single call to SISO-PRF, independent of the size of the payload, where we return a pointer to a fresh secret-sharing of a payload that does not reveal which key that payload is associated with. To reiterate, the restriction is that every key lookup can be used only once. We proceed to describe this primitive and its implementation:

We are given two arrays of the same length: a secret-shared array of keys $[\![K]\!] = [[\![k_0]\!], \ldots, [\![k_{(n-1)}]\!]]$ where each k_i is a secret-sharing of unique (typically short) key k_i, and an array of secret-shared payloads $[\![P]\!] = [[\![\vec{p_0}]\!], \ldots, [\![\vec{p_{n-1}}]\!]]$ of length q bits each[7].

USE-ONCE KEY-VALUE STORE supports two operations:

- KVS.INITIALIZE$([\![k_1]\!][\![\vec{p_1}]\!]), ([\![k_2]\!][\![\vec{p_2}]\!]), \ldots ([\![k_n]\!][\![\vec{p_n}]\!]))$.
- $([\![\vec{p_i}]\!]) \leftarrow$ KVS.READ$([\![k_i]\!])$ This operation outputs the secret shared payload of q bits. As mentioned, an important restriction is that each x_i can only be retrieved once.

KVS.INITIALIZE:

1. Generate a secret-shared seed $[\![s]\!]$ for SISO-PRF
2. For $1 \leq i \leq n$, compute $[\![tag_i]\!] \leftarrow$ SISO-PRF$([\![s]\!], [\![k_i]\!])$
3. $[\![\pi]\!] \leftarrow$ SAMPLE-FRESH-PERMUTATION
4. $[\![A]\!] \leftarrow$ SECURE-SHUFFLE$[([\![tag_1]\!][\![\vec{p_1}]\!]), \ldots, ([\![tag_n]\!][\![\vec{p_n}]\!])], [\![\pi]\!])$
5. $[\![B]\!] \leftarrow$ reconstruct all tags in $[\![A]\!]$ and build a lookup table indexed by tags.
6. Note that $[\![B]\!]$ are n pairs of the form $(tag_i, [\![\vec{p_i}]\!])$ where tag_i is in the clear and can be used to look up and retrieve secret-shared $[\![\vec{p_i}]\!]$.

KVS.READ$([\![k_i]\!])$

1. $[\![tag_i]\!] \leftarrow$ SISO-PRF$([\![s]\!], [\![k_i]\!])$
2. $tag_i \leftarrow$ open$[\![tag_i]\!]$
3. $[\![\vec{p_i}]\!] \leftarrow$ Lookup tag_i in $[\![B]\!]$ and retrieve payload that matches $[\![tag_i]\!]$

Theorem 5. (Oblivious Use-Once KVS) *Assume the existence of honest-but-curious* SECURE SHUFFLE *protocol with linear work and $O(1)$ rounds for c servers, resilient against any collusion of at most $u < c$ servers. Further, assume that there exists a c-server MPC protocol for ABB operations, including SISO-PRF, tolerating at most $u < c$ colluding servers. We are given two arrays of the same length: a secret-shared array of keys $[\![K]\!] = [[\![k_0]\!], \ldots, [\![k_{(n-1)}]\!]]$ where each k_i is a secret-sharing of unique (typically short) key k_i, and an array of secret-shared payloads $[\![P]\!] = [[\![\vec{p_0}]\!], \ldots, [\![\vec{p_{n-1}}]\!]]$ of length q bits each. There exists an* USE-ONCE OBLIVIOUS KEY-VALUE STORE *(Use-Once OKVS) with the following properties:*

[7]We use this primitive for secure Dijkstra, where each payload will, in fact, be an array of d edges. Therefore, the payload will not be individual bits but rather an array of d tuples, where each tuple is a secret sharing of an edge (or a $\perp$ of the same bit-length as edge representation) together with its weight as well as other information. However, this does not change the semantics of a large payload $\vec{p_i}$ in USE-ONCE KEY-VALUE STORE .

1. *Initializations takes $O(n)$ secure operations. Specifically, it takes n calls to SISO-PRF, and one secure shuffle of an array of size n elements each of size $q + k$, where k is the security parameter, which is also the length of the SISO-PRF output.*

2. *Each read takes $O(1)$ operations. Specifically, one execution of SISO-PRF.*

Oblivious Sort We are given two arrays of the same length: a secret-shared array of keys $[\![K]\!] = [[\![k_0]\!], \ldots, [\![k_{(n-1)}]\!]]$ where each k_i is a secret-sharing of a key k_i, (not necessarily unique) and an array of secret-shared values $[\![W]\!] = [[\![w_0]\!], \ldots, [\![w_{(n-1)}]\!]]$. We restrict individual keys k_i and individual values w_i to be bounded by a fixed $O(1)$ words of RAM memory in length. OBLIVIOUS SORT should also be provided with secure comparison function IS-LESS-THEN that outputs a boolean value β:

$$[\![\beta]\!] \leftarrow \text{IS-LESS-THEN}(([\![k_i]\!], [\![w_i]\!])([\![k_j]\!], [\![w_j]\!]))$$

which returns a secret-sharing of a boolean $\beta = 1$ only if $(k_i, w_i) < (k_j, w_j)$ according to predefined implicit ordering, which we will define every time we invoke OBLIVIOUS SORT. The running time is $O(n \log n)$ secure comparison operations and $O(\log n)$ rounds. The communication complexity is the communication complexity of a single secure comparison times $O(n \log n)$. We stress here that since we already assume SECURE SHUFFLE, the sorting can be done efficiently and does not need AKS sorting network. Jumping ahead, we will be sorting lists of size $O(V + E)$ where each item is a constant number of words of memory and, therefore, will take $O((V + E) \log V)$ secure operations and $O(\log V)$ rounds.

Running Dijkstra: privacy-preserving data-structures

In order to support Dijkstra, we have to have several data structures, one that we already discussed: USE-ONCE KEY-VALUE STORE table, where for each (replicated) vertex, we can retrieve its normalized adjacency list of size exactly d edges. We stress that the main efficiency of our protocol comes from the fact that edges (broken up into chunks of size d) are not inside DORAM but rather are stored inside USE-ONCE KEY-VALUE STORE as payloads $\vec{p_i}$ and hence do not incur multiplicative overhead. That is, USE-ONCE KEY-VALUE STORE stores, for each vertex, its adjacency list, broken up into chunks of size d. We can perform Dijkstra's "relax" operation for the adjacency list of d edges *in parallel* since all d elements in the adjacency list are either distinct or $\perp$.

We also create a DORAM that maintains a distance vector with n entries, and we separately create a secure priority queue that supports parallel decrease-key operations. The issue of the priority queue requires a separate discussion:

Oblivious Priority Queue with Parallel Decrease Key Dijkstra requires a min-heap Priority Queue. Specifically, it requires three operations from Priority Queue: Build Min-Heap with V items; execute Decrease-Key E times, and Extract-Min V times. All of these operations must be done securely. We use (a modification of) privacy-preserving Oblivious Heap of Jafargholi at. al., [58, 59]. We have to overcome several challenges:

- [59] proves their result in the ORAM client-server model instead of the MPC model with two or more servers and no client. Further, the client in [58, 59] has $O(\log n)$-words local memory and, therefore, can download and sort "for free" lists of size $O(\log n)$. We need to adopt this for the MPC model without a client, where we cannot sort "for free" any non-constant size lists.

- [59] data structure supports oblivious extract-min and decrease-key with $O(\log n)$ overhead. However, to minimize rounds, we will need to perform d decrease-key operations *in parallel*. We show that for our d-NORMALIZED REPLICATED ADEJENCY LIST, we only need Exclusive Read Exclusive Write (EWER) parallel operations since we are going to be processing (e.g., relaxing) a single (d-normalized) adjacency list in parallel. As shown in Section 6, we achieve amortized $O(\log n)$ work per operation and amortized $O(\log n \log\log\log n)$ rounds per all d such operations jointly.

In the MPC setting, our amortized work will remain $O(\log n)$ for all operations, where n is the total number of items. In Section 6, we show how to execute w decrease-key operations in parallel with $O(w \log n)$ work and $O(\log\log\log n)$ round complexity with the following functionalities:

- PQ.INITIALIZE($(\llbracket k_1 \rrbracket, \llbracket p_1 \rrbracket), \ldots, (\llbracket k_w \rrbracket, \llbracket p_w \rrbracket)$).
  Initialization takes a secret shared list of items $(k_1, p_1) \ldots (k_w, p_w)$, of keys and their priorities and executes

- PQ.INSERT($\llbracket k_i \rrbracket, \llbracket p_i \rrbracket$)
  for all items in the list, one at a time. For a list of size w, this takes $O(w \log n)$ work and $O(w \cdot \log\log\log n)$ rounds.

- PQ.INSERT($\llbracket k \rrbracket, \llbracket p \rrbracket$).
  Inserts item (k, p) into the Priority queue, at the root. Since we must maintain the order at the root, this takes $O(\log n)$ work and $O(1)$ rounds.

- PQ.PARALLEL-DECREASE-KEY($(\llbracket k_1 \rrbracket, \llbracket p_1 \rrbracket), \ldots, (\llbracket k_w \rrbracket, \llbracket p_w \rrbracket)$).
  This operation inserts a copy of items $(k_1, p_1) \ldots (k_w, p_w)$, with updated priorities. This takes $O(w \log n)$ work and $O(\log w \log\log\log n)$ rounds.

- $(\llbracket k \rrbracket, \llbracket p \rrbracket) \leftarrow$PQ.EXTRACT-MIN.
  This outputs secret-shared item (k, p) which has the lightest priority p within the PQ. This takes $O(\log n)$ work and $O(1)$ rounds.

We refer the reader to Section 6 for further details.

Shared Distance Vector Finally, as with Dijkstra, we have to maintain an array of all vertices and their distance from the source, where the source vertex is initialized with 0, and the rest are distance $\inf$. This array is kept in a DORAM structure of size V and updated it for each decrease-key/extract min sequence.

DORAM DORAM-SETUP$(k, d, n, \llbracket B \rrbracket)$: Given public parameters k, d, and n, and a secret-shared array $\llbracket B \rrbracket = [\llbracket (a_0, \alpha_0) \rrbracket, \ldots, \llbracket (a_{(n-1)}, \alpha_{(n-1)}) \rrbracket]$ where each a_i is k bits long and each α_i is d bits long, define a (virtual, exponential-size) array A containing 2^k (virtual) secret-shared elements of size d each. All locations a_i are initialized to contain $\llbracket \alpha_i \rrbracket$ whereas all other locations are $\perp$, supporting the following *Access*$(\llbracket op \rrbracket, \llbracket i \rrbracket, \llbracket y \rrbracket)$:

1. If $(op = read)$ return a fresh $\llbracket A_i \rrbracket$;
2. If $(op = write)$ set $A_i = y$.

We will also use in several places a DORAM distributed data structure [41]. It maintains a virtual memory that can hold up to n words stored in arbitrary locations, addressable by word-size arbitrary "address" value. DORAM allows read/write access into this virtual memory with $O(\log n)$ secure operations for each individual read/write.

