GraphOS: Towards Oblivious Graph Processing

Javad Ghareh Chamani
HKUST
jgc@cse.ust.hk

Ioannis Demertzis
UC Santa Cruz
idemertz@ucsc.edu

Dimitrios Papadopoulos
HKUST
dipapado@cse.ust.hk

Charalampos Papamanthou
Yale University
charalampos.papamanthou@yale.edu

Rasool Jalili
Sharif University of Technology
jalili@sharif.edu

ABSTRACT

We propose GraphOS, a system that allows a client that owns a graph database to outsource it to an untrusted server for storage and querying. It relies on doubly-oblivious primitives and trusted hardware to achieve a very strong privacy and efficiency notion which we call oblivious graph processing: the server learns nothing besides the number of graph vertexes and edges, and for each query its type and response size. At a technical level, GraphOS stores the graph on a doubly-oblivious data structure, so that all vertex/edge accesses are indistinguishable. For this purpose, we propose Omix++, a novel doubly-oblivious map that outperforms the previous state of the art by up to 34×, and may be of independent interest. Moreover, to avoid any leakage from CPU instruction-fetching during query evaluation, we propose algorithms for four fundamental graph queries (BFS/DFS traversal, minimum spanning tree, and single-source shortest paths) that have a fixed execution trace, i.e., the sequence of executed operations is independent of the input. By combining these techniques, we eliminate all information that a hardware adversary observing the memory access pattern within the protected enclave can infer. We benchmarked GraphOS against the best existing solution, based on oblivious relational DBMS (translating graph queries to relational operators). GraphOS is not only significantly more performant (by up to two orders of magnitude for our tested graphs) but it eliminates leakage related to the graph topology that is practically inherent when a relational DBMS is used unless all operations are “padded” to the worst case.

PVLDB Reference Format:
doi:10.14778/3625054.3625067

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jgharehchamani/graphos.

1 INTRODUCTION

Motivated by numerous real-world applications where the outsourced sensitive data can be modeled as graphs (e.g., semantic web, GIS, social networks, web graphs, transportation networks), in this work we focus on the problem of privacy-preserving graph processing on the cloud. We consider a setting with two parties, a client (data owner) and an untrusted server. The first is willing to outsource her sensitive graph database to the second under encryption, and later requests the evaluation of graph queries. Crucially, we want to restrict the information that is revealed to the server to a minimum. E.g., initially the server learns just the size of the graph (number of vertexes and number of edges), whereas for every graph query the server only learns the size of the result and the query type. We refer to this as oblivious graph processing. Moreover, we want to limit the client’s participation in computing. In a standard client-server model the client issues a query and receives a response; no additional interaction should be required and the computation should be undertaken solely by the server.

From Oblivious Relational DBMS to Oblivious Graph Processing. One way to achieve graph processing is via relational database management systems (DBMS) that can be naturally used for graph query workloads [65, 66]. Vertexes and edges are stored in relational tables and graph queries are translated to relational database query operators (e.g., multiple self-joins) on these tables. Privacy-preserving DBMS have been proposed previously, e.g., CryptDB [89] and Monomi [111]. However, these systems leak sensitive information even before executing any graph query1 so they fail to achieve our strong privacy requirement outlined above.

Recently, Zheng et al. [124], Eskandarian et al. [47], and Pribe et al. [91] proposed oblivious relational DBMS. These systems combine trusted hardware with oblivious algorithms to minimize the leaked information to just the size of accessed and created tables. It is important to note that trusted hardware alone [14, 96] is not sufficient as it does not hide the memory access pattern; enclave side channels allow attackers to exploit data-dependent memory accesses to extract enclave secrets [72, 76, 112]. To defend against these attacks, one must guarantee that all algorithms running inside the trusted hardware are oblivious, i.e., data-input independent. In practice, an oblivious algorithm means that for any two input instances of the same size, the algorithm executions (including their resulting memory accesses patterns) are indistinguishable. Hence, one may hope that these systems that combine the two techniques for relational databases can achieve oblivious graph processing.

Surprisingly, it turns out this is not true. When an oblivious relational DBMS is used for graph processing it may still leak sensitive graph information due to the need to translate graph queries to

---

1They are based on deterministic and order-preserving encryption that leak the distribution of the input data and/or their relative order. Devastating leakage-abuse attacks have been proposed against both of them (e.g., [85]).
We implement GraphOS using step 1. We propose a new we outline the novelties of GraphOS. (BFS) traversal query, as shown in Figure 1. With a relational DBMS, weve revealing the accessed element, being in the ballpark of prior plaintext asymptotic complexity and practical performance than the state- Giraph [13], GraphLab [80], Trinity [98]). approaches of "native graph" DBMS proposals (e.g., Pregelix [20], Giraph [13], GraphLab [80], Trinity [98]). Omix++ achieves a better asymptotic complexity and practical performance than the state-of-the-art DOMAP (Omix) [83] and can be used as a stand-alone solution in many applications besides graph queries as we show in Sec 6. We build Omix++ by storing an AVL tree inside an array in OBLIX [83]. Crucially, we use a new eviction strategy that evicts one-path-at-a-time individually, which improves the performance of Omix++ over the single key-value DOMAP that can be constructed based on the approach [83], both asymptotically (more than a logarithmic factor) and experimentally.

We also propose an oblivious initialization process for Omix++, which is significantly faster than the only existing one for DOMAP (setting up an empty DOMAP and obliviously inserting each key-value pair). Finally, to alleviate the context-switching overhead when transferring data between unprotected and protected memory (which can be significant in a trusted enclave) we propose a path-caching mechanism to temporarily store eviction results inside the protected memory of the trusted hardware. Each eviction corresponds to a path of the DORAM tree; since the adversary already knows the corresponding leaves, there is no need to obliviously access them and no extra leakage is introduced due to caching.

Graph-algorithms with fixed execution trace. It is important to note that using Omix++ is not sufficient for eliminating query execution leakage because, even though the code is loaded into the trusted hardware enclave encrypted, still the specific position of each fetched instruction is observable by a "hardware-level" attacker at the machine where the enclave lies. One could try to eliminate this leakage by loading the code itself in a doubly-oblivious primitive; indeed this approach has been explored by recent works [1, 122] but it can significantly hurt performance as discussed in Sec 2.

In this work, we achieve an efficient solution, by proposing graph query algorithms that have a deterministic execution trace, i.e., the sequence of executed CPU instructions executed is fixed a priori (modulo the graph size) and independent of the specific input values. In particular, we propose algorithms for BFS/DFS, minimum spanning tree, and single-source shortest path queries that have a deterministic execution trace and only reveal the vertex/edge accesses each time. Our algorithms eliminate all data-dependent loops and branches by using a small number of dummy operations and the loop-coalescing technique [78]. E.g., instead of padding the number of neighbor accesses to the worst case (number of vertices) for each vertex in BFS, we hide the transition between vertexes in the BFS algorithm to prevent any access pattern leakage.

These techniques work in a complementary manner with our DOMAP in GraphOS by first loading the graph into a Omix++ and then executing our graph algorithms with fixed execution trace replacing all graph accesses with calls to the DOMAP. Doubly-oblivious primitives eliminate any leakage from the graph data-accesses, whereas the deterministic sequence of fetched and executed instructions eliminates any leakage from instruction-accesses. Implementation and benchmarking. We implement GraphOS using Intel-SGX as a proof of concept and compare it with Opaque, the oblivious relational DBMS of [124] on a number of graph algorithms, in terms of leakage and query performance. Note that GraphOS can be implemented on any trusted hardware that provides specific characteristics explained in Sec 3.1. As described in more detail below, GraphOS outperforms Opaque for all query types (by up to two orders of magnitude), and achieves overall less leakage (strictly less for BFS/DFS traversal and single-source shortest paths, and

![Figure 1: BFS Traversal. Tables V, E contain graph vertexes and edges. (Step 1:) performs a selection on E for initial vertex s—server learns vertex s has 3 neighbors. (Step 2:) joins the previous output with table E—server learns 2 vertexes are 2 hops away from s. (Step 3:) joins the previous output with table E—server learns that 1 vertex is 3 hops away.](image-url)
equivalent for minimum spanning tree). All our implementations are publicly available in [51] constituting also the first open-source implementation of doubly oblivious primitives.

**Experimental evaluation.** We experimentally evaluate both the performance of our WOMAP (OMIX++) and our oblivious graph processing scheme GraphOS. The results are shown in Sec 6.

OMIX++ evaluation. For OMIX++, we compare its performance with the single-value WOMAP built from [83], which we call WMIX, in three applications: private contact discovery, key transparency logs, and searchable encryption. The results show that an OMIX++ access (look-up) is overall 1.8–20× faster, resulting in the most efficient existing WOMAP. This improvement is larger in applications that impose access in-batch. E.g., used for searchable encryption, OMIX++ leads to 17× and 25× improvement over WMIX in search and update operations, respectively. Signal, the secure messaging app [8], has recently moved to adopt techniques inspired by those of [83] for private contact discovery (via oblivious key/value look-ups) [90]. Our experimental evaluation shows that OMIX++ significantly outperforms [83], e.g., one look-up access with 224 entries takes 37ms computation time with OMIX++ vs. 767ms with WMIX.

GraphOS evaluation. We compare its performance with OPAQUE, the state-of-the-art approach for private graph processing from oblivious relational DBMS [124]. We measure the execution times for initialization, adding/removing/retrieving vertex and edge, BFS/DFS traversal, minimum spanning tree, and single-source shortest path for various graph sizes/denseness. Our results show that GraphOS is 2.6–13.6× and 2.4–136× faster for adding/removing an edge and a vertex, respectively, and 95–150× for retrieving one. Its query execution time is 6–410× smaller for BFS/DFS, 1.4–86× for MST, 1–22× for SSSP. Recall that for SSSP and BFS/DFS OPAQUE reveals information about the graph topology; eliminating this leakage (via worst-case padding) would make it prohibitively slower. We also considered a distributed version of GraphOS using the split-ORAM approach of [38]. Finally, we tested an “integrated approach” where GraphOS is deployed on-the-fly to build its indexes when a query is to be processed. That is, upon receiving a query, we create all the required for GraphOS indexes, and then we execute this graph query. Somewhat surprisingly, even in this configuration, the query time of GraphOS (which includes the initialization costs for building the indexes) is significantly faster than OPAQUE. It is worth noting that usually better security is achieved at the cost of worse performance. However, compared to OPAQUE, GraphOS not only has less leakage for graph queries but is also more efficient.

2 OTHER RELATED WORK

Here we discuss works relevant to ours, besides those on oblivious relational DBMS and doubly-oblivious primitives described above.

**Oblivious execution of arbitrary code.** Eliminating the leakage from memory accesses when running programs in the trusted hardware enclave has been the focus of a recent line of works, e.g., [77, 95, 102] that explore this based on different hardware assumptions. The most advanced of these works focus on oblivious execution of arbitrary code [1, 122]. At a high level, this is achieved by loading the code itself on doubly-oblivious storage/memory. Obfuscuro [1] uses an oblivious array for the data and one for the code in order to make arbitrary program execution oblivious (formally, their target is cryptographic obfuscation). Klotzki [122] improved the performance of Obfuscuro at the cost of extra leakage. These approaches can also be used to achieve double-obliviousness for any graph algorithm; however, they both have limitations in terms of low efficiency/scalability. Moreover, they assume that both the input data and the program must fit inside the enclaves, which makes them not directly applicable to our case. Our OMIX++ can be used as a drop-in replacement both to address the above limitation and to improve their performance (e.g., replacing multiple sequential scans over the position map with faster oblivious accesses).

**MPC-based doubly-oblivious approaches.** A different approach (in a different model) is based on secure multi-party computation (MPC), where one or more parties secret-share their data across multiple non-colluding servers [5, 16, 17, 44, 45, 48, 55, 56, 71, 75, 78, 87, 108, 113, 117, 121]. The vast majority of these works focus on challenges arising from the communication and interactive nature of MPC [3, 6, 7, 9, 61, 114] that are not applicable to our setting. The doubly-oblivious nature of these approaches can inspire the designing of doubly-oblivious algorithms for hardware enclaves. ObliVM [78] proposes a platform for general-purpose oblivious computation and GraphSC [87] builds a platform on top of ObliVM specifically for distributed graph computation. GraphSC relies on garbled circuits and is reportedly up to three orders of magnitude slower than OPAQUE [124]. [78] also proposed an optimized oblivious DFS in the MPC setting; however these approaches are not always suitable for trusted hardware environments (see Sec 6.4).

**Other doubly-oblivious approaches.** Recently, Shi [99] proposed a novel doubly-oblivious heap which we appropriately implemented in trusted execution environment (TEE) and integrated it with GraphOS (for more efficient SSSP queries). ZeroTrace [95] proposes doubly-oblivious PathORAM and CircuitORAM constructions; however, as shown in [83] it is outperformed by Oblix.

Shroid [79] parallelizes across multiple co-processors the Binary Tree ORAM [101]—both Shroid and Binary Tree ORAM can trivially be doubly-oblivious but they require super-linear storage and increased (compared to PathORAM) access time. Pyramid ORAM [31] is a hierarchical ORAM designed for Intel SGX (requiring constant oblivious memory). POSUP [63] and MOSE [62] are two additional CircuitORAM-based approaches. Recently, Snoopy [35] introduced an efficient and secure doubly-oblivious key-value store designed for high-throughput, but with increased latency.

**Other ORAM approaches.** There are parallel/distributed/concurrent non doubly-oblivious approaches based on different models, i.e., relying on the existence of a trusted-proxy [33, 38, 59, 106]; the existence of multiple servers [22]; sharing (in a non doubly-oblivious manner) an encrypted log on top of a hierarchical ORAM [119], or on top of a tree-based ORAM [21]; requiring specialized-hardware [49]. RingORAM [93] is a (non doubly-oblivious) PathORAM-based approach with a more efficient eviction strategy. PRO-ORAM [109] is a read-only ORAM running inside an enclave which requires $O(\sqrt{N})$ oblivious/private memory. ObliVM [2] recognizes the importance of doubly-oblivious algorithms supporting doubly-oblivious read and write operators; however, it does not discuss how to make the eviction algorithm doubly-oblivious. There is also a different, more theoretical line of works which focuses on the problem of Oblivious Parallel RAMs [18, 23–25, 27, 86, 92].
Oblivious relational DBMS. There exist two additional works for oblivious relational DBMS [47, 91], besides [124]. However, they both require large amounts of hardware-oblivious memory that is not compatible with early trusted hardware implementations.

Searchable/Structured graph encryption. Query evaluation over encrypted graphs has been studied previously. Chase and Kamara [26] propose the notion of structured encryption (SE) that can be used, as a special case, for encrypting a graph. Their solution supports limited types of graph queries (only neighbors and adjacency). SE leaks additional information about the structure of the graph, i.e., the neighbors of each vertex and the general graph topology. Subsequent SE graph-works (e.g., [67, 74, 82]) suffer from this limitation. SE-based solutions also offer support for a plethora of queries, including point/keyword-search queries[36, 37, 41, 42], range queries[38–40], and more general SQL queries[68].

3 PRELIMINARIES

Graph Notation. We consider directed graphs \((V, E)\) where \(V\) denotes the set of vertices and \(E\) denotes the set of edges. Each vertex \(v \in V\) is identified by a unique identifier \(id\). For simplicity, we assume that vertices are labeled from 1 to \(|V|\). Each directed weighted edge \((init, trm, weight) \in E\) has an integer weight and is associated with its initial \(init\) and terminal \(trm\) vertices.

3.1 Threat Model

We adopt a similar threat model as the one proposed by prior works that combine oblivious primitives with trusted execution environments (TEE), e.g., [83, 124]. We assume a hardware-level attacker that can fully observe the location of all memory accesses and can also control the server’s software stack, as well as have full control of the OS. Figure 2 illustrates a key difference between the TEE model and the client-server model. In the client-server model (which corresponds to standard ORAM), the client maintains a fully trusted machine that may be actively involved in parts of the computation (e.g., running the client-side routines of ORAM). In contrast, in the TEE model, the user encrypts his/her data and uploads it to the untrusted server. The computation is then fully outsourced to the TEE, which is located on the untrusted server that may be compromised by the hardware adversary.

Our adversary cannot attack the secure processor stealing information from inside it (including the processor’s secret keys). The adversary also cannot access the plaintext values loaded in the secure processor’s protected enclave portion of the memory (but can observe the accessed memory locations). Protected memory is encrypted with the processor’s secret key. In line with previous works, we consider as out of scope enclave side-channel leakages (e.g., cache-timing, power analysis, or other timing attacks—[19, 57, 60, 84, 97, 115]), rollback attacks [112], as well as denial-of-service attacks. There are complementary techniques (e.g., [1, 28, 32, 59, 102, 103, 122]) that can potentially mitigate such attacks.

Trusted Execution Environment (TEE). GraphOS and our proposed doubly-oblivious data structure can be implemented using any trusted hardware environment (e.g., Intel-SGX [81]; AMD enclave [70]; ARM TrustZone [10]) which provides isolation, sealing, and remote attestation. This is particularly important in view of the recent attacks against Intel-SGX [76, 112]. As a proof of concept, we implemented it using Intel-SGX [81]. Intel-SGX provides three important properties as follows. Isolation is provided by reserving a portion of the system’s memory, called Enclave Page Cache (EPC), used to store the user’s code and data and maintain its content in encrypted form (the total EPC memory size is 128MB). It is important to note, although the new version of Intel-SGX (v2) provides bigger EPC support, the performance of accessing small EPC (less than 128MB) is significantly better than larger EPC sizes due to the paging overhead [46]. Sealing allows the enclave to persistently store its data outside the secure environment. Remote attestation ensures the correctness of the running code.