Parallel DORAM

DORAM has been extensively analyzed in a *parallel* setting, where multiple reads/writes can be executed with $O(\log n)$ computational overhead per each read/write and can be done in parallel using $O(\log n)$ rounds [64, 11]. The work of [11] considered the original ORAM setting of [50] in the client-server model, where oblivious shuffles require *tight compaction* [72, 10] and low depth circuit for the building of a cuckoo hash-table. However, the best results of tight compaction [37] still require an enormous hidden constant in the big-O notation. Of course, this gives $O(\log n)$ overhead asymptotic ORAM, but we can also avoid enormous constants in the MPC setting [41]. Specifically, building the cuckoo hash table is not known with logarithmic depth and requires substantial additional work [11]. In contrast, cuckoo hash tables can be constructed by one of the servers in the clear (using tags) in the MPC setting [62, 41, 56, 69], and tight compaction is not needed if we have access to secure shuffle. We will use:

- PARALLEL-DORAM.INITIALIZE$((\llbracket \text{addr}_1 \rrbracket, \llbracket x_1 \rrbracket), \ldots, (\llbracket \text{addr}_n \rrbracket, \llbracket x_n \rrbracket))$. Initialize takes as input a list of secret shared address-value pairs $(\text{addr}_1, x_1) ... (\text{addr}_n, x_n)$, and constructs a virtual address that holds this n values at specified addresses This takes $O(n \log n)$ secure operations and $O(n \log n)$ words of memory per server and $O(n \log n)$ rounds. We stress that addresses a need not be consecutive and can be any arbitrary virtual address that fits into a single word.

- $\llbracket z_1 \rrbracket \ldots \llbracket z_w \rrbracket \leftarrow$ PARALLEL-DORAM.DISJOINT.READ$(\llbracket a_1 \rrbracket, \ldots, \llbracket a_w \rrbracket)$ reads value z_i at virtual address a_i for $1 \leq i \leq w$, where $w \leq n$, and if all a_i are distinct, it returns in a secret-shared form values $\llbracket z_1 \rrbracket, \cdots, \llbracket z_w \rrbracket$. If a_i are not all disjoint, obliviousness is not guaranteed. If some a_i was not previously written to, $\llbracket z_i \rrbracket$ is defined and returned as $\perp$. Reading takes $O(w \log n)$ secure operations and $O(\log n)$ rounds and can be executed repeatedly without leaking any partial information.

- PARALLEL-DORAM.DISJOINT.WRITE$((\llbracket a_1 \rrbracket, \llbracket y_1 \rrbracket), \ldots (\llbracket a_w \rrbracket, \llbracket y_w \rrbracket))$ writes secret shared value y_i to secret shared virtual address a_i, under the assumption that all a_i are distinct for all $1 \leq i \leq w$. Again, we stress that a_i can be an arbitrary address (that fits into a word of memory) within the PARALLEL DORAM virtual memory. If address a_i already holds some value, it is overwritten by y_i. Again, this takes $O(w \log n)$ secure operations and $O(\log n)$ rounds.

4 Graph Conversions for MPC

We start with the following lemma that does not require any assumptions:

Lemma 6. *Let G be any graph with n vertices and m edges, and let $d \leftarrow 2\lceil \frac{m}{n} \rceil$. Then, there exists d-normalized replicated adjacency list representation of G with exactly $2n$ replicated vertices.*

Proof. We start with the (standard) secret-shared adjacency list of $G(V, E)$, and modify it as follows:

1. For all vertices v with degree $\delta(v) \leq d$ pad the adjacency list to be of length d by adding dummy elements ($\perp$). Note that there could be at most n vertices. If there are fewer vertices, add $\perp$ vertices until there are exactly n adjacency lists all of length d. There are now n vertices (real and $\perp$), each with an adjacency list of length d.

2. There could be at most $n/2$ vertices with degree $\delta(x) > d$ since $\mathbb{E}\big[(\delta(n)\big] = \frac{m}{n}$ and by Markov inequality, since $d = 2\lceil \frac{m}{n} \rceil$:

$$\mathbb{P}\Big[\delta(n) \geq d\Big] \leq \frac{\mathbb{E}[\delta(n)]}{d} \leq \frac{1}{2}$$

For each vertex of size $\delta(v) > d$, partition its adjacency list into multiple parts: full blocks that are divisible by d and a "remainder". We will handle chunks of length exactly d and the reminder separately:

- There could be at most $n/2$ copies of vertices with "remainder' edges $< d$. Treat this set of reminder edges as in Step (1) above, and make it into exactly $n/2$ vertices with d edges each – by padding with $\perp$'s.
- For $\alpha \in \mathbb{Z}^+$ "chunks" of adjacency lists of length exactly d, observe that there are at most m edges, and therefore, $\alpha \leq m/d \leq n/2$. If $\alpha < n/2$, pad the total number of such chunks to be exactly $n/2$ using condition 4 of the definition 1.

To summarize, we have exactly n (replicated) vertices from step 1 and two $n/2$ sets of vertices from step 2, proving the Lemma. □

We are given as input a secret-shared Adjacency List, abbreviated as $[\![AL]\!]$ of length $q = V + E$. We further assume that all vertices are named $1, \ldots, V$ and their adjacency list appear in order from 1 to V.

Further, we can ignore edge weights for our d-normalization algorithm, as these do not play a role in our algorithm. Looking ahead, if the edges have weights, we have to keep these weights together with edges and pad vertex representation so that the representation of edges with weights and vertices is of the same length and is indistinguishable. Thus, ignoring weights, $[\![AL]\!]$, consists of vertices and their adjacency lists. All of them are concatenated together so that there is no length difference between the representation of vertices and edges and no information about where one adjacency list (of some vertex ends) and another vertex's adjacency list begins. The only information available to the servers is the total number of vertices V and the total number of edges E edges in the graph $G(V, E)$ represented by $[\![AL]\!]$.

For example, consider a 3-node graph $G(V, E)$ where node 1 points to vertices 2 and 3; node 2 points to node 3; and node 3 has no edges. The standard adjacency list of G is

$$[(1, (2, 3)), (2, (3)), (3, ())]$$

In $[\![AL]\!]$ the adjacency list will be represented as secret-sharing of the following ordered list

$$[\![AL]\!] = (([\![1]\!], [\![1]\!]), ([\![0]\!], [\![2]\!]), ([\![0]\!], [\![3]\!]), ([\![1]\!], [\![2]\!]), ([\![0]\!], [\![3]\!]), ([\![1]\!], [\![3]\!]))$$

where the first coordinate of each pair is a boolean variable $[\![b_i]\!] = 1$ if it is a vertex, and $[\![b_i]\!] = 0$ if it is an edge. Formally, $[\![AL]\!]$ representation of G is a list of q ordered pairs $([\![b_i]\!], [\![a_i]\!])$ where b_i is a boolean variable and a_i is a vertex name $\in \{1, 2, \ldots, V\}$. Further, we assume that all vertex names are padded to have the same length. Thus, all secret-shared pairs of the form $(1, \star) \in AL$ are vertices, while all pairs of the form $(0, \star) \in AL$ are edges.

Since servers know the total number of vertices V and the total number of edges E, servers can calculate twice the average degree $d = \lceil 2E/V \rceil$. The goal of our first algorithm is to compute in a private way a $d-$normalized replicated adjacency list $[\![A]\!]$, a two-dimensional array of size $2V \times d$, where each row consists of d edges $([\![c_i]\!], [\![a_i]\!])$ where $c_i, a_i \in \{\{1, \ldots, 2V\} \cup \{\perp\}\}$ and if both are not $\perp$, we require that (c_i, a_i) is an edge in the graph $G(V, E)$, in other words, $(c_i, a_i) \in E$. We reqruie that on the same row of $[\![A]\!]$, all c_i's are equal to each other, and if there are multiple rows with the same c_i's all must be located in adjacent rows of A. We aim to obliviously compute $[\![A]\!]$ given $[\![AL]\!]$.

4.1 Secure d-*normalization* algorithm

In this section, we already assume that the graph $G(V, E)$ vertex names are integers from 1 to V, and the adjacency list has these vertices appearing in the sorted, increasing order. To see how to convert a graph where vertex names are arbitrary alphanumeric strings into this form, see section 4.3. We describe our d-*normalization* algorithm in a series of steps, each of which can be computed either silently or in a constant number of rounds. We start with an ordered list $[\![AL]\!]$ consisting of $q = V + E$ ordered tuples (b_i, a_i). We aim to output a two-dimensional array $[\![A[i, j]]\!]$ where $1 \leq i \leq 2V$ and $1 \leq j \leq d$. Each row of $[\![A]\!]$ will consist of d entries $([\![c_i]\!], [\![a_i]\!])$ where $a_i \in \{\{1, \ldots, 2V\} \cup \{\perp\}\}$ is the name of the vertex to which the edge points, and $[\![c_i]\!]$ is the name of the vertex from the which the edge points from. In the same row of $[\![A]\!]$, all c_i's are equal to each other. We begin by describing all entries that will be added to each tuple, The tuple will eventually consist of seven entries:

$$[\![(b_i, a_i, c_i, r_i, \mathit{addr}_i, \mathit{fe}_i, \mathit{le}_i)]\!]$$

with the following semantics, where a_i and b_i is the initial tuple in $[\![AL]\!]$:

- <u>*Position*</u> 1: $b_i \in \{0, 1\}$ where b_i is a boolean variable indicating if this tuple is a representation of a vertex, in which case $b_i = 1$, or an edge, in which case $b_i = 0$
- <u>*Position*</u> 2: $a_i \in \{1, \ldots, V\}$ where a_i's is a vertex name which is an integer from 1 to V. To reiterate, in this algorithm, it is assumed that in $[\![AL]\!]$, for all tuples where $b_i = 1$, a_i appears in strict increasing order from 1 to V.
- <u>*Position*</u> 4: $r_i \in \{1, \ldots, 2V\}$ where r_i is a row in $[\![A]\!]$ where tuple i will be written to. Jumping ahead, we will divide $[\![AL]\!]$ into $\lceil q/d \rceil$ equal-length *blocks*, padding the last block with $\perp$'s if needed. Each distinct adjacency list of $[\![AL]\!]$ within each block gets its own row, which is then padded as necessary with $\perp$'s to be of length exactly d.
- <u>*Position*</u> 5: $\mathit{addr}_i \in \{1, \ldots, d\}$. In each block, tuples are numbered 1 to d. Tuple i gets addr_i.
- <u>*Position*</u> 6: $\mathit{fe}_i \in \{0, 1\}$ where fe_i is a boolean variable indicating whether this tuple is the first edge of a distinct adjacency list in that block.
- <u>*Position*</u> 7: $\mathit{le}_i \in \{0, 1\}$ where le_i is a boolean variable indicating whether this tuple is the last edge of a distinct adjacency list of that block.

In the algorithm description below, we "add" additional entries from the above list to each tuple at a point in our exposition where additional entries are logically needed. Naturally, this is only to make our explanation easier. To code our algorithm, all entries must be allocated ahead of time. We also allocate additional arrays, such as *Continuation* and PRELIM-1 and PRELIM-2 (that are used in the Row-setup step.

Our presentation does not consider additional variables that each tuple should have, such as *weights* of the edges. For Dijkstra, the tuple will also have weights as an additional entry for each tuple. Naturally, all tuple should be indistinguishable from each other, so all tuples should have entries for weights, even if for vertices, there are always just padded $\perp$'s.

We now present our d-*normalization* algorithm:

Input: $[\![AL]\!] = [([\![a_1]\!], [\![b_1]\!]), \ldots, ([\![a_q]\!], [\![b_q]\!])]$ where all vertices are named from 1 to V and vertex names appear in sorted order from smallest to largest.