3.2 Oblivious Primitives

Oblivious operations. Similar to [83], we assume oblivious routines for selection and comparison. \(Osel\) on input values \(a, b\) and selection bit \(c\) outputs \(a\) if \(c = 1\), else \(b\). \(Ocmp\) takes two \(l\)-bitlength inputs \(a, b\) and outputs 1, 0, −1 if \(a > b\), \(a = b\), or \(a < b\) respectively. Both routines must run obliviously. In our code, assuming that \(c\) is the all-0s or all-1s string of the same bitlength as \(a, b\) we implement \(Osel\) and \(Ocmp\) to return

\[
Osel(c, a, b) = (c \& a) | (\neg c \& b)
\]

\[
Ocmp(a, b) = -((a - b) \gg (l - 1)) + ((b - a) \gg (l - 1)),
\]

where !, &,, , >> are bitwise negation, conjunction, disjunction, and right-shift respectively. For brevity, we do not explicitly include \(Ocmp\) in our pseudocodes, but all comparisons are implemented with it (detailed pseudocodes with \(Osel\) and \(Ocmp\) can be found in the extended version). Our algorithms rely on oblivious sorting, i.e., sorting where the pattern of accessed memory locations does not depend on the actual data. We used Bitonic sort [15] that achieves \(O(N \log^2 N)\) complexity for \(N\) elements using \(Ocmp\) for comparison and two calls to \(Osel\) for oblivious swap.

Oblivious RAM (ORAM)/MAP (OMAP). This notion was introduced by Goldreich and Ostrovsky [54] more than two decades ago.
and has been further improved by a plethora of subsequent works (e.g., [12, 29, 34, 50, 88]). Intuitively, it hides array access pattern by accessing extra data blocks and random-shuffling after each access. Indeed, even repeated requests for the same data are indistinguishable from random. In this paper, we focus on PathORAM of Stefanov et al. [107]. In PathORAM, the server stores a binary tree of $N$ buckets each of which has $C$ blocks, and the client maintains a position map (a map from block id to leaf) and a stash that keeps overflowed and temporary blocks. In each block access, the client searches stash and if it is not found there it asks the server to send back the path corresponding to the target block (using position map). It then decrypts them and extracts the entry that matches the target index. The client chooses a new random leaf and then repositions the retrieved nodes from along the path (freshly re-encrypted), together with the entries in stash, in a way that "pushes" entries as deep as possible from root to leaf depending on their mapped positions. Any overflowing entries are stored in stash. The new encrypted path is stored at the server who updates the binary tree.

On the other hand, Oblivious MAP is a privacy-preserving version of a map data structure (we focus on the construction proposed by Wang et al. [118]). At a high level, it uses ORAM to implement an AVL-tree to store/access key-values in an oblivious way. In particular, OMAP provides three protocols, namely Setup, Find, and Insert, to initialize the structure, retrieve the value for a given key, and insert a key/value pair. These protocols are detailed in the extended version. During initialization, Setup creates a PathORAM and saves an empty node for the root of the AVL tree at a randomly selected position called rootID. Subsequent Find and Insert calls traverse the AVL tree from the root to find or insert a matching node, with each node traversal requiring a separate ORAM access. The ORAM position for a child node is stored at the parent. All accessed nodes are then re-encrypted and mapped to fresh random positions before being stored again at the PathORAM. For insertions, an AVL tree rebalancing process is executed via ORAM read/write accesses.

3.3 Doubly-Oblivious Primitives

The above oblivious primitives assume the client’s memory is protected from the adversary. To provide security in a model where the adversary can observe the client memory accesses, Mishra et al. [83] proposed the notion of doubly-oblivious primitives where access to the client’s memory and instructions is done in an oblivious way too. The importance of such high level of security is clear when considering code executed in TEE, as in this setting even data-oblivious protocols like classic ORAM (e.g., [54, 107]) are no longer secure due to running the client-side routines on the server. Hence, an adversary can easily distinguish different traces of instruction executions by analyzing the instruction access pattern, e.g., monitoring jump locations in the assembly code. Although there are other doubly-oblivious constructions such as CircuitORAM [116] (whose accesses can be implemented by circuits), here we focus on the schemes of [83], as the state of the art. Next, we briefly explain their proposed constructions for array and map data structures (details in the extended version).

**Doubly-Oblivious RAM (DORAM).** Mishra et al. [83] introduced a doubly-oblivious data structure called Oblix, constructed from a doubly-oblivious version of Path-ORAM, i.e., accessing the stash and the client’s memory via oblivious routines, with some additional optimizations. Oblix provides two procedures: **Initialize** and **Access**. In the initialization procedure, it gets a list of $n$ blocks of data and constructs a Path-ORAM tree level-by-level, from the leaf to the root. At each level, it uses oblivious sort and sequential scan to assign the unassigned blocks to that level’s buckets. Access allows the client to read/write a block in the path of leaf $l$. To do that, the client fetches buckets in the path from the root to leaf $l$ and stores their corresponding blocks in the stash. Then, it executes a sequential scan to find the target block and changes its position (and its value for write operations). It then calls **Evict**, to assign blocks to retrieved buckets. It first computes the capacity of each bucket via a sequential scan over the path buckets for each block in the stash. Then, it constructs the buckets of the target path by executing an oblivious sort over the stash blocks to group together blocks with the same bucket id and sends them to the server. The asymptotics of Oblix initialization (with local position map) and access are $O(CN \log N)$ and $O(k(C^2 \log^2 N))$, where $k$ is the number of retrieved paths before calling **Evict** and $C$ is the bucket size.

**Doubly-Oblivious MAP (DOMAP).** [83] also proposed a Doubly-Oblivious Sorted Multimap (DOSM) which supports multiple values for each key and batch sorted accesses. In this work, we do not need these features, so we focus on DOMAPs that support one value per key. We refer to a version of their DOSM limited to the single-value case as Omix. Omix is a DOMAP that uses an AVL-tree on top of Oblix. All stash accesses are performed in an oblivious manner using sequential scans. All other procedures remain the same as the AVL-tree based OMAX of [118] and Path-ORAM accesses are replaced by Oblix. The complexity of **Find/Insert** is $O(C \log^{4} N)$ because OMAP eviction is called after log $N$ path retrievals.

**DORAM and DOMAP Security.** The security of **DORAM** and **DOMAP** [83], is defined in the real/ideal paradigm. An adversary interacts either with the real scheme or with a simulator that only gets the memory size, i.e., $N$, as the initial input. In both cases, the adversary can execute **Initialize** and any number of **Access** (in DORAM) or **Find/Insert** queries (in DOMAP). Furthermore, it can observe the communication channel between the client and server, as well as the access pattern of the client’s and server’s memories. A DORAM/DOMAP scheme is secure if no efficient polynomial-time adversary can distinguish between these two cases with non-negligible probability. I.e., the security definition of DORAM/DOMAP is the same as the security definition of ORAM/OMAP with an additional constraint that enforces the client’s memory accesses to be oblivious too. For the formal definition, we refer readers to [83].

**Opaque.** Opaque [124] is an oblivious distributed data analytics platform. It uses TEE over Apache Spark [11] and provides strong security guarantees for computation integrity and obliviousness. At a high level, it constructs new oblivious SQL operators based on oblivious algorithms (such as oblivious sort and oblivious permutation). In Opaque, the cost of running oblivious queries is mostly affected by the oblivious sort algorithm.

4 OUR DOUBLY-OBLIVIOUS PRIMITIVES

In this section, we propose our doubly-oblivious primitive Omix++. The obliviousness of our approach follows from the fact that all
**Algorithm 1 Omix++ Initialization Procedure**

1: function INITIALIZE([b1]₀, N, baseline)
2: Nodes ← [b1]₀ ∪ Create AVL Nodes from key-value pairs
3: Pad Nodes with dummy blocks to a power of 2
4: Obliviously sort Nodes based on their keys
5: root ← CreateAVLTree(Nodes, 0, Nodes.size-1)
6: Add N–Nodes.size dummy nodes
7: DORAM.INITIALIZE(N, Nodes)
8: return root
9: end function
10: 
11: function CreateAVLTree(Nodes, start, end)
12: if (start < end) return (-1, 0) ▷ (node leaf, node key)
13: mid ← ((start + end) / 2)
14: curRoot ← Nodes[mid]
15: (curRoot.leftChildKey, curRoot.leftChildPos) ← CreateAVLTree(Nodes, start, mid - 1)
16: (curRoot.rightChildKey, curRoot.rightChildPos) ← CreateAVLTree(Nodes, mid + 1, end)
17: set curRoot.pos value using PRF evaluation % N
18: return (curRoot.pos, curRoot.key)
19: end function

distinct operations create indistinguishable memory access traces as can be seen by inspecting the pseudocodes. Below, we provide the high-level idea of our construction and discuss its security and efficiency. For full details and security proof, we refer readers to Appendix D in the extended version.

**4.1 Omix++: New Doubly-Oblivious MAP**

Internally, Omix++ uses Oblix to store nodes of an AVL tree. Each node holds (besides its key, value, and its children’s keys) the PathORAM binary tree leaf positions (pos, childrenPos) for itself and its children. Hence, an Omix++ access consists of multiple Oblix accesses, always starting from the root node and continuing to the maximum AVL-tree height for N nodes. There are two main new features in Omix++: An oblivious initialization process that can be executed directly at the server and an early eviction strategy that makes Omix++ asymptotically and concretely faster than Omix.

**Initialize.** The initialization procedure (Algorithm 1) gets as input an array of data blocks with size n and the maximum number of data blocks Omix++ will maintain (denoted by N). First, it creates an AVL node for each key-value pair after padding them with dummy pairs up to the next power of 2, and obliviously sorts them based on their keys (lines 2-4). In this way, a unique AVL-tree can then be built for them obliviously in a deterministic manner, just by using blocks’ indexes recursively (e.g. the first block will be the leftmost leaf, the second block will be the parent of the first leaf, …, the last block will be the rightmost leaf). Then, it creates the AVL-tree recursively (CreateAVLTree) and assigns each AVL node to a leaf using PRF evaluation (modulo N). CreateAVLTree traverses the AVL-tree using DFS strategy and sets the children keys and positions of each AVL node in the AVL-tree structure. Finally, it creates dummy blocks up to N and runs the Oblix initialization process, using the leaf positions that have been already assigned during the AVL-tree construction (line 17). Note that, unlike the initialization procedure of Oblix that randomly generates positions of data blocks, we need to use the AVL node positions (that are also assigned randomly) in the setup procedure of Oblix so that we can keep the AVL-tree structure. After the Oblix setup, the root node is returned so that future accesses can be bootstrapped.

**Find.** During lookups (Algorithm 2), the client traverses the tree from the root to the maximum height \((1.44 \cdot \log N)\) in order to find the node with the requested key, each time performing an Oblix Access. The major novelty of Omix++ is its eviction strategy. In Omix++, all ORAM accessed blocks during AVL-tree traversal are stored in stash, until one eventual “large” eviction is used to place all of them back at the end of the query. On the other hand, Omix++ calls the Evict procedure one path at a time and as “early” as possible for each path. In other words, Omix++ evicts the fetched ORAM blocks after each Oblix Access (line 5). To do this, we evaluate the random position of the left/right child node (depending on the comparison of the search key) ahead of time and evict the current AVL node with the updated child position. This position is then used at the next iteration as the new position of the retrieved AVL node (lines 6-8). This individual eviction strategy significantly improves the performance of Omix++ compared to Omix, as we show in our experimental evaluation (Sec. 6). The primary reason for this improvement is that by evicting one path at a time we keep the stash size small, which directly affects the performance of oblivious sort which is the bottleneck during evictions for Omix.

**Insert.** The Insert algorithm is similar to Find due to the similarity of these procedures in an AVL tree. It gets a key-value pair, the root node of the AVL tree, and the maximum capacity N. It starts from the root until the node is either found and updated, or created by adding a new AVL leaf node, updating its corresponding parent in the tree path, and storing the new node by an Oblix write. Creating a new node may make the tree unbalanced. Rebalancing is done in the standard way executing left or right rotation depending on the height difference between the children. To do this obliviously
and efficiently we proceed as follows. First, along the traversed AVL path, all “sibling” nodes are also fetched (as they may be necessary for rebalancing) for a total of \(2 \cdot [1.44 \cdot \log N]\) calls to OBLIX access, and fetched nodes are stored in a temporary node stash. The same path is traversed again, this time from leaf to root. At each level, relevant nodes and their parents are extracted from the node stash (via sequential scan for obliviousness) and we check whether rebalancing at that level is necessary. To hide the level and type of rebalancing (left/right/left-left/right-right/left-right/right-left), a “dummy” rebalance is done at each level (via OBLIX access calls).

**Path-Caching Mechanism.** An observant reader may note that a side-effect of our individual path eviction is that during insertions the same nodes are accessed and evicted twice (one in the root-to-leaf traversal and one in the opposite direction). In the TEE setting, data transfer between the enclave and untrusted storage is a slow operation and may introduce considerable overhead. To alleviate the overhead from these duplicate accesses, we propose an intermediate path-cache mechanism that stores paths previously evicted for faster access. Our cache is implemented by a simple non-oblivious tunable map inside the enclave memory. Whenever the enclave needs to fetch a path (during FIND/INSERT), it first checks whether it exists in the cache—if not, it requests it from the untrusted storage. On the other hand, when a path is evicted, the corresponding buckets are written in the cache and can be subsequently fetched without the context-switch overhead. This is particularly helpful for INSERT, where the same nodes are accessed more than once. This cache is iteratively evicted to untrusted storage to ensure it can always fit inside the enclave memory. It is important to note that accessing this path-cache map can be done non-obliviously (hence efficiently) without revealing any extra information to the server. This holds since the specific positions that are accessed only have to do with the corresponding OBLIX leaves and this information is already known to the adversary. As we show in Sec. 6, this optimization improves the performance of Omix++ considerably.

**Eviction Policy Improvement.** As we mentioned in Sec 3, OBLIX executes a nested loop in the eviction procedure to assign each block to its corresponding bucket. We propose an eviction policy that improves the access of OBLIX asymptotically from \(O(C \log^2 N)\) to \(O(C \log N \log^2 \log N)\) and that of Omix++ from \(O(C \log^3 N)\) to \(O(C \log^2 N \log^2 \log N)\). Note that this refers to OBLIX eviction and is independent of the individual eviction for Omix++ we explained above. The high-level idea is to replace the nested loop with two oblivious sorts and a sequential scan. We explain this in more detail with a simple eviction example for a tree with four leaves and bucket size 2 in Figure 3. After fetching the target path of the tree (path from root to leaf 1), storing it in the stash, and updating the target data block, the client first assigns each non-dummy block to the lowest possible level in the stash (step 1 in the figure). Then, the client adds two (equal to the bucket capacity) dummy blocks to the end of the stash (step 2) and obliviously sorts all blocks based on how deep they can be assigned, prioritizing real blocks over dummy ones at each level (step 3). In the next step, it scans all blocks sequentially and tries to construct buckets of blocks based on the capacity of each bucket, and reassigns the overlapped ones to the other non-full buckets in the upper levels (step 4). Finally, it executes another oblivious sort to group together all the blocks of the same bucket (step 5). At this point, the first six blocks (2 blocks for each bucket) create the eviction path and the next two blocks create the new stash with permanent size 2. Although our new eviction strategy improves OBLIX asymptotically, in practice the improvement is small (e.g., \(<8\%\)). Therefore, due to space limitations, we defer the detailed analysis to Appendix C in the extended version.

**Efficiency and Security.** The initialization complexity of Omix++ is \(O(CN \log^3 N)\), since it requires two sequential scans, an oblivious sort, an OBLIX initialization (with \(O(CN \log^3 N)\) cost), and the recursive process for building the AVL-tree \(O(N)\) since it iterates over all AVL nodes. The INSERT and FIND asymptotics are \(O(C \log^2 N \log^2 \log N)\), since they need \(O(N)\) OBLIX accesses, including padding (using our optimized OBLIX eviction). For comparison, the corresponding time for Omix is \(O(C \log^4 N)\).

### 5 OBLIVIOUS GRAPH PROCESSING

Our main objective is to design a system that handles graph queries in an oblivious manner, i.e., without leaking the structure of the graph (or any other meaningful information about the graph besides the number of vertices and edges). Achieving obliviousness against an adversary that can observe the memory access pattern, as is the case with a system relying on TEE, is tricky as this entails two types of memory accesses: (i) *data-access*, i.e., accessing a graph vertex/edge, and (ii) *instruction-access*, i.e., fetching the next CPU instruction to be executed. Eliminating the leakage from both of them is crucial, as the following “toy” examples highlight.

Consider an algorithm that performs a scan of an array of \(n\) integers (stored sequentially in memory) incrementing a counter each time it sees an odd number and decrementing it each time it
Therefore, the adversary can correlate the conditional of different
graphs. Even when the code is encrypted (as is the case with TEE),
the position of the fetched instruction is still harmful information
because the execution trace of the above simple algorithm leads to
a conditional evaluation and a jump (based on the condition result).
Therefore, the adversary can correlate the conditional of different
arrays with each other and identify that specific indexes
of the array have similar properties. In other words, an adversary
that sees \( x \) accesses to one instruction and \( n - x \) to another knows
the array contains \( x \) odd and \( n - x \) even numbers, or vice versa.