Output: $[\![A]\!]$, a two dimensional array of size $2V \times d$, where each row consists of d edges $([\![c_i]\!], [\![a_i]\!])$. $c_i, a_i \in \{\{1, \ldots, 2V\} \cup \{\perp\}\}$, where for pairs that do not contaom $\perp$, $(c_i, a_i) \in E$ In the same row of $[\![A]\!]$, all c_i's are equal to each other, and if the same c_i appear in more than one row, they all such rows are adjacent to each other. In additionally, we output a secret-shared bit-vector of size $2V$ called $[\![\textit{Continuation}]\!]$. Each position $i \in \{1, \ldots, 2V\}$ of $[\![\textit{Continuation}]\!]$ corresponds to row r_i in $[\![A]\!]$, such that if c_i in row r_i and $c_{(i+1)}$ in row r_{i+1} are equal, then $[\![\textit{Continuation}[i]]\!] = [\![1]\!]$. Otherwise, $[\![\textit{Continuation}[i]]\!] = [\![0]\!]$.

d-*Normalization* Algorithm

1. **Block partition**: Partition $[\![AL]\!]$ into consecutive blocks of d tuples each. Let $\gamma = \lceil q/d \rceil$ blocks, so γ is the number of blocks. Pad the last block with $([\![0]\!], [\![\perp]\!])$'s as needed so that each block is of the same length and consists of tuples of the same length. In order not to complicate the notation, we re-define $q \leftarrow \lceil q/d \rceil \cdot d$.

2. **Adding variable** c_i: Let $[[\![c_1]\!], \ldots, [\![c_q]\!]] \leftarrow$ SILENT PREFIXSUM$([[\![b_1]\!], \ldots, [\![b_q]\!]])$. Consider any edge entry $(b_i = 0, a_i, c_i)$. Observe that c_i is the name of the vertex to whose adjacency list this edge belongs. We add to each tuple in $[\![AL]\!]$ a third element c_i, so each tuple in $[\![AL]\!]$ consists of:

$$([\![b_i]\!], [\![a_i]\!], [\![c_i]\!])$$

3. **Block variable:** For each block j, $1 \leq j \leq \gamma$ define a boolean variable $[\![\textit{cont}_j]\!]$ to be equal to 1 if the first entry of block j is an edge and 0 if the first entry of the block j is a vertex. That is, we look at b_i in the first tuple of each block. Note that $\textit{cont}_1 = 0$ always.

4. **Row-setup:** Define secret-shared array [PRELIM-1] of length $q + \gamma$, where you insert $\textit{cont}_i$ in front of each block i:

   4.1 $[\![\text{PRELIM-1}]\!] = [\![0, b_1, b_2, \ldots, b_d, \textit{cont}_2, b_{(d+1)}, \ldots, b_{2d}, \textit{cont}_3, b_{2d+1} \ldots]\!]$

   4.2 $[\![\text{PRELIM-2}]\!] \leftarrow$ SILENT PREFIXSUM$([\![\text{PRELIM-1}]\!])$

   4.3 Now we compute a secret-shared vector

$$[\![\text{ROW}]\!] = [\![r_1, \ldots, r_q]\!]$$

   of length q by deleting γ positions in the PRELIM-2 vector that corresponded to the positions of $\textit{cont}_i$ variables that were inserted into [PRELIM-1]. We now extend each tuple in $[\![AL]\!]$ to be of the form:

$$[\![(b_i, a_i, c_i, r_i)]\!]$$

   where tuple number i gets the i'th element of [ROW].

5. **Numbering within block**: Recall that within each block there are exactly d consecutive tuples, we number them (silently) from 1 to d and add this *count* to each tuple. $[\![AL]\!]$ now becomes:

$$[\![(b_i, a_i, c_i, r_i, \textit{addr}_i)]\!]$$

6. **"First edge" (fe):** We define a boolean variable $[\![fe_i]\!]$ for each tuple.
Consider any two consecutive tuples within any block:

$$[\![(b_{(i-1)}, a_{(i-1)}, c_{(i-1)}, r_{(i-1)}addr_{(i-1)})]\!] \quad \text{and} \quad [\![(b_i, a_i, c_i, r_i, addr_i)]\!]$$

6.1 $[\![fe_i]\!] \leftarrow (([\![addr_i]\!] = 1) \wedge ([\![b_i]\!] = 0)) \vee (([\![b_{(i-1)}]\!] = 1) \wedge ([\![b_i]\!] = 0))$
In other words, if the element is either the first in the block and entry $i-1$ does not exist in that block, or the previous element is a vertex, and this element is an edge, then this element is the first edge. The tuple now becomes:

$$[\![(b_i, a_i, c_i, r_i, addr_i, fe_i)]\!]$$

7. **"Last edge" (le):** We define a boolean variable $[\![le_i]\!]$ for each tuple.
Consider any two consecutive tuples within any block:

$$[\![(b_i, a_i, c_i, r_i, addr_i)]\!] \quad \text{and} \quad [\![(b_{(i+1)}, a_{(i+1)}, c_{(i+1)}, r_{(i+1)}, addr_{(i+1)})]\!]$$

7.1 $[\![le_i]\!] \leftarrow (([\![addr_i]\!] = d) \wedge [\![b_i]\!] = 0)) \vee (([\![b_{(i+1)}]\!] = 1) \wedge ([\![b_i]\!] = 0))$
In other words, if the element is either the last in the block or the subsequent element is a vertex, AND this element is an edge, then this element is the last edge. The tuple now becomes:

$$[\![(b_i, a_i, c_i, r_i, addr_i, fe_i, le_i)]\!]$$

8. **Create fake fe and le:**

8.1 $[\![Fake]\!] \leftarrow (2V - [\![r_q]\!])$ ▷ counting number of empty rows we need to fill with "fakes".

8.2 Create arrays $[\![F1]\!]$ and $[\![F2]\!]$, each of size $2V$ tuples ▷ each tuple has 7 entries

8.3 For all $t \in \{1, \ldots, 2V\}$ in parallel do:
If $t < [\![Fake]\!]$: put $[\![(\perp, \perp, \perp, \perp, \perp, 0, 0)]\!]$ into $[\![F1]\!]$ at position t. ▷ to be ignored in 9.4
Else: $(t \geq [\![Fake]\!])$ put $[\![(\perp, \perp, \perp, t, 1, 1, 0)]\!]$ into $[\![F1]\!]$ at position t ▷ $|fe_i = 1| = 2V$

8.4 For all $t \in \{1, \ldots 2V\}$ in parallel do:
If $t < [\![Fake]\!]$: write $[\![(\perp, \perp, \perp, \perp, \perp, 0, 0)]\!]$ into $[\![F2]\!]$ at position t. ▷ to be ignored in 10.4
Else: $(t \geq [\![Fake]\!])$ Write $[\![(\perp, \perp, \perp, t, 2, 0, 1)]\!]$ into $[\![F2]\!]$ at position t. ▷ Creating fake rows

9. **Total-First:**

9.1 $[\![\pi]\!] \leftarrow$ SAMPLE-FRESH-PERMUTATION ▷ $\pi \in S_{q+2V}$

9.2 $[\![F3]\!] \leftarrow$ SECURE-SHUFFLE$([\![AL]\!] \cup [\![F1]\!]), [\![\pi]\!])$

9.3 $[\![L6]\!] \leftarrow$ the $6th$ component of each tuple in $[\![F3]\!]$ ▷ extracting fe_i boolean array

9.4 $[\![F4]\!] \leftarrow$ OBLIVIOUS $2V$-COMPACT$(2V, [\![L6]\!], [\![F3]\!])$ ▷ F4 is of size 2V

9.5 F-first$[x, y]$ is a $2V \times d$ array. Initialize it as follows:

9.5.1 For $1 \leq x \leq 2V$:

9.5.2 Let $[\![(b_i, a_i, c_i, r_i, addr_i, fe_i, le_i)]\!] \leftarrow F4[x]$

9.5.3 For $1 \leq y \leq d$: ▷ Fill row with $\perp$'s prior to first edge appearance

$$\text{F-first}[x,y] \leftarrow \begin{cases} \text{If} \quad y < addr_i & \text{then create tuple } [\![(\perp,\perp,c_i,r_i,y,\perp,\perp)]\!] \\ \text{Else } y \geq addr_i & \text{then create tuple } [\![(\perp,\perp,\perp,\perp,\perp,\perp,\perp)]\!] \end{cases}$$

10. **Total-Last:**

10.1 $[\![\pi]\!] \leftarrow$ SAMPLE-FRESH-PERMUTATION ▷ $\pi \in S_{q+2V}$

10.2 $[\![F5]\!] \leftarrow$ SECURE-SHUFFLE$([\![AL]\!] \cup [\![F2]\!]), [\![\pi]\!])$

10.3 $[\![L7]\!] \leftarrow$ the $7th$ component of each tuple in $[\![F5]\!]$ ▷ extracting le_i boolean array

10.4 $[\![F6]\!] \leftarrow$ OBLIVIOUS $2V$-COMPACT$(2V, [\![L7]\!], [\![F5]\!])$ ▷ F6 is of size 2V

10.5 F-last$[x,y]$ is a $2V \times d$ array. Initialize it as follows:

10.5.1 For $1 \leq x \leq 2V$ in parallel do:

10.5.2 Let $[\![(b_i,a_i,c_i,r_i,addr_i,fe_i,le_i)]\!] \leftarrow F6[x]$

10.5.3 For $1 \leq y \leq d$ in parallel do: ▷ Fill row with $\perp$'s after the last edge appearance

$$\text{F-last}[x,y] \leftarrow \begin{cases} \text{If} \quad y > addr_i & \text{then create tuple } [\![(\perp,\perp,c_i,r_i,y,\perp,\perp)]\!] \\ \text{Else } y \leq addr_i & \text{then create tuple } [\![(\perp,\perp,\perp,\perp,\perp,\perp,\perp)]\!] \end{cases}$$

11. **Preparing $[\![A]\!]$:**

11.1 $[\![\pi]\!] \leftarrow$ SAMPLE-FRESH-PERMUTATION ▷ $\pi \in S_{q+4V \cdot d}$

11.2 $[\![A1]\!] \leftarrow$ SECURE-SHUFFLE$([\![AL]\!] \cup [\![\text{F-first}]\!] \cup [\![\text{F-last}]\!]), [\![\pi]\!])$ ▷ Treat arrays as 1-dimensional

11.3 $[\![A2]\!]$ is a boolean vector of length $q + 4V \cdot d$ where,
For each tuple $1 \leq i \leq (q + 4Vd)$ in parallel do:

$$[\![(b_i,a_i,c_i,r_i,addr_i,fe_i,le_i)]\!] \leftarrow A1[i]$$

$$A2[i] \leftarrow \begin{cases} [\![0]\!] & \text{If } r_i = \perp \\ [\![1]\!] & \text{Else } r_i \neq \perp \end{cases}$$

Note that the number of 1's in $[\![A2]\!]$ is exactly $2V \cdot d$.

11.4 $[\![A3]\!] \leftarrow$ OBLIVIOUS $(2V \cdot d)$-COMPACT$(2V \cdot d, [\![A2]\!], [\![A1]\!])$.

11.5 For $1 \leq i \leq 2Vd$ in parallel do:

$$([\![b_i]\!], [\![a_i]\!], [\![c_i]\!], [\![r_i]\!], \ [\![addr_i]\!], [\![fe_i]\!], [\![le_i]\!])] \leftarrow A3[i]$$

Open the fourth and the fifth value of each tuple (i.e. $(r_i, addr_i)$), and put into $(r_i, addr_i)$ address of A the following secret-shared values:

$$A(r_i, addr_i) \leftarrow ([\![c_i]\!], [\![a_i]\!])$$

12 **Binary Continuation Marker**: Initialize a secret-shared bit vector of size $2V$ called $[\![Continuation]\!]$. Each position $i \in \{1, \ldots, 2V\}$ of $[\![Continuation]\!]$ will correspond to row r_i in $[\![A]\!]$.

12.1 For $1 \leq i \leq 2V - 1$ in parallel do:

12.2 $(\llbracket c_i \rrbracket, \llbracket a_i \rrbracket) \leftarrow A[i, 1]$

12.3 $(\llbracket c_{i+1} \rrbracket, \llbracket a_{i+1} \rrbracket) \leftarrow A[i+1, 1]$

12.4
$$\textit{Continuation}[i] \leftarrow \begin{cases} \llbracket 1 \rrbracket, & \text{if } (\llbracket c_i \rrbracket = \llbracket c_{i+1} \rrbracket) \\ \llbracket 0 \rrbracket, & \text{if } (\llbracket c_i \rrbracket \neq \llbracket c_{i+1} \rrbracket) \end{cases}$$

12.5 $\textit{Continuation}[2V] \leftarrow \llbracket 0 \rrbracket$

Looking ahead, these binary continuation markers will be used in our implementation of Dijkstra's algorithm to see if the adjacency list in row r_i continues in row r_{i+1}.

4.2 Analysis of d-*normalization* algorithm

At a high level, the goal of d-*normalization* is to obliviously place all items from our initial secret-shared adjacency list $\llbracket AL \rrbracket$ into a new secret-shared array $\llbracket A \rrbracket$ with $2n$ rows and d columns. The first entry of each row will contain vertex name $v_i \in \{1 \dots V\}$, and the remaining d rows will contain entries of edges of v_i or $\perp$s.[8] The goal of our algorithm is to obliviously compute for each item in $\llbracket AL \rrbracket$ its exact eventual address in $\llbracket A \rrbracket$. But notice that $|A| > |AL|$. Therefore, computing where each item of AL will eventually be placed into A is insufficient. We also need to generate a single $\perp$ with a unique address for each empty location of A once all items of AL have been placed into A. Once we (somehow) obliviously generate unique $\perp$s for each empty location, where for each location the algorithm guarantees to have exactly one item, we can mix together all items that are designed to A using secure shuffle. Once everything is shuffled, we can open all earmarked locations and place them into these locations in the clear. This is exactly what we do. We now proceed to analyze each step of the algorithm:

In step 1, we partition the adjacency list into blocks of length d, filling the last block with $\perp$'s in case q is not divisible by d. Again, this is done silently. We redefine $q = \lceil q/d \rceil \cdot d$. In step 2, we tell each edge which vertex it "belongs to". This takes $O(q)$ work and is done silently, i.e., it does not require any communication.

In steps 3 and 4, we mark the blocks where edges continue from the previous block. This takes $O(1)$ rounds, and the work is proportional to the number of blocks. Now, observe that in each tuple, we have indicator random variable r_i for all entries of $\llbracket AL \rrbracket$, which indicates which *row* of A each item should go into.

In step 5, we independently assign consecutive numbers to all tuples in the block from 1 to d for each block. This can be done silently. The idea here is that within each row of A, edges for some vertex will go into the same positions as they are in their block. This can be accomplished since $\llbracket AL \rrbracket$ has blocks of size d, and this is exactly the number of columns in $\llbracket A \rrbracket$. So if there is a vertex in the middle of the block with edges a, b, c in positions $12, 13, 14$ of that block, they will go into the same row of A in columns $12, 13, 14$ with columns 1 through 11 of that row filled with $\perp$ and columns 15 to d of that column also filled with $\perp$. Jumping ahead, obliviously filling with $\perp$s is done in steps 9 and 10. More technical, we just create entries with proper addressing before we mix everything and then reveal addresses. Observe that since we do this in parallel for all tuples, this can be done in $O(1)$ rounds and $O(q)$ work.

Steps 6 and 7 mark in $\llbracket AL \rrbracket$ all the first and the last edges of each adjacency list, broken up into blocks, so for each adjacency list in each block, there is the first and last edge. As stated before, when we divide $\llbracket AL \rrbracket$ into blocks, some adjacency list may start in the middle of a block. The edges of this adjacency list

[8] We remark that it is not necessarily the case that the first edges will appear in each row, followed by $\perp$s. Instead, in a typical situation, we will see several $\perp$s, followed by real edges and additional $\perp$s.

will go into some rows in exactly the same positions as they appear in this block. Therefore, we must pad this row before and after this adjacency list with $\perp$s. To accomplish this, we need to mark the first and the last entry of this adjacency list. This is exactly 6 and 7 do. Because we look at all triples of adjacent tuples in parallel, each triple can be decided in $O(1)$ rounds and constant work. This means that the total rounds are constant and the work is linear in q.

Recall that our array A has d columns and $2V$ rows. But out of these rows, how many are occupied at all, and how many are completely empty? Recall that in step 4.3, we computed a vector of length q called $[\![\text{ROW}]\!]$. The value in the last entry of $[\![\text{ROW}]\!]$ called $[\![r_q]\!]$ indicates how many rows of A we will use for contents of $[\![AL]\!]$. Since A has a total of $2V$ rows, the number of totally empty rows will be $[\![Fake]\!] = 2V - [\![r_q]\!]$. Step 8 looks at $[\![Fake]\!]$ and obliviously creates additional $[\![Fake]\!]$ first edges and $[\![Fake]\!]$ last edges to pad the number of both fe and le to $2V$. These are added to arrays $F1$ and $F2$, from which we will later (steps 9.4 and 10.4) extract exactly $2V$ of the non-ignore first edges. Step 8.4 assigns each fake first and last edge to an empty row into positions 1 and 2, respectively. This is so that the subroutine that computes bots following the last edge will add bots to fill these empty rows. Observe that this can be done in parallel for all created tuples, meaning this takes $O(1)$ rounds and $O(E)$ work.

Finally, in steps 9 and 10 we create arrays secret shared arrays F-first and F-last, each of size $2Vd$, which are filled with $\perp$s, some with valid addresses and some with $\perp$ as their address. Later, these 'ignore' elements with no valid address will be ignored in step 11.3 All of these steps together take $O(1)$ rounds and $O(q)$ work. In step 11, we first shuffle and then open all the addresses. We can then bring all the entries for each location of A to its proper position in a clear. This takes $O(q)$ work and $O(1)$ rounds, as sorting by revealed addresses can be done without interaction.

In step 12 creates the Binary Continuation Marker, a boolean variable for each row that tells whether the same vertex adjacency list continues to the next row. This is done in parallel for all rows and takes $O(V)$ work and $O(1)$ rounds.

Observe that in total, our algorithm takes $O(1)$ rounds and $O(V + E)$ work.

4.3 Oblivious Graph Renaming Algorithm

Given a secret-shared adjacency list $[\![AL]\!]$ with arbitrary alphanumeric names of vertices of equal length, we show how to obliviously convert all vertex labels to integers from 1 to V that appear in sorted order.

For any graph $G(V, E)$, servers are given secret-shared adjacency list $[\![AL]\!]$ of length $q = V + E$. $[\![AL]\!]$ consists of q tuples $([\![b_i]\!], [\![a_i]\!])$. Here, b_i is a boolean variable that indicates if a tuple is a vertex or an edge, where $[\![b_i = 1]\!]$ denotes the vertex and $[\![b_i = 0]\!]$ is an edge; and a_i is the alphanumeric vertex label.

For example, consider a graph where vertex Bob points to vertices Alice and Eve, Eve points to Alice, and Bob points to nobody. The standard adjacency list representation would be:

[(Bob, (Alice, Eve), (Eve, (Alice)), (Bob, ())].

The $[\![AL]\!]$ would be 6 *tuples*:

$$[([\![1]\!], [\![\text{Bob}]\!]), ([\![0]\!], [\![\text{Alice}]\!]), ([\![0]\!], [\![\text{Eve}]\!]), ([\![1]\!], [\![\text{Eve}]\!]), ([\![0]\!], [\![\text{Alice}]\!]), ([\![1]\!], [\![\text{Bob}]\!])].$$

where each tuple is exactly of the same length, and alphanumeric names are padded to be exactly word-size. We now describe our

We begin by describing all entries that will be added to each tuple during our algorithm execution. The tuple will eventually consist of five entries:

$$[\![A]\!] = ([\![b_i]\!], [\![a_i]\!], [\![c_i]\!], addr_i, [\![z_i]\!])$$

- *Position* 1: $b_i \in \{0, 1\}$ where b_i is a boolean variable indicating if this tuple is a representation of a vertex, in which case $b_i = 1$, or an edge, in which case $b_i = 0$.
- *Position* 2: a_i, an alphanumeric name that fits into the word size. All names should ahve the same length, padded as needed.
- *Position* 3: $c_i \in \{1, \ldots, V\}$ where, if $b_i = 1$ then c_i is the integer renaming of this vertex, and if $b_i = 0$ then c_i is the integer vertex name to which this edge belongs to.
- *Position* 4: $addr_i \in \{1, \ldots, q\}$ where $addr_i$ is the index of the ith tuple in $[\![AL]\!]$ in its original order. After oblivious shuffling, this address will later be revealed to sort the adjacency list into its original order.
- *Position* 5: $z_i \in \{1, \ldots, V\}$ where for $b_i = 1$, $z_i = c_i$ and for $b_i = 0$, z_i represents the vertex to which the edge points to. In other words, for all tuples z_i is the new integer vertex name which faithfully keeps the graph topology.

We are now ready to describe our graph renaming algorithm. We first start with I/O specification:

Input: $[\![AL]\!] = ([\![b_1]\!], [\![a_1]\!]), \ldots, ([\![b_q]\!], [\![a_q]\!])$

Output: $[\![A]\!] = ([\![b_i]\!], [\![a_i]\!], [\![z_i]\!])$. Here, each z_i is the new integer vertex name $\in \{1, \ldots, V\}$, and a_i is the alphanumeric name that is kept. We additionally guarantee that all vertices appear in the strict increasing order from 1 to V.

Oblivious Graph Renaming Algorithm:

1. **Numbering Vertices and Edge Attribution**:

   $$[[\![c_1]\!], \ldots, [\![c_q]\!]] \leftarrow \text{SILENT PREFIXSUM}([[\![b_1]\!], \ldots, [\![b_q]\!]])$$

   We modify $[\![AL]\!]$ to add c_i to each corresponding tuple. $[\![AL]\!]$ now consists of q tuples of the form (b_i, a_i, c_i). If $b_i = 1$, the tuple (b_i, a_i, c_i) is a vertex. Observe that SILENT PREFIXSUM computes c_i in the tuple $(1, a_i, c_i)$ to be *vertex numbering* of corresponding vertices in $[\![AL]\!]$ in increasing order from 1 to V. If $b_i = 0$, this is a tuple corresponding to an edge, where $c_i \in \{1, \ldots, V\}$. Observe that c_i of this edge indicates which vertex c_i in the *vertex numbering* this edge belongs to.