On the other hand, leakage from data access is also harmful. Con-
sidering a BFS/DFS traversal on a graph (and even if instructions-
access leakage is ignored), the number of times the memory location
of a certain vertex is accessed is related to its degree.

Based on these two types of leakage, to achieve our goal of obliv-
iouss graph processing we first store the graph using our doubly-
oblivious primitives and then propose graph query algorithms that
have a deterministic sequence of instruction execution and are
independent of the graph data. These two techniques are comple-
mentary; the first eliminates data-access leakage and the second
eliminates instruction-access leakage. We implemented this ap-
proach with OMIX++ based on hardware enclaves to store and
query the graph and we call the resulting system GraphOS. Figure 4
depicts the architecture of our system. The first step involves the
user uploading the input graph in encrypted form to the server.
Next, the user begins the GraphOS initialization procedure to set
up the hardware enclave and create the required doubly-oblivious
data structure indexes. Once initialization is complete, the user
can securely execute graph queries by interacting with GraphOS.
Below, we first explain the architecture and basic operations
of GraphOS. Then, we describe our algorithms for four fundamental
graph queries in Sec 5.2. For BFS/DFS and MST we provide our own
efficient versions of these algorithms that do not have instruction-
access leakage. For SSSP, we rely on the algorithm of [78].

5.1 GraphOS—Architecture and API
GraphOS uses OMIX++ to store the graph. It is initialized (in time
\( O(|E| + |V|) \)) to contain the following key-value pairs:

(1) For each vertex \( v \), we store an entry with key \( ("V"|v) \) and value
\((deg_{out}, deg_{in})\), where \( "V" \) is a label showing this entry is for a
vertex, \( v \) is the vertex id, and \( (deg_{out}, deg_{in}) \) are its degrees.

(2) For each edge from vertex \( v_{init} \) to vertex \( v_{term} \) with weight \( w \),
we store three key-value pairs:

- This pair has key \( ("EOut"|v_{init}, cnt) \) and value \((v_{term}, w)\) where
subsection {"EOut" is a label showing this is an outgoing edge, and \( cnt \)
is the index of this edge in the outgoing edge set of \( v_{init} \).
- This pair has key \( ("EIn"|v_{term}, cnt) \) and value \((v_{init}, w)\) where
"EIn" is a label showing this is an outgoing edge, and \( cnt \) is the
index of this edge in the incoming edge set of \( v_{term} \).
- This pair has key \( ("E", v_{init}, v_{term}) \) and value \((w, cnt_{init}, cnt_{term})\),
where "E" is a label showing this is an edge.

This structure allows GraphOS to efficiently extract information
in comparison to other methods, such (e.g., adjacency list).
Specifically, it can determine the degree of each vertex with a single
Omix++ lookup (using the \( ("V"|v) \) key) rather than requiring a
sequential scan over all edges. Additionally, adding a vertex or edge
incurs no extra overhead and only requires a constant number
of OMIX++ accesses. Moreover, a vertex can be easily removed by
extracting its degree and removing its edges. This approach improves
efficiency in large graphs with a small average degree by avoiding
the need for unnecessary sequential scans over a large list of edges.
Now, we present the basic procedures of GraphOS. We provide the
detailed pseudocodes in Appendix E in the extended version.

Setup. To setup GraphOS for a graph \((V, E)\) the client encrypts
it, establishes a secure channel with TEE, attests the GraphOS en-
clave to ensure the authenticity of the code, and runs the enclave.
Then, it sends the decryption key and other parameters needed
for the setup of OMIX++. We do not assume the graph is provided
in a specific key-value format, so TEE must handle this. First, it
initializes a temporary OMIX++ only with vertex entries. It iter-
ates over the list of edges, each time retrieving from OMIX++ its
source and target vertices, computing the in/out-degree of each
vertex, and building the key-value pairs needed for edges (as ex-
plained above). Note that doubly-oblivious primitives (OMIX++) is
necessary; otherwise, setup would leak the structure of the graph.
Finally, TEE discards the temporary DOMAP and runs the Initial-
ization procedure of OMIX++ for all created key-value pairs. Setup
performs a loop over all edges and corresponding OMIX++ Inserts
\((O(|C|\log^2 |E| \log^2 |E|)\) assuming \(|E| \geq |V|\)). Hence its complex-
ity is \( O(|C|\log^2 |E|) \), dominated by the OMIX++ initialization.

We can add some auxiliary key-value pairs to improve specific
graph algorithms’ execution time. As per Sec 4, OMIX++ insertion
is slower than lookup, due to re-balancing. Precomputing and storing
some keys during setup “converts” future OMIX++ insertions to
faster OMIX++ lookup-and-set. E.g., in the BFS algorithm, we know
ahead of time that all vertices will be visited. Indeed, we can create
a key-value pair with a dummy value for each of them and use it
to emulate queue operations by just updating their values.

Lookup Queries. GraphOS provides oblivious lookup queries via
OMIX++. It supports the following: (i) find a vertex/edge, (ii) find an
degree, and (iii) find the in/out-degree of a vertex. All these
queries only need one OMIX++ query. For example, executing a
lookup query with key \( "V"|v_i \) gives the degree of node \( v_i \).
We now explain how four well-known graph algorithms are run in GraphOS; we focus on making the code execute oblivious versions that is asymptotically more efficient. However, our evaluation in Sec 6 shows that, in TEE it outperforms our version only for very dense graphs. We highlight that the required modifications in the plaintext graph algorithms are relatively small, but this is desired in oblivious algorithms since it can lead to comparably small overhead between oblivious and non-oblivious algorithms. BFS/DFS. These two queries are graph traversals that load and unload vertices to and from a queue and a stack, respectively. Oblivious versions of these data structures can be emulated in a standard manner, using a DOMAP and two index counters for the first and last item. However, textbook implementations of them still have leakage due to instruction accesses. E.g., BFS runs a double-loop over the vertices where the internal loop is over the number of neighbors each time; each time the code exits the internal loop, a different (dequeu) instruction is executed. To avoid this leakage, we ensure our algorithm runs in a single loop using the loop-coalescing technique [78] and oblivious Osel/Ocmp operators. In particular, we partition the nested loop into chunks of blocks each of which corresponds to a branch. The number of execution times for each block is used for a bound for the innermost loop that contains that block and their sum represents the total number of iterations in the single-loop version. Next, we convert the nested loop into a single loop and use an extra state variable for each block to simulate the inner loop for each code block. Furthermore, the end branch statements will be converted to state change for these variables. Minimum Spanning Tree. Our MST algorithm is based on the classic Kruskal [73] where edges are sorted based on their weights. Instead of running $|E|$ DOMAP queries, we do this efficiently by obliviously sorting the edges using a copy of DOMAP blocks (to avoid data corruption in DOMAP) which are then fetched sequentially (ElList). After this, we assign each vertex to a separate tree (in MST sub-trees) and execute an oblivious version of Kruskal’s algorithm, following a similar approach as in BFS/DFS above. At a high level, checking of loop creation for the new edge in MST (which is done using a recursive function in the textbook version), is implemented by keeping the root of the subtrees in Omix++. Single-Source Shortest Paths. For SSSP, we implement MinHeap-based Dijkstra [43] with the oblivious MinHeap of Shi [100] and apply the optimization of [78] to avoid weight update operations. We combined [100] with Omix (instead of PathORAM) and made its operations (e.g., Insert and ExtractMin) doubly oblivious to implement a doubly-oblivious MinHeap. To eliminate instruction-access leakage, we use [78] with loop-coalescing optimization. Efficiency and Privacy. The complexity of BFS/DFS and SSSP in GraphOS is $O(C log^2 |V| log^2 log|E|)$ while for MST it is $O(Clog^2 |E| log^3 |V| log^2 log|E|)$ assuming $|E| \geq |V|$. For comparison, opaque’s complexity for BFS/DFS and MST is $O(C(V^2 |E| log^2 log|E|), O(C|E||V|^2 log^2 log|V|)$ and for SSSP is $O(C|V|^3 log^2 |V|)$ respectively, i.e., GraphOS improves the best prior results. Due to the use of Omix++ and oblivious operators, GraphOS only leaks $|V|$ and edges $|E|$ when executing the above algorithms. It hides data access pattern leakage by using doubly-oblivious data structures and instruction access pattern by converting the algorithms to their doubly-oblivious versions. These doubly-oblivious algorithms use oblivious sort (e.g., Bitonic sort [15]), oblivious operators such as Osel and Ocmp to hide conditions, dummy operations to hide loops. Implementing other Graph Algorithms. In Sec 2, we explained that “Obfuscuro-like” approaches [1] can make any code double-oblivious—pairing this with Omix++ would improve its efficiency. Besides, we now provide general guidelines for implementing other graph algorithms in GraphOS; we focus on making the code execution trace deterministic, utilizing Omix++ and doubly-oblivious algorithms, to achieve more efficient graph query solutions. Balance conditions: We need to ensure the same number of Omix++ accesses are executed in all branches of any condition. This is done by adding dummy read/write operations at the end of each branch, and/or making extra dummy Omix++ accesses. Besides, conditions needs to be implemented using oblivious operators (see Sec 3). Balance loops: For algorithms that perform different types of operations in each loop, we need to pad the number of loop iterations to an upper bound. Also, for nested loops (when the overall complexity of all these queries is equal to the complexity of Omix++ Find because they execute a single Omix++ operation. Update. To add vertex $v$, GraphOS adds entry (“V”$\mid |v|$) with value $(0,0)$ to Omix++. To add edge $(v_{\text{init}}, v_{\text{term}}, w)$, it first fetches the current number of incoming edges to $v_{\text{term}}$ (denoted by $\text{init}_{v_{\text{term}}}$) and the number of outgoing edges from $v_{\text{init}}$ (denoted by $\text{out}_{v_{\text{init}}}$). Then, it increments the corresponding counters and writes the new values back and the new edge key-value pairs in Omix++. To remove edge $(v_{\text{init}}, v_{\text{term}})$, GraphOS has to remove the corresponding data from $v_{\text{init}}$ and $v_{\text{term}}$. It extracts the related counters of the target edge by fetching the edge counters of the initial and terminal vertices ($\text{cnt}_{v_{\text{init}}}$ and $\text{cnt}_{v_{\text{term}}}$) using key (“E”, $v_{\text{init}}$, $v_{\text{term}}$) and removes their entries from DOMAP. This invalidates the counter indexes in the two lists. We fix this by “pruning” removed entries in Omix++ (swapping the counter value of the last edge and the deleted edge, see [53]). To remove vertex $v$, we first delete all incoming and outgoing edge counters with key (“V”$\mid |v|$). Then, we fetch all vertices connected to $v$ via edges, and we delete said edges via the process explained above. This inherently reveals the degree of the deleted vertex, unless one is willing to pad with $|V|$ dummy accesses.