2. **Adding address i to each tuple:** Since $[\![AL]\!]$ is an ordered list of q tuples, we can assign to each tuple its position in the list, from 1 to q, in increasing order. We call this position $addr_i$ of the tuple number $1 \leq i \leq q$, so tuple number i becomes $([\![b_i]\!], [\![a_i]\!], [\![c_i]\!], [\![addr_i]\!])$

3. **Oblivious Sort** We now invoke Oblivious Sort (see section 3.2):
   $[\![A]\!] \leftarrow \text{OBLIVIOUS SORT}[\![AL]\!]$ with the following comparison predicate:

   $$\text{IS-LESS-THEN}((b_i, a_i, c_i, addr_i)(b_j, a_j, c_j, addr_j)) =$$

   $$= \begin{cases} [\![1]\!], & \text{if } (a_i \prec a_j) \vee ((a_i = a_j) \wedge (b_i > b_j)) \\ [\![0]\!], & \text{otherwise.} \end{cases}$$

   where $a_i \prec a_j$ is TRUE if string a_i is alphabetically smaller then string a_j.

4. **Renaming Edges**.
We now wish to add a new variable z_i to each tuple $(b_i, a_i, c_i, addr_i) \in A$. Before we show how this can be efficiently accomplished, we define z_i:

$$[\![z_i]\!] = \begin{cases} [\![c_i]\!], & \text{if } (b_i = 1) \\ [\![c_j]\!] & \text{if } (b_i = 0) \wedge \exists j \text{ s.t. } (a_i = a_j) \wedge (b_j = 1) \end{cases}$$

Observe that A is sorted by a_i, where the vertices are a_i where $b = 1$. Further, observe that A has q tuples, out of each E are edges (i.e. $b_i = 0$) and V are vertices (i.e. $b_i = 1$). We first define a procedure:

EXTEND(i, j) where $1 \leq i < j \leq q$:

$$\forall k \in \{i, \ldots j\} \text{ if } (a_i = a_j) \wedge (z_i \neq \perp) \text{ then } z_k \leftarrow z_i \text{ else } z_k \leftarrow z_k$$

We stress that all (i, j) pairs are public and are fixed in advance. In corner cases, we further specify that if EXTEND(i, j) is called with $i < q$ but $j > q$, the call automatically becomes EXTEND(i, q). We are now ready to define:

PARALLEL NAME EXTENSION Subroutine:

4.1 For all $1 \leq i \leq q$ in parallel do: if $b_i = 1$ then $z_i \leftarrow c_i$, else $z_i \leftarrow \perp$
4.2 For s from 1 to $\lceil \log q \rceil$ do:
  4.2.1 For all $1 \leq h \leq \lceil \frac{q}{2^s} \rceil$ in parallel do:
    4.2.2 EXTEND(i, j) for $i = 2^s(h-1)+1,\ j = 2^s h$
    4.2.3 EXTEND(i, j) for $i = 2^s(h-1)+2^{s-1}+1,\ j = 2^s h + 2^{s-1}$
4.3 For s from $\lceil \log q \rceil$ down to 1 do:
  4.3.1. For all $1 \leq h \leq \lceil \frac{q}{2^s} \rceil$ in parallel do:
    4.3.2 EXTEND(i, j) for $i = 2^s(h-1)+1,\ j = 2^s h$
    4.3.3 EXTEND(i, j) for $i = 2^s(h-1)+2^{s-1}+1,\ j = 2^s h + 2^{s-1}$
4.4 For all $1 \leq h \leq \lceil \frac{q}{2} \rceil$ in parallel do:
  4.4.1 EXTEND$(2(h-1)+1,\ 2h)$
  4.4.2 EXTEND$(2(h-1)+2,\ 2h+1)$

5. **Secure Shuffle** We now have $[\![A]\!]$ consisting of q tuples of the form $([\![b_i]\!], [\![a_i]\!], [\![c_i]\!], [\![addr_i]\!], [\![z_i]\!])$ where $\forall i, z_i \neq \perp$. We can now shuffle $[\![A]\!]$:

   1. $[\![\pi]\!] \leftarrow$ SAMPLE-FRESH-PERMUTATION $\triangleright\ \pi \in S_q$
   2. $[\![B]\!] \leftarrow$ SECURE-SHUFFLE$([\![A]\!]), [\![\pi]\!])$

6. **Final Cleanup**

   1. $\forall i, 1 \leq i \leq q$, open $addr_i \in ([\![b_i]\!], [\![a_i]\!], [\![c_i]\!], [\![addr_i]\!], [\![z_i]\!])$
   2. $[\![C]\!] \leftarrow$ sort in the clear all q tuples of $[\![B]\!]$ by (opened) $addr_i$.

4.4 Analysis of Graph Renaming Algorithm

Steps 1 and 2 can be done silently or in parallel for each tuple, taking $O(1)$ rounds and $O(q)$ work. Step 3 sorts all tuples in $[\![AL]\!]$ by alphanumeric names with priority to vertices. The result of this step is all tuples

with the same alphanumeric name will be adjacent to each other, and the vertex with this name will precede the edges with this name. This takes $O(\log V)$ rounds to sort and $O(q \log V)$ work.

After Step 4 is complete, we need to show that all tuples in $[\![A]\!]$ have $z_i \neq \perp$. To prove this, we abstract the problem as follows: consider an oriented line graph with q nodes, going left to right, so there is a distinct leftmost and distinct rightmost node on the line. We number the nodes from 1 to q, where 1 is the leftmost node. Exactly n of the nodes are colored with distinct colors, $n < q$, where the leftmost node is always colored and $q - n$ are initially uncolored. There is no other restriction where colored nodes appear on the line. In addition, every node on the line has an alphanumeric name, not necessarily distinct, but where all colored nodes do have distinct names. The uncolored nodes have the same name as their closest node on the left which is colored. A local coloring rule is: if the node has some color c and its immediate right-hand neighbor is uncolored, color it with the same c. The question that we are trying to address is how to color the entire line obliviously and as efficiently as possible.

A naive algorithm would be to do a linear scan on the line going left to right and coloring all the nodes as you go. We make the following observation: consider any interval on the line from i to j. If a node in the position i is already colored, and node j has the same alphanumeric name as node i, it is safe to immediately color the entire interval from i to j with the color of node i. We call this process *extend*(i, j). Observe that we can apply multiple non-overlapping intervals at the same time. This is what we do: apply intervals of geometrically increasing length, where for each interval length, we apply it twice: for all consecutive intervals of that length, and all consecutive intervals shifted to the right by half its length (we call it the "brick" pattern), and doing a smaller interval at the end if the full interval does not fit.

We define a "gap" as the largest interval that can be colored at once, before any *extend* calls – the interval from i to j, where position i is a colored node and position $j + 1$ is the next colored node to the right of it. The gap does not change after its nodes begin getting colored. Our objective is to prove that every gap is colored in step 4. For any one gap, we define step s as having "failed" for this gap if the number of nodes colored in this gap after step $s - 1$ and after step s is the same number.

Lemma 7. *Consider any gap of length h in the interval from i to j. Initially, only node i is colored. After step 4.2, nodes from i to $i + \lceil \frac{h}{2} \rceil$ are colored.*

Proof. We will show that when step 4.2 fails for any gap, the left half of the gap is colored. Observe that after $s = 1$ is executed in step 4.2, $i + 1$ is colored. After $s = 2$ is executed for step 4.2, observe that either interval $(i, i + 4)$ or $(i, i + 5)$ is colored, depending on whether i was odd or even. Assume s non-failing steps. Note that intervals of step $s + 1$ start at every index $1 \mod 2^s$. If step $s + 1$ fails, both the interval starting at i and $i + 2^s$ have failed. That means that the distance between $i + 2^{s+1}$ and j is strictly less than 2^s. This means that after step s, more than half of the nodes in the gap are colored. □

Lemma 8. *Consider any gap of length h in the interval from i to j. After step 4.4, all nodes of the gap are colored.*

Proof. We already showed in lemma 7 that after step 4.2, nodes i to $i + \lceil \frac{h}{2} \rceil$ are colored. If in step 4.2 $s = k$ failed, then the number of uncolored nodes in this gap is less than 2^k. We know that any number can be summed with powers of 2 by writing this number in a binary representation. Additionally, observe that as we geometrically decrease from $s = \lceil \log q \rceil$ to $s = 1$, any interval that does not fail will cut the remaining uncolored nodes down by half. After completing $s = 1$, there are either 0 or 1 remaining uncolored nodes. Repeating $s = 1$ again at the end, all nodes are colored. □

Since we are doing powers of 2, step 4 takes a total of $O(q \log V)$ work and $O(\log V)$ rounds. Step 5 shuffles $[\![A]\!]$, and in step 6, we reveal the addresses and move all elements to the addresses (row, column) that

they previously secret-shared, completing the oblivious conversion to an adjacency list with vertex names. This conversion took, in total, $O((V + E) \log V)$ work and $O(\log V)$ rounds.

5 Oblivious Priority Queue with Parallel Decrease Key

We will now briefly describe the original [58, 59] Oblivious Priority Queue. The paper constructs a full binary tree with n items, where each node of the tree is a buffer that can fit $B = O(\log n)$ items, and where each item is a (key, priority)-pair, both assumed to fit into a single word of RAM.

Their construction combines hierarchical ORAM [50] and Path-ORAM [77]. Specifically, items are assigned to a random root-to-leaf path in the tree (and never re-assigned) and traverse this path during operations. The path is determined by the output of a $O(\log n)$-wise independent hash function. Note that this means the prefix of the hash output on key v from index 0 to j gives the address of v at level j. Therefore, each item "knows" which node it belongs to at any given level.

To avoid overflow or underflow of any buffer, Jafargholi et al. define two procedures:

1. PUSHDOWN(i): Consider contents of all vertices at level i. Each node sends all of its items to its two children at level $i + 1$. Each item moves according to its PRF evaluated on the item's key.

2. PULLUP(i): Consider contents of all vertices at level i. Each node of level i gets the lightest $B/2$ elements from both of its children at level $i + 1$.

The [59] updates the tree during Priority Queue operations as follows. It keeps a counter of the number of operations performed (e.g., decrease-key, extract-min). Just like hierarchical ORAM, every 2^i operations, the tree executes PUSHDOWN(0),...,PUSHDOWN(i), PULLUP(i),...PULLUP(0) in that order. We call this operation *maintenance* procedure and in our algorithm, we will have the same maintenance procedure for the frequencies of PUSHDOWN and PULLUP depending on the counter, thought the specification of both PUSHDOWN and PULLUP will be different for our purposes. Note that if i surpasses the number of levels in the tree, the tree will simply push down to the maximum level. This is done by pushing down and pulling up from i^*, where $i^* = \min\{i, \text{largest level}\}$ as defined in [59].

Recall that [59] works in the client-server model, so the client can download any non-leaf vertex and its two children and sort contents of these lists locally "for free" to choose the lightest elements to take to the parent. However, as our algorithm operates in a three-server model and not a client-server model, we do not have "for free" sorting operations and have to "pay" for all operations. We observe that in the original work [59], sorting occurs by the client only during PULLUP operation. Specifically, the contents of the two children are sorted by the ORAM client in the clear (once downloaded into the client's memory), and the lightest items are sent to the parent of these two children. This is done for the entire level of the tree.