Each of these queries needs a different number of Omix++ accesses (e.g., adding a vertex only needs one Insert while adding an edge needs two Find and five Insert). We can eliminate this leakage by padding all queries to the maximum needed Omix++ queries. The overall complexity of adding a vertex/edge and removing an edge needs two Find and five Insert (because they execute a single Find because they execute a single Insert and 5 Update). However, textbook implementations of them still have leakage due to instruction accesses. E.g., BFS runs a double-loop over the vertices where the internal loop is over the number of neighbors each time; each time the code exits the internal loop, a different (dequeu) instruction is executed. To avoid this leakage, we ensure our algorithm runs in a single loop using the loop-coalescing technique [78] and oblivious Osel/Ocmp operators. In particular, we partition the nested loop into chunks of blocks each of which corresponds to a branch. The number of execution times for each block is used for a bound for the innermost loop that contains that block and their sum represents the total number of iterations in the single-loop version. Next, we convert the nested loop into a single loop and use an extra state variable for each block to simulate the inner loop for each code block. Furthermore, the end branch statements will be converted to state change for these variables.

5.2 Graph Queries

We now explain how four well-known graph algorithms are run in GraphOS. In particular, we consider breadth/depth-first traversal, minimum spanning tree, and single-source shortest paths. For the first three, we propose our own oblivious versions that avoid instruction-access leakage. This is done by ensuring fixed deterministic sequences of operations, entirely independent of the actual data values. For the last one, we use the algorithm of [78]. In all cases, to eliminate data-access leakage and achieve oblivious query processing that only reveals $|V|$ and $|E|$, all graph accesses are via Omix++. We note that [78] proposed an optimized oblivious DFS version that is asymptotically more efficient. However, our evaluation in Sec 6 shows that, in TEE it outperforms our version only for very dense graphs. We highlight that the required modifications in the plaintext graph algorithms are relatively small, but this is desired in oblivious algorithms since it can lead to comparably small overhead between oblivious and non-oblivious algorithms.
inner-loop execution depends on the outer-loop, e.g., BFS), the loop-coalescing technique [78], i.e., rewriting the code as a single loop, can improve efficiency. Use of OMIX++ or oblivious data accesses: Input data and intermediate results must either be loaded in OMIX++ or accessed obliviously (e.g., via a sequential scan).

6 EXPERIMENTAL EVALUATION

We evaluate the performance of OMIX++ and GraphOS and compare it with state-of-art competitors. In our experiments, for OMIX++, we consider variable synthetic datasets with total size between $2^8 - 2^{24}$ and evaluate it in three real-world applications. For GraphOS, we consider variable random synthetic graphs with size $(|V| + |E|)$ between $2^{15} - 2^{19}$. Note that the security property of oblivious graph processing means that performance does not depend on the structure of the graph (just $|V|$ and $|E|$). That’s the reason why we do not need to repeat our experiments for real datasets. We evaluate GraphOS and Opaqe for BFS/DFS, MST, and SSSP on three different graphs with variable denseness: (i) very dense $(|E| \approx |V|^2)$, (ii) sparse $(|V| = 0.13|E|)$, and (iii) very sparse $(|V| = 0.8|E|)$. Although we measured the performance of GraphOS over all our test graph sizes, we ignored Opaqe execution time for sizes which would take several days/months. In addition to Opaqe, we compared GraphOS execution time with oblivious code/data retrieval methods based on DOMAP such as Obfuscuro [1], provided a comparison between GraphOS and Liu et al.’s [78] DFS algorithm, and evaluated a distributed version of GraphOS.

Experimental Setup. We use C++11, Intel-SGXv1 (SDK v2.4), and SGX OpenSSL extension [105] for cryptographic operations in our experiments. We ran our experiments on a machine with an eight-core Intel Xeon E-2174G 3.8GHz processor with SGX support (AES-NI enabled), 64GB RAM, 1TB SSD, and Ubuntu 16.04 LTS. We limited the enclave’s trusted memory to 94MB. Unless otherwise noted, the DORAM block size is set to 128 bytes and $C = 4$ blocks/bucket. We report the average of 10 executions (standard deviation $\sigma < 2\%$ across all experiments). In all experiments, first we warm up DORAM/DOMAP data structures with 10K dummy operations to reach the steady state of their performance. Furthermore, in all setup experiments, we included remote attestation time (excluding Intel server communication) which takes less than 50ms.

Implementation. We implemented OMIX++ as well as OMIX for comparison. Since the code of [83] is not “fully” doubly oblivious (specifically the tree rotation needed for their insert operation is implemented non-obliviously), we had to write our own implementation. For oblivious graph processing, we implemented GraphOS using OMIX++ and our SGX-based implementation of Shi’s Min-Heap [100]. The latter operates in the client-server model, therefore we replaced its ORAM with Oblix. In addition to this, we made all its client-side operations (e.g., insert and extract-min) doubly oblivious. For GraphOS, we applied additional optimizations to the graph query execution process. E.g., for BFS/DFS queries, since we know that all vertices will be placed in the queue/stack eventually, we put their corresponding key-values (where the value is set to NULL) in the initial key-value list of GraphOS setup. This removes the need for lots of insert operations in the query execution. Such an optimization lead to 40% improvement in BFS/DFS execution time because we have removed the need for complex oblivious rotation. For Opaqe experiments, we extended its released code [123] to support the necessary graph operations and implemented the graph algorithms discussed in Sec 5.2. In particular, since Opaqe does not support some of the needed operators such as encrypted outer joins and encrypted union, we implemented their equivalent operations with the supported operators. All our implementations are publicly available in [51]. They are the first open-source doubly oblivious libraries and may find use in other applications.

6.1 Doubly-Oblivious Data Structure (DOMAP)

First, we examine the performance of our PathORAM-based doubly-oblivious data structure. Figure 5(a) shows the setup time of OMIX++ and OMIX. In OMIX++, the main overhead is the Oblix initialization—the AVL tree construction takes a small portion of the time, e.g., it takes 983s to initialize Opaqe with size $2^{20}$ while the AVL tree only takes 31s. Recall that Omix does not provide an explicit oblivious initialization, other than the “naive” process of Opaqe setup, followed by inserting key-value pairs one-by-one. Throughout all our experiments, OMIX++ setup is 1.5–11x faster than OMIX.