We now describe how to modify [58, 59] Oblivious Priority Queue to match our model and allow for parallel decrease-keys. Inductively, we assume that each node is already sorted from heaviest to lightest priority. When examining any node and its children, we can use linear-time secure merge algorithm [19] to merge the two sorted lists. Then, we can select the lightest B/2 elements in the sorted list to move to the parent. During PULLUP(i), we move selected $B/2$ lightest items from level $i + 1$ to level i in parallel. To keep the remaining "heavier" items that did not move up in the correct sorted order, observe that the items were sorted from heaviest to lightest. Only proper suffix of each list for both children could move up (without omissions). Furthermore, we can mark which items are moving up. Therefore, each item knows if it moved up or not, and can be (in parallel) replaced by $\perp$.

Let us examine the round and work-complexity of the modified PULLUP(i). By [19], for w items, merging the lists will take $O(w)$ work and $O(\log \log w)$ rounds. Moving the items will also take $O(w)$

work and $O(1)$ rounds since we are moving them all in parallel. This means the total is $O(w)$ work and $O(\log \log w)$ rounds. Given that the buffer size at any node is $O(\log n)$, PULLUP(i) for every node at level i takes linear (therefore $O(\log n)$) work and $O(\log \log \log n)$ rounds. Since we will do PULLUP(i) in parallel for all vertices of level i, PULLUP(i) will take $O(\log \log \log n)$ rounds and would take $O(2^i \cdot \log n)$ work. Observe that this is the same work complexity as in [59], and therefore we refer the reader to [59] for the proof of correctness for amortization analysis.

Now, let us consider how we can modify PUSHDOWN and keep the inductive hypothesis. To keep each node holding its items in sorted order from heaviest to lightest, we must PUSHDOWN in a similar way: when examining node v and its two children, we will use [19] to merge the three lists. Now we have the large sorted list and two empty children between which the list will be distributed, each item according to its $O(\log n)$-wise independent hash function. For each item, we compute the output of the hash function once during the initial build and once for every decrease-key operation and store the output together with the value. $O(\log n)$-wise independent hash function is represented as a random polynomial of degree $O(\log n)$ where all coefficients are secret-shared. Therefore, evaluating such a polynomial takes $O(\log n)$ secure operations, namely exponentiations, multiplications, and additions.

Specifically, once the function is evaluated, we store it in a secret-shared form, and it moves together with the item. For each item we extract the i'th bit of the stored hash function, which indicates if this value has to go left or right. This can be done in parallel for all items inside the bit-decomposition black-box operator. Now, mark all items that go to the left as secret-sharing of $[\![0]\!]$ and all items that go to the right as secret-sharing of $[\![1]\!]$. Now, we can take a Silent Prefix-Sum of this vector, and this will reveal to which address each right-bound item must be written to. The same process can be repeated for the left-bound items.

Then, obliviously shuffle all items and put all items in place in the correct place and order. All items will move in parallel, and we will therefore execute PUSHDOWN of the entire level in parallel, meaning that in total, a PUSHDOWN will take $O(\log \log \log n)$ rounds, and $O(2^i \log n)$ work for level i. Observe that this is the same work complexity as in [59], and therefore, we refer the reader to [59] for proof of the correctness of the amortization analysis.

So far, we have assumed inductively that each node is already sorted, and described how the tree remains sorted when items are moved between levels. Now, we will describe how all vertices will remain sorted even when we insert or remove items. Recall that [59] has four procedures that will either add or remove an item from the data structure: Insert, Decrease-Key, Extract-Min, and Delete. Note that the Insert procedure simply calls Decrease-Key.

- **Initialization of Priority Queue with n items:** As in [59], to initialize the Priority Queue we insert all elements at the root, one at a time. If we insert n items, we will also run n *maintenance* steps. We follow the maintenance schedule as in [59] (i.e., the scheduled PUSHDOWN and PULLUP procedures).

- **Insert n elements:** We insert one element at a time. Observe that the root node has space for at most $O(\log n)$ items; by induction, these items are already sorted. Therefore, we just have to insert the new item in the sorted order. While it is easy to do such a step, observe that this can be thought of as a secure merge of a single item and a sorted list, which we already know how to do by [19] with linear work in the node size. That establishes the base case of the induction. Of course, we run maintenance steps as well.

- **Parallel Decrease-Key of w items:** Before we perform parallel decrease-key of w items, we execute PUSHDOWN(0),... PUSHDOWN(γ), for $\gamma = \log(w)$. This means level γ and all levels above it are completely empty. We refer to this step as "PushDown-manual" maintenance.

After this is done, to Decrease-key w items, we insert w items at level γ, which has w vertices. Here, it is important that we make our buffer size B in each node twice as large as in [59]. Observe that the total number of vertices in a binary tree of depth γ has $2w - 1$ vertices. Furthermore, [59] Lemma 6 proves that none of the vertices will overflow after w sequential insertions. Therefore, in our construction, with w vertices but with each node's buffer twice as big, we have $2w$ buffer space in total with uniform distribution, and therefore Lemma 6 holds.

One unresolved matter is that when we insert items at level γ, the items that fall into the same node must be sorted. Now, for each item i, we can compute in parallel which node id $id(i)$ at level γ item i must go into by considering γ-prefix the item's key $O(\log n)$-wish hash function[9].

This gives us the address of that item at level γ. Now, for any item i, consider a pair of the address $id(i)$ and item's priority p_i, and obliviously reverse-sort all w items by concatenation of address and priority $id(i)|p(i)$, where "|" is the concatenation operator. This way, when we sort the items, all items belonging to the same node will be adjacent, and appear in order from heaviest to lightest priority.

However, writing all of these items to the correct vertices in parallel in $O(1)$ rounds takes additional work since we must prevent collisions of items that are going to the same node. This is accomplished as follows:

- After w items are sorted by $id(i)|p(i)$, we have (a secret shared) list of (that sorted order) w items. For all items i we assign consecutive numbers $\beta_w, \ldots, \beta_1$ to this sorted list from w down to 1. We also mark some items with a secret-shared predicate, which we call "lead." Lead items are defined as the heaviest items for each node. This is done by comparing each item (under MPC) to its predecessor in the sorted order and checking if the predecessor belongs to a different node. The first item in the sorted order is always the lead item.

- Recall that all vertices have buffer size $B = O(\log n)$ and are empty. Now, we obliviously put item i into buffer $id(i)$ at buffer location ($\beta_i \mod B$). Observe that since items that belong to the same $\mathrm{id}(i)$ are already sorted from heavy to light, they will have consecutive numbering $\beta_j, \beta_{j+1}, \ldots$. Since we are placing items into the buffer at the location $\beta_j \mod B$, all of them will be placed into the buffer in consecutive order but with some (circular) shift. This placement can be done by obliviously shuffling all locations of all vertices together, and then putting items in the clear into shuffled locations, and then "un-shuffling." Observe that all w items are placed into correct buffers without collisions, but they may be shifted inside each buffer. The final step is to adjust the shift in each buffer so that the heaviest item is placed in location 1 of each buffer.

- For each buffer, we compute the following value: for the "lead" item, we compute its current location in the buffer and secret-share this value. This defines the amount δ that the contents of the buffer must be shifted by. For all non-lead items, we secret-share $[\![0]\!]$. We now silently add (using Silent Prefix-Sum) all values for each node. This gives us a single secret-shared "shift-by" value. Now, in parallel for all vertices, we can compute their current location and shift-by location. We finally obviously shuffle all the items and put them in the correct locations.

[9] Computing a hash function takes $\mathrm{O}(\log n)$ secure operations to compute it only once and attach the output of the hash function it to the item, so that we don't have to compute it again. That hash function value will travel together with the item and will need not be recomputed.

Next, we execute PULLUP(γ),...PULLUP(0), which we call as PullUp-manual maintenance. We refer to the PushDown-manual maintenance and PullUp-manual maintenance jointly as manual maintenance. Finally, we perform w steps of "traditional" maintenance, by increasing the counter of operations by w and performing all the maintenance operations. Observe that PUSHDOWN(0), ... PUSHDOWN(γ) PULLUP(γ),..., PULLUP(0) is executed during w steps of maintenance, so the manual maintenance is $\frac{1}{2}$ of the maintenance routine. This means, in total, we did 1.5 times the work of maintenance. That is just a constant that goes out in big-O notation. Therefore, we refer the reader to [59] for the amortization analysis.

- **Extract-Min:** We Extract-Min in an identical way to the original [59]. Assuming the root is sorted in order from heaviest to lightest priority, taking the lightest priority element up takes $O(\log n)$ work and $O(1)$ rounds. Specifically, for every item, check if the item next to it to the right is $\perp$. If so, this is the last item. The order does not need to be changed, since we took off only the last element. We handle the delete item in exactly the same way [59] does [10] Again, we match the work of [59], and refer the reader to [59] for the amortization analysis.

5.1 Correctness

We will now prove that the modified Priority Queue has the same behavior as [59] and, therefore, we can rely on the [59] correctness proof. Note that we replicate the procedures for Insert, Extract-Min, and Initialization. Therefore, if we prove that the Parallel Decrease-Key, PULLUP(i), and PUSHDOWN(i) procedures all work as exactly as in [59], all of the proofs of [59] still hold.

We will begin with PULLUP(i) and PUSHDOWN(i). By construction, we are moving the same amount of items between levels. For each PUSHDOWN(i), we move all items at level i to level $i+1$. For each PULLUP(i), we move 2^{i-1} of the lightest items from level $i+1$ to level i. Therefore, both operations mimic the original operations, and the data structure works as originally intended. Let $\mathcal{T}$ be a complete binary tree with B buffer space at each node, as in [59].

For the Parallel Decrease-Key procedure, we use a lemma similar to Lemma 3 of [59] that does not require any assumptions:

Lemma 9. *Let $\mathcal{S} := op_1, \ldots op_N$ be a sequence of operations on the tree $\mathcal{T}$. A modified procedure maintains the structure of the Priority Queue if and only if after op_t, it holds that: (1) Extract-Min returns the item with the lightest priority. (2) No node $u \in \mathcal{T}$ is overflowed. That is, the number of items in u does not exceed B.*

We must prove the above holds for the combination of PushDown-manual maintenance, insertion of w elements in parallel, and PullUp-manual maintenance (all in succession).

We assume that before PushDown-manual maintenance, $\mathcal{T}$ has all items in each node ordered, and that the lightest items are in the root. This is given by [59]. Recall that for w elements, PushDown-manual maintenance will execute PUSHDOWN(0),... PUSHDOWN(γ), where $\gamma = \log w$. That means the current lightest item is at level $\gamma + 1$. Now, we insert all w elements to level γ. As shown in our analyses of Parallel Decrease-Key, assuming doubling the size of the buffer at each node, no node at level γ is overflowed.

At this point, the lightest item in the tree is either at level γ or $\gamma + 1$. This depends on whether one of the w items had priority lighter than the previous minimum. Either way, we execute PULLUP(γ),... PULLUP(0); the PullUp-manual maintenance. Because the first step examines all vertices at level γ and their children

[10] As in [59] we delete an item by inserting an item with $+\infty$ priority, and having it "kill" all its siblings as it travels down the tree during the regular maintenance cycles.

at level $\gamma + 1$, the minimum will be pulled up no matter where it is. This means once we have completed the manual maintenance, the Extract-Min operation will return the minimum. After manual maintenance is done, w steps of maintenance occur, meaning the counter that dictates maintenance will be the same as if we followed the traditional decrease key and inserted all elements at the root. This will not take the minimum out of the root node, as this maintenance work is in the original in [59]. Therefore, clause 1 of Lemma 9 holds. After w maintenance steps, the distribution of items is exactly the same as if we inserted all w "new" items at the root sequentially. Observe that the trajectory of the items is the same and the ordering of the items is preserved.

If these items are heavier than all those we pushed down, they will stay at level γ. After w operations, they will only come up if they are lighter than those beneath them. Lemma 6 of [59] proved that with scheduled maintenance, no node $u \in \mathcal{T}$ is overflowed after N operations.

We have shown that inserting w items in parallel distributes them along the Priority Queue in the same way that w traditional decrease-key operations would. Additionally, the counter of operations increases by w. This means that Lemma 6 [59] holds, and therefore, clause 2 of Lemma 9 holds.

5.2 Complexity Analysis

Each $O(\log n)$-wise independent hash function evaluation takes $O(\log n)$ secure operations. We do it once for each insert and each decrease key operation. In addition, recall that we showed that the parallel decrease key has the same running time as the one decrease key in [59]. Therefore, since we have shown that work is the same as in [59], the $O(\log n)$ amortized per operation work cost holds.

However, the round complexity is different. We pay $O(\log\log\log n)$ rounds for any PushDown(i) or PullUp(i) for any level i. Each Extract-Min and Insert takes $O(1)$ rounds. Every w Parallel Decrease-Key operation takes the following two steps: (a) $O(\log n)$ rounds to insert and sort the w keys into level $\log w$ in parallel. (b) perform manual maintenance, which takes $O(\log w \cdot \log\log\log n)$ rounds. For scheduled maintenance, we can amortize the rounds in the same way we do the work. We refer the reader to [59] for this amortization. This means the amortized cost of maintenance is $O(\log n \log\log\log n)$. Let r denote the number of times we call w-parallel decrease-key operations. (Recall that for our version of Dijkstra, $r = 2n$.) Because $w \leq n$, we will combine the cost of manual and non-manual maintenance. Putting it all together, the cost in all rounds becomes $O(r \cdot \log n \cdot \log\log\log n)$.

6 Secure Dijkstra and its Analysis

Given all the build-up, the general outline of our algorithm is straightforward: given any graph G in a secret-shared adjacency list representation, we first change the representation to the one that is MPC-friendly. Specifically, if the adjacency list of G has arbitrary alphanumeric vertex names, we can first convert it to ordered integer vertex names using our name conversion algorithm 4.3. This gives us a representation of secret-shared reset-panted of the adjacency list where all vertices and numbers from 1 to V appear in sorted order. We can now run our *d-normalization* algorithm 4.1 and put each d-length adjacency list into USE-ONCE KEY-VALUE STORE data structure. This is exactly where our savings come from: since in Dijkstra, we explore each vertex only once, we need to retrieve each block only once, and USE-ONCE KEY-VALUE STORE incurs only additive overhead for the retrieval of each block. The efficiency of our algorithm comes from the fact that we can process d edges in parallel without any overhead and then hide, for each node, if there are more edges or we are just processing another d edges for another vertex. We keep the following secret-shared data-structures:

- secret-shared USE-ONCE KEY-VALUE STORE with $2V$ entries for d-normalized adjacency list;
- secret-shared DORAM array of length $2V$ that maintains distances to all vertices. As in regular Dijkstra, initially, all distances are set to $+\infty$. Since we are also given a secret-sharing of a start vertex, we initialize the start vertex to have distance 0 inside DORAM, hence not revealing what the start vertex is;
- we also initialize the OPQ 5 with $2V$ entries, where distance is the priority.

Algorithm 1 Parallel Edge Relaxation

Input:

- $[\![e_i]\!] = ([\![k_1]\!], [\![x_1]\!]), \ldots, ([\![k_d]\!], [\![x_d]\!])$
- current distance $[\![h]\!]$ of the node being explored

PARALLEL RELAX($[\![e_i]\!]$):

1. $[\![a_1]\!] \ldots [\![a_d]\!] \leftarrow$ PARALLEL-DORAM.DISJOINT.READ($[\![k_1]\!], \ldots, [\![k_d]\!]$)
2. $[\![b_1]\!] \ldots [\![b_d]\!] \leftarrow$ RELAX($[\![x_1]\!], [\![h]\!], [\![a_1]\!]$), $\ldots$, RELAX($[\![x_d]\!], [\![h]\!], [\![a_d]\!]$)
3. PARALLEL-DORAM.DISJOINT.WRITE(($[\![k_1]\!], [\![b_1]\!]$), $\ldots$ ($[\![k_d]\!], [\![b_d]\!]$))
4. PQ.PARALLEL-DECREASE-KEY(($[\![k_1]\!], [\![b_1]\!]$), $\ldots$, ($[\![k_d]\!], [\![b_d]\!]$))
5. **return** $[\![$Continue?$]\!]$

RELAX($[\![x]\!], [\![h]\!], [\![a]\!]$):

1. **return** $\mathcal{F}_{ABB}$.min$\{[\![x + h]\!], [\![a]\!]\}$

6.1 Edge Relaxation for a single block

A single block consists of d weighted edges. The PARALLEL EDGE RELAXATION subroutine for vertex v_i begins by reading in DORAM the current estimate of distance h of v_i. For all edges in a block of edges from node v_i, we read in parallel distances to all vertices that v_i points to in the current block. Of course, for $\perp$'s we pretend to read as if these are real entires. We do this with a parallel DORAM read, which takes at most $O(\log V)$ rounds and $O(d \log V)$ work. We call these distances $(a_1, \ldots a_d)$. Next, we relax all edges by take the minimum of each a_i and $h + x$, the distance from v_i to that neighbor. We call this $(b_1, \ldots b_d)$. This takes constant rounds and $O(d)$ secure operations. Next, we do a parallel DORAM write, which takes $O(d \log V)$ work and $O(\log V)$ rounds by updating all the distances in DORAM and executing d decrease key priority queue operations. This takes $O(d \log n)$ work and $O(\log V \cdot \log \log \log V)$ rounds. Finally, we return the Cotninue? secret-shared marker, which lets the algorithm know if there are additional d-blocks for the same vertex. This takes constant rounds. In total, a single d block Edge Relaxation subroutine takes $O(d \log V)$ work and $O(\log d \log \log \log V)$ rounds. Since $d < n$, we get $O(V \log V)$ work and $O(\log V \log \log \log V)$ rounds.

Algorithm 2 Secure Oblivious Dijkstra

Input: $[\![G]\!]$ = secret-shared adjacency list with n vertices ($k_1 ... k_n$), m edges, and $[\![s]\!]$ as start node.

SECURE-DIJKSTRA($[\![G]\!]$):

1. CHANGE-GRAPH-REPRESENTATION($[\![G]\!]$) ▷ Includes initializing KVS
2. PARALLEL-DORAM.INITIALIZE($([\![k_1]\!], [\![\infty]\!]), \ldots, ([\![k_n]\!], [\![\infty]\!])$)
3. PQ.INITIALIZE($([\![k_1]\!], [\![\infty]\!]), \ldots, ([\![k_w]\!], [\![\infty]\!])$)
4. DORAM.WRITE($[\![s]\!], [\![0]\!]$)
5. PQ.DECREASE-KEY($[\![s]\!], [\![0]\!]$)
6. $[\![x_q]\!] \leftarrow [\![s]\!]$
7. **for** num from 1 to $2n$ **do**
8. $([\![\vec{e_i}]\!]) \leftarrow$ KVS.READ($[\![x_q]\!]$)
9. RELAX($([\![\vec{e_i}]\!])$)
10. **if** [Continue?] **then**
11. Pretend to EXTRACT-MIN
12. $x_q \leftarrow x_{q+1}$
13. **else**
14. $x_q \leftarrow$ EXTRACT-MIN
15. **end if**

Final output distance vector is now stored in DORAM.

6.2 Secure Dijkstra

In the Secure Oblivious Dijkstra algorithm, we start by changing the graph representation to the d-normalized replicated adjacency list, and initializing the KVS. This takes $O(E + V \log V)$ work and $O(\log V)$ rounds. We then initialize the Parallel DORAM structure, which takes $O(V \log V)$ work and $O(V \log V)$ rounds. Then, we initialize the Parallel Oblivious Priority Queue, which takes $O(V \log V)$ work and $O(V \log \log \log V)$ rounds. Then, we begin our main loop, which we will execute $2V$ times, to go through all $2V$ entries of our d-normalized replicated adjacency list. $[\![x_q]\!]$ represents the name of the current node we are exploring. We start by setting x_1 equal to the source node s. Then, we execute KVS.read on x_q, getting the associated entry with that node. This takes $O(d)$ work and $O(1)$ rounds. We then do the Edge Relaxation algorithm. Finally, we do extract min, or pretend to, depending of the Continue? marker which the Edge Relaxation returned. Both paths of the if/else statement take the same amount of work, and are indistinguishable. They both take $O(\log V)$ work and $O(1)$ rounds. The total work for inside the loop comes from the Relaxation subroutine, which is $O(d \log V)$. The total rounds is $O(\log V \log \log \log V)$

Since we execute this loop $2V$ times, we multiply all work and rounds within the loop by $2V$. That means we do $O(V \cdot d \log V)$ work and $O(V \log V \log \log \log V)$ rounds. Because $d = 2\frac{E}{V}$, the total work from the loop is $O(E \log V)$. Putting it all together, the total cost is $O((E + V) \log V)$ work and $O(V \log V \log \log \log V)$ rounds.

For security proofs, we observe that we invoke secure algorithms as building blocks, and all our opera-

tions are data-independent and hence oblivious. By the composability framework, the overall protocol does, therefore, remain secure.

The most efficient instantiation of our Dijsktra algorithm would be a three-party honest-but-curious setting without collusion. For this model one can use a secure shuffle of [28], a secure merge of [19], and ABB implementations and DORAM from [41]. For two-party case, use secure shuffle of [30], DORAM [62] and secure merge of [19].

7 Conclusions and Further Work

To conclude, our d-NORMALIZED REPLICATED ADJACENCY LIST conversion takes $O(1)$ rounds and $O(V + E)$ secure operations. We also show $O(\log V)$ rounds and $O((V + E) \log V)$ oblivious graph renaming algorithm from an adjacency list where vertices are arbitrary alphanumeric labels to an equivalent adjacency list representation, where vertices are integers and appear ordered in the resulting secret-shared adjacency list, from 1 to V. We believe both algorithms are of independent interest and will have additional applications.

Our Secure Dijkstra algorithm takes $O((V+E) \log V)$ secure operations and $O(V \cdot \log V \cdot \log \log \log V)$ rounds. We remark that our version of Dijkstra's algorithm is inherently *oblivious*. Therefore, for the constant-round MPC protocols, implementing our Dijkstra's algorithm using Yao's Garbled Circuits and resorting to Garbled RAM only in places where we invoke DORAM may result in better asymptotic computation and communication complexity. Our general techniques are likely to also be applicable to making other secure graph algorithms more efficient, such as computing BFS and MST.

Acknowledgments

I would like to thank the anonymous reviewers of CRYPTO 2024 PC for their helpful suggestions. I am grateful to the instructors at UCLA's Olga Radko Math Circle for hooking me on mathematics in second grade and continuing over the years to introduce me to high-level ideas well beyond the school curriculum. I am grateful to Sanjam Garg, Vassilis Zikas, and Yuval Ishai for multiple inspiring conversations and for providing feedback on early drafts of this manuscript. Finally, I want to thank my dad, Rafail Ostrovsky, for inspiring me to study theoretical computer science from an early age, pointing me to multiple resources on cryptography, and patiently answering numerous questions, even in the middle of the night.