Figure 5(b), (c) show the INSERT/FIND execution times for variable DOMAP sizes. We separated these two experiments due to their different number of memory accesses (because of AVL balancing). Our evaluation shows that OMIX++ clearly outperforms OMIX. This is due to (i) the individual eviction policy and (ii) the path-caching

---

3 Alternatively, DOMAP can potentially be built from other ORAMs. However, ORAM schemes that need periodic rebuilds (e.g., hierarchical solutions [54]) are inherently less practical than our OMIX++ when run in TEE, due to the high cost of making the rebuild doubly oblivious. Moreover, deamortization would make this even more expensive as it needs maintaining/accessing multiple ORAM copies, and executing polylogarithmically many steps each time.
Table 1: GraphOS and Opaque basic graph query benchmark for two different graph sizes (RA denotes remote attestation).

<table>
<thead>
<tr>
<th>Operation</th>
<th>System</th>
<th>Time (seconds)</th>
<th>Size (2^{12}/2^{18})</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup+RA</td>
<td>GraphOS</td>
<td>99 / 19566</td>
<td>2^{12}</td>
</tr>
<tr>
<td></td>
<td>Opaque</td>
<td>0.9 / 13</td>
<td>2^{18}</td>
</tr>
<tr>
<td>look-up vertex/edge</td>
<td>GraphOS</td>
<td>0.01 / 0.02</td>
<td>2^{12}</td>
</tr>
<tr>
<td></td>
<td>Opaque</td>
<td>1 / 1.9</td>
<td>2^{18}</td>
</tr>
<tr>
<td>add vertex</td>
<td>GraphOS</td>
<td>0.02 / 0.06</td>
<td>2^{12}</td>
</tr>
<tr>
<td></td>
<td>Opaque</td>
<td>0.8 / 8.2</td>
<td>2^{18}</td>
</tr>
<tr>
<td>add edge</td>
<td>GraphOS</td>
<td>0.3 / 0.6</td>
<td>2^{12}</td>
</tr>
<tr>
<td></td>
<td>Opaque</td>
<td>0.8 / 8.2</td>
<td>2^{18}</td>
</tr>
<tr>
<td>remove vertex</td>
<td>GraphOS</td>
<td>0.07 / 0.15</td>
<td>2^{12}</td>
</tr>
<tr>
<td></td>
<td>Opaque</td>
<td>0.7 / 4.4</td>
<td>2^{18}</td>
</tr>
<tr>
<td>remove edge</td>
<td>GraphOS</td>
<td>0.3 / 0.7</td>
<td>2^{12}</td>
</tr>
<tr>
<td></td>
<td>Opaque</td>
<td>0.7 / 4.4</td>
<td>2^{18}</td>
</tr>
</tbody>
</table>

As shown in the Table 1, universe operations in GraphOS is 2.6–13.6 times faster than Opaque++. E.g., for graph size 2^{18}, the improvement of using GraphOS over Opaque++ is 25x faster than Opaque. To separately measure the effect of these on Omix++, we disabled the cache mechanism in a new experiment (denoted by Omix++-NC in Figure 5(b,c)). This shows the cache is more impactful for small DOMAP sizes. Besides, the early eviction strategy led to proper improvements for larger DOMAP. E.g., for 2^{24}, Omix++-NC insert is 19.4× faster than Omix and Omix++ is 1.7× faster than Omix++-NC.

6.2 Basic Graph Operations

We report the performance of basic operations (setup, searching/adding/removing a vertex/edge) in GraphOS and Opaque in Table 1. Setup Time. Overall, Opaque has a faster setup than GraphOS. E.g., it takes 13s to setup a graph with size 2^{10} for Opaque but 19566s for GraphOS. This would come as no surprise since GraphOS has a poly-logarithmic search time while Opaque++ as it only requires a DOMAP access to query execution time (for BFS/DFS, MST, etc.).

Search/Update Times. Accessing a vertex/edge in GraphOS is significantly faster (95–150×) than Opaque as it only requires a DOMAP access (poly-logarithmic search time) while Opaque must execute a sequential scan over the whole vertex/edge encrypted table for obliviousness. E.g., for graph size 2^{18} GraphOS requires 0.02s and Opaque 1.9s—clearly this gap increases for bigger graphs. Similar observations hold for updates, i.e., GraphOS is 2.6–13.6× faster in adding/removing an edge and 2.4–136× faster in removing a vertex.

6.3 Graph Query Evaluation

BFS/DFS. Figure 6(a) shows the execution time of BFS/DFS for variable graph sizes |V| + |E|. As expected, there is a notable gap in performance between the two systems, e.g., Opaque takes more than 7.5h to run BFS/DFS on a very sparse graph (|V| = 0.8|E|) with size 1024, while GraphOS runs in 67s. For graph sizes 2^{18}–2^{25}, GraphOS is 6–410× faster than Opaque. Experiments with bigger sizes for Opaque were omitted as they would require several days or weeks—it is clear that GraphOS would become orders of magnitude faster than Opaque.
faster. This agrees with its achieved asymptotic improvement of \(O(V^2/\log^2 \log E)\) over \text{Opaqe}. Recall that this improvement in performance is accomplished by strictly less leakage. GraphOS only reveals \(|V|\) and \(|E|\), whereas \text{Opaqe} reveals the number of vertexes at each distance from the source, unless it uses worst-case padding, making it up to five orders of magnitude slower than GraphOS.

**Minimum Spanning Tree (Kruskal).** Figure 6(b) shows the execution time for MST. The comparison between the two systems has similar characteristics as for BFS/DFS. GraphOS is 1.4–86× faster in graphs with size \(2^8 \sim 2^{14}\) (e.g., it takes 212s for graph size 512 while \text{Opaqe} takes 5h). It is clear that the gap can again increase arbitrarily, as also indicated by the asymptotic difference. Unlike the case for BFS/DFS, both systems only reveal \(|V|\) and \(|E|\).

**Single Source Shortest Path (Dijkstra).** Figure 6(c) shows the execution time of SSSP. Similar to the above cases, GraphOS outperforms \text{Opaqe} in executing Dijkstra. E.g., GraphOS is 1–22× faster for sizes up to \(2^{17}\). Furthermore, GraphOS only reveals \(|V|\) and \(|E|\), whereas \text{Opaqe} trivially reveals the number of neighbours of each vertex (again, eliminating this leakage of \text{Opaqe} would require tremendously expensive worst-case loop-padding (\(|V|\)).

### 6.4 Additional Experiments

**Variable block-size DOMAP.** To evaluate the effect of block size in \text{OMix++}, we measured the Find/Insert time varying the block size between 128–2048 bytes while fixing the size to \(2^8\) (Figure 5(d)). For fairness, we disabled the path-cache of \text{OMix++}, as this can be used in both schemes. As shown, \text{OMix++} clearly outperforms \text{OMix} for all block sizes, both for Insert (I) and Find (F). Concretely \text{OMix++} with disabled path-cache is \(1.9–10.6\times\) faster in Find and \(2.8–10.6\times\) faster in Insert, across all block sizes. Since path-cache is disabled, this is solely due to our individual eviction strategy.

**Oblivious vs. textbook graph algorithms.** Our graph algorithms have deterministic execution traces at the cost of additional "dummy" operations. To measure this overhead, we compared them with running their "textbook" versions, replacing data accesses with DOMAP ones in both cases. For BFS/DFS the overhead for our tested graphs is \(3.5\times–4.98\times\). This follows directly from the pseudocode: textbook BFS/DFS makes \(2|V| + |E|\) DOMAP accesses, whereas ours makes \(5(|V| + |E|)\). For dense graphs this is close to \(5\times\), whereas for very sparse ones it is close to \(2.5\times\). The gap for our MST is \(1.2\times–8.5\times\). As a point of comparison, Obfuscuro [1] eliminates leakage from instruction accesses by loading code in doubly-oblivious storage and reports slowdowns of \(16–231\times\), for simpler algorithms.

**Comparison with the DFS of [78].** Liu et al. [78] proposed a DFS with deterministic execution for MPC applications, optimized for dense graphs. Although it is more efficient asymptotically, our evaluation in the TEE setting, and compared it with our DFS (Figure 8), shows that [78] is faster only for very dense graphs (\(0.9–2.5\times\)). For more sparse graphs, ours is faster \(0.8–374\times\) Increasingly so for larger sizes, due to fewer untrusted memory accesses.

**Distributed GraphOS.** We also tested the performance of GraphOS implemented in a distributed manner. Due to space limitations, the details can be found in Appendix F in the extended version. Our experimental results show that distributed GraphOS can outperform (an idealized distributed version of) \text{Opaqe} for BFS and SSSP.

**Integrating \text{Opaqe} and GraphOS.** We also evaluated an "integrated" approach of \text{Opaqe} with GraphOS, following a recent trend from the database community which combines in one system the benefits of relational and graph databases (e.g., [120]). We store the graph in \text{Opaqe} in two relational encrypted tables for vertices and edges, and we execute complex graph queries by initializing GraphOS on-the-fly and running these queries with it to minimize leakage. Notably, this approach outperforms \text{Opaqe} and achieves very similar speed-ups with those presented in Sec 6.3 for BFS (2–161×), MST (1–42×), and SSSP (0.8–9×). E.g., for a graph of size \(2^{12}\) running BFS, MST, and SSSP takes \(0.9 + 99 + 368 = 468s\), \(0.9 + 99 + 4386 = 4486s\), and \(0.9 + 99 + 356 = 456s\) while in \text{Opaqe} it takes \(0.9 + 37328 = 37329s\), \(0.9 + 6429 = 6430s\), and \(0.9 + 1462 = 1463s\), respectively (0.9s is for \text{Opaqe} setup and 99s is for GraphOS setup).

### 7 CONCLUSION

We proposed GraphOS, a system for oblivious graph processing based on trusted hardware. It eliminates leakage from memory accesses for graph data via doubly-oblivious data structures and for instruction fetching via algorithms that have data-independent, fixed execution trace. Compared to previous works, GraphOS achieves less leakage (only the number of edges and vertexes in the graph, and for each query its type and response size). At the same time, it outperforms previous solutions both concretely and asymptotically. That said, although GraphOS is the fastest existing system for oblivious graph processing, it is still far from practical (the non-private version of these algorithms may take \(<1s\) to run, whereas GraphOS may take several hours). We hope this work can motivate further research and new results in this area, whereas our doubly-oblivious primitive may find other applications beyond graphs.


Algorithm 4 Improved Oblix Eviction AssignBlocksToBuckets

1: function AssignBlocksToBuckets(Allblocks, i, cnt, level, curBucketID, N)
2:    block ← Allblocks[i]
3:    cond1 = Osel((cnt − (log N + 1 − level)) ≥ C, 1, 0)
4:    cond2 = Osel((block, bucketID = curBucketID), 1, 0)
5:    cond3 = Osel((block is dummy or cond ≥ C(log N + 1)), 1, 0)
6:    tmpBucketID = bucket id of level (log N + 1)−[(cnt/C)] in path l
7:    nextBucketID = bucket id of level + 1 in path l
8:    block.bucketID = Osel((cond1 & cond2 & cond3), Osel(tmpBucketID, block.bucketID))
9:    block.bucketID = Osel((cond1 & cond2 & cond3), Osel(nextBucketID, block.bucketID))
10:   cnt = Osel((cond1 & cond2 & cond3), cnt + 1, cnt)
11:   cond1 = Osel((cond1 & cond2), cnt + 1, cnt)
12:   level = Osel((cond1 & cond2), level + 1, level)
13:   i = Osel((cond1 & cond2), i + 1, i)
14:   curBucketID = Osel((cond1 & cond2), nextBucketID, curBucketID)
15: end function

Algorithm 3 Improved Oblix Eviction Procedure

1: function Evict(l)
2:    Pblocks ← Download blocks of path l
3:    Sblocks ← Download all the blocks from the stash
4:    Allblocks ← Pblocks ∪ Sblocks
5:    Assign non-dummy blocks to lowest level in path l
6:    Add C dummy blocks to each level and set their level
7:    Obliviously sort Allblocks by their assigned level
8:    giving priority to the non-dummy blocks in each level
9:    level = log N + 1
10:   curBucketID = bucket id of the lowest level in path l
11:   for i = 1 to |Allblocks| do
12:       Assign Allblocks[i] to curBucketID if it is not full,
13:      Otherwise:
14:         If Allblocks[i] is non-dummy, assign it to its closest
15:            non-full bucket in the upper levels.
16:         If Allblocks[i] is dummy, mark it as ∞
17:      Decrement level and update curBucketID
18:   end for
19:   Perform a sequential scan over Allblocks and mark 0
20:      all the blocks with level ∞ and 1 the remaining ones
21:   Perform an oblivious sort on Allblocks
22:   for i = 1 to (log N + 1) do
23:      currentBucket = Allblocks[C · i · ... · C · (i + 1)]
24:      Store currentBucket at the server in level l
25:   end for
26:   Store remaining blocks in the stash up to its capacity
27: end function

A OBLIVIOUS DATA STRUCTURES

In Path-ORAM [107], the server maintains a full binary tree where each node stores a bucket of encrypted blocks (typically 4). The client maintains two data structures: (i) a position map that stores the mapping of each data block to a leaf number in the tree, (ii) a stash that contains temporary/overflowed blocks. Whenever the client wants to access a block, it extracts the block’s leaf number from the position map, retrieves the corresponding path from the server, finds the target block and re-assigns it to a random leaf, and writes back the path with fresh encryptions. The cost of Path-ORAM access (assuming recursive storage) is $O(C \log^2 N)\omega(1)^4$ where $N$ is the total number of blocks and $C$ is the bucket size. The term $\omega(1)$ is related to the stash storage. However, to provide simplicity in the asymptotics, we removed it from the paper’s asymptotics.

Below, we explain Path-ORAM API and explain how the client initializes/accesses data:

- $(\sigma, T) \leftarrow \text{Initialize}(\lambda, N)$: Given a security parameter $\lambda$ and memory size $N$, the client initializes a binary tree $T$ such that it contains at least $N$ blocks. Each block stores the encryption of target data or a dummy value under a secret key $sk$ selected by the client. A position map $M$ with a size equal to the number of binary tree nodes is initialized (we denote the number of leaves by $L$). Each encrypted block is assigned to a random leaf number $(between 1 to L)$ and this mapping is stored in $M$. This assignment enforces the structure to store each block in the blocks within the path from the given leaf to the root of the tree.

- $(\sigma, T) \leftarrow \text{Access}(r/w, y, null/vol, M, S, sk; T)$: To read (denoted by $r$) the block corresponding to the given index $y$, current state of the position map $M$, stash $S$, and tree $T$, the client searches the stash. If it was not found there, asks the server to send back the blocks corresponding to the path extracted from leaf $M[y]$. Then, the client decrypts the blocks, extracts the block with index $y$, and chooses a new random position for the block. Finally, it updates $M$ and calls Evict procedure.

- $(\sigma, S', T') \leftarrow \text{Evict}(S, M[y], sk; T)$: Given the current state of the stash $S$ and the leaf number $M[y]$, the client constructs the blocks of the retrieved path such that each block is stored as deep as possible in the path from the root to leaf based on its leaf position. The evicted blocks will be sent to the server who updates $T$.

Oblivious MAP (OMAP). An oblivious MAP is a privacy-preserving version of a map data structure. We focus on the construction proposed by Wang et al. [118] which is based on maintaining an AVL-tree inside a Path-ORAM. Each node of the AVL tree contains $(id, data, pos, childrenPos)$ where $id$ is the key of the mapping, $data$ is the value of the mapping, $pos$ is node’s leaf number in Path-ORAM tree, and $childrenPos$ contains the leaf numbers of node’s children in the AVL-tree. Using the above structure, the client does not need to store the Path-ORAM position map; it just keeps the position of the AVL root. An OMAP provides three procedures: (i) $\text{Initialize}$,
In the first step, the client downloads all blocks and are re-encrypted freshly before being stored on the server. In this section, we provide the detailed pseudocode of OBLIX algorithms. For more details, we refer readers to [83].

B OBLIX ALGORITHMS

In this section, we provide a brief description of OBLIX algorithms. For more details, we refer readers to [83].

- **initialize**($N, \lfloor b_i \rfloor_i^2$): OBLIX proposes an initialization mechanism that gets the maximum number of blocks $N$ and an initial list of $n$ blocks of data $\lfloor b_i \rfloor_i$. It constructs a Path-ORAM tree such that each node stores a bucket with a constant number of $\lfloor b_i \rfloor_i$ blocks. It uses a level-by-level approach from the bottom to the top of the tree. At each level: (i) it obliviously sorts all $\lfloor b_i \rfloor_i$ based on their leaf positions, (ii) sequentially scans the blocks to compute the capacity of each bucket and assign the (unassigned) blocks to the (non-full) buckets, and (iii) it obliviously sorts the blocks based on their assigned bucket to put them in the buckets.

- **bl ← Access($l, y$)**: To read/write the block with index $y$ in leaf $l$, the client fetches buckets in the path from the root to leaf $l$ and stores blocks of these buckets in the stash. Then, it executes a sequential scan to find the block ($bl$) with index $y$, changes its position (and its value if it is a write operation), and calls Evict procedure. Finally, it outputs $bl$.

- **Evict($l$)**: To evict blocks from the stash to path $l$, the client has to assign stash blocks to the buckets. To do that, it computes the capacity of each bucket and assigns each block to the deepest non-full bucket by performing a sequential scan over the path buckets for each block in the stash. After bucket assignment, it constructs the buckets of path $l$ by executing an oblivious sort over the stash to group together all blocks