References

[1] Abraham, I., Fletcher, C.W., Nayak, K., Pinkas, B., Ren, L.: Asymptotically tight bounds for composing oram with pir. In: PKC. pp. 91–120. Springer (2017)

[2] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms (1974)

[3] Ajtai, M., Komlós, J., Szemerédi, E.: An o(nlog n) sorting network. In: STOC. pp. 1–9 (1983)

[4] Aly, A., Cleemput, S.: An improved protocol for securely solving the shortest path problem and its application to combinatorial auctions. ePrint p. 971 (2017)

[5] Aly, A., Cleemput, S.: A fast, practical and simple shortest path protocol for multiparty computation. In: ESORICS. pp. 749–755. Lecture Notes in Computer Science (2022)

[6] Aly, A., Cuvelier, E., Mawet, S., Pereira, O., Vyve, M.V.: Securely solving simple combinatorial graph problems. In: Financial Crypto. pp. 239–257. Lecture Notes in Computer Science (2013)

[7] Anagreh, M., Laud, P., Vainikko, E.: Privacy-preserving parallel computation of minimum spanning forest. SN Comput. Sci. p. 448 (2022)

[8] Anagreh, M., Vainikko, E., Laud, P.: Parallel privacy-preserving shortest paths by radius-stepping. In: PDP. pp. 276–280 (2021)

[9] Asharov, G., Hamada, K., Ikarashi, D., Kikuchi, R., Nof, A., Pinkas, B., Takahashi, K., Tomida, J.: Efficient secure three-party sorting with applications to data analysis and heavy hitters. In: CCS (2022)

[10] Asharov, G., Komargodski, I., Lin, W., Nayak, K., Peserico, E., Shi, E.: Optorama: Optimal oblivious RAM. J. ACM pp. 4:1–4:70 (2023)

[11] Asharov, G., Komargodski, I., Lin, W., Peserico, E., Shi, E.: Optimal oblivious parallel RAM. In: SODA. pp. 2459–2521 (2022)

[12] Aziz, M., Alhadidi, D., Mohammed, N.: Secure approximation of edit distance on genomic data. BMC Med Genomics p. 41 (2017)

[13] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: STOC. pp. 1–10 (1988)

[14] Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: EUROCRYPT. pp. 169–188 (2011)

[15] Berger, B., Waterman, M.S., Yu, Y.W.: Levenshtein distance, sequence comparison and biological database search. IEEE transactions on information theory pp. 3287–3294 (2020)

[16] Bienstock, A., Patel, S., Seo, J.Y., Yeo, K.: Near-optimal oblivious key-value stores for efficient psi, PSU and volume-hiding multi-maps. In: USENIX. pp. 301–318 (2023)

[17] Blanton, M., Steele, A., Aliasgari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: CCS. pp. 207–218 (2013)

[18] Blelloch, G.: Scans as primitive parallel operations. IEEE Transactions on Computers pp. 1526–1538 (1989)

[19] Blunk, M., Bunn, P., Dittmer, S., Lu, S., Ostrovsky, R.: Secure merge in linear time and o(log log N) rounds. ePrint p. 590 (2022)

[20] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: CESORICS. pp. 192–206 (2008)

[21] Brickell, J., Shmatikov, V.: Privacy-preserving graph algorithms in the semi-honest model. In: ASIACRYPT. pp. 236–252. Lecture Notes in Computer Science (2005)

[22] Brodal, G.S.: Worst-case efficient priority queues. In: SODA. pp. 52–58 (1996)

[23] Brodal, G.S., Lagogiannis, G., Tarjan, R.E.: Strict fibonacci heaps. In: STOC. pp. 1177–1184 (2012)

[24] Bunn, P., Katz, J., Kushilevitz, E., Ostrovsky, R.: Efficient 3-party distributed ORAM. In: SCN. pp. 215–232 (2020)

[25] Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000)

[26] Canetti, R.: Universally composable security. J. ACM pp. 28:1–28:94 (2020)

[27] Carter, H., Lever, C., Traynor, P.: Whitewash: outsourcing garbled circuit generation for mobile devices. In: ACSAC. pp. 266–275 (2014)

[28] Chan, T.H., Katz, J., Nayak, K., Polychroniadou, A., Shi, E.: More is less: Perfectly secure oblivious algorithms in the multi-server setting. In: ASIACRYPT. pp. 158–188 (2018)

[29] Chan, T.H., Shi, E., Lin, W., Nayak, K.: Perfectly oblivious (parallel) RAM revisited, and improved constructions. In: ITC 2021. pp. 8:1–8:23

[30] Chase, M., Ghosh, E., Poburinnaya, O.: Secret-shared shuffle. In: ASIACRYPT. pp. 342–372 (2020)

[31] Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: STOC. pp. 11–19 (1988)

[32] Chen, C., Cui, J., Liu, G., Wu, J., Wang, L.: Survey and open problems in privacy preserving knowledge graph: Merging, query, representation, completion and applications (2020)

[33] Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: CRYPTO. pp. 247–264 (2003)

[34] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO. pp. 643–662 (2012)

[35] Decoster, K., Billard, D.: HACIT: A privacy preserving and low cost solution for dynamic navigation and forensics in VANET. In: VEHITS. pp. 454–461 (2018)

[36] Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)

[37] Dittmer, S., Ostrovsky, R.: Oblivious tight compaction in o(n) time with smaller constant. In: SCN. pp. 253–274 (2020)

[38] Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: CCS. pp. 523–535 (2017)

[39] Falk, B.H., Nema, R., Ostrovsky, R.: Linear-time 2-party secure merge from additively homomorphic encryption. J. Comput. Syst. Sci. pp. 37–49 (2023)

[40] Falk, B.H., Ostrovsky, R.: Secure merge with o(n log log n) secure operations. In: ITC. pp. 7:1–7:29. LIPIcs (2021)

[41] Falk, B.H., Ostrovsky, R., Shtepel, M., Zhang, J.: Gigadoram: Breaking the billion address barrier. In: USENIX (2023)

[42] Feng, Y., Ma, H., Chen, X., Zhu, H.: Secure and verifiable outsourcing of sequence comparisons. In: ICT-EurAsia. pp. 243–252 (2013)

[43] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. In: FOCS. pp. 338–346 (1984)

[44] Garg, S., Lu, S., Ostrovsky, R.: Black-box garbled RAM. In: FOCS. pp. 210–229 (2015)

[45] Garg, S., Lu, S., Ostrovsky, R., Scafuro, A.: Garbled RAM from one-way functions. In: STOC. pp. 449–458 (2015)

[46] Garimella, G., Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: Oblivious key-value stores and amplification for private set intersection. In: CRYPTO. pp. 395–425 (2021)

[47] Goldreich, O.: Secure multi-party computation. Online book draft (1998)

[48] Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM (1986)

[49] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: STOC. pp. 218–229 (1987)

[50] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM pp. 431–473 (1996)

[51] Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint (2014)

[52] Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: ICISC. pp. 202–216 (2012)

[53] Hazay, C., Scholl, P., Soria-Vazquez, E.: Low cost constant round MPC combining BMR and oblivious transfer. In: ASIACRYPT. pp. 598–628 (2017)

[54] Heath, D., Kolesnikov, V., Ostrovsky, R.: Epigram: Practical garbled RAM. In: EUROCRYPT. pp. 3–33 (2022)

[55] Heath, D., Kolesnikov, V., Ostrovsky, R.: Tri-state circuits - A circuit model that captures RAM. In: CRYPTO. pp. 128–160 (2023)

[56] Hemenway, B., Noble, D., Ostrovsky, R., Shtepel, M., Zhang, J.: DORAM revisited: Maliciously secure RAM-MPC with logarithmic overhead. In: TCC. pp. 441–470 (2023)

[57] Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: CCS. pp. 772–783 (2012)

[58] Jafargholi, Z., Larsen, K.G., Simkin, M.: Optimal oblivious priority queues and offline oblivious RAM. ePrint p. 237 (2019)

[59] Jafargholi, Z., Larsen, K.G., Simkin, M.: Optimal oblivious priority queues. In: SODA. pp. 2366–2383 (2021)

[60] Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: ASIACRYPT. pp. 506–525. Lecture Notes in Computer Science (2014)

[61] Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: Oblivm: A programming framework for secure computation. In: SOSP. pp. 359–376 (2015)

[62] Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: TCC. pp. 377–396 (2013)

[63] Lu, S., Ostrovsky, R.: How to garble RAM programs. In: EUROCRYPT. pp. 719–734 (2013)

[64] Lu, S., Ostrovsky, R.: Black-box parallel garbled RAM. In: CRYPTO. pp. 66–92 (2017)

[65] Mood, B., Gupta, D., Carter, H., Butler, K.R.B., Traynor, P.: Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In: EuroS&P (2016)

[66] Myers, S.A., Sharma, A., Gupta, P., Lin, J.: Information network or social network?: the structure of the twitter follow graph. In: WWW. pp. 493–498 (2014)

[67] Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: CRYPTO. pp. 681–700 (2012)

[68] Noble, D.: Distributed Oblivious RAM: Progress and Pitfalls. Ph.d. thesis, University of Pennsylvania, Philadelphia, PA (May 2024)

[69] Noble, D., Falk, B.H., Ostrovsky, R.: Metadoram: Breaking the log-overhead information theoretic barrier. ePrint p. 11 (2024)

[70] Ostrovsky, R.: Efficient computation on oblivious rams. In: STOC. pp. 514–523 (1990)

[71] Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: STOC. pp. 294–303 (1997)

[72] Patel, S., Persiano, G., Raykova, M., Yeo, K.: Panorama: Oblivious RAM with logarithmic overhead. In: FOCS. pp. 871–882 (2018)

[73] Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: STOC. pp. 73–85 (1989)

[74] Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: A programming language for generic, mixed-mode multiparty computations. In: SOSP. pp. 655–670 (2014)

[75] Shi, E.: Path oblivious heap: Optimal and practical oblivious priority queue. In: SOSP. pp. 842–858 (2020)

[76] Songhori, E.M., Hussain, S.U., Sadeghi, A., Schneider, T., Koushanfar, F.: Tinygarble: Highly compressed and scalable sequential garbled circuits. In: SOSP. pp. 411–428 (2015)

[77] Stefanov, E., van Dijk, M., Shi, E., Chan, T.H., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious ram protocol. J. ACM pp. 18:1–18:26 (2018)

[78] Vadapalli, A., Henry, R., Goldberg, I.: Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. In: USENIX (2023)

[79] Wang, L., Liu, G., Sun, L.: A secure and privacy-preserving navigation scheme using spatial crowdsourcing in fog-based vanets. Sensors p. 668 (2017)

[80] Wang, X., Ranellucci, S., Katz, J.: Global-scale secure multiparty computation. In: CCS. pp. 39–56 (2017)

[81] Wu, D.J., Zimmerman, J., Planul, J., Mitchell, J.C.: Privacy-preserving shortest path computation. In: NDSS (2016)

[82] Yang, Y., Peceny, S., Heath, D., Kolesnikov, V.: Towards generic MPC compilers via variable instruction set architectures (visas). In: CCS. pp. 2516–2530 (2023)

[83] Yao, A.C.: Protocols for secure computations (extended abstract). In: FOCS. pp. 160–164 (1982)

[84] Zahur, S., Evans, D.: Obliv-c: A language for extensible data-oblivious computation. ePrint p. 1153 (2015)

[85] Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: CCS. pp. 813–826 (2013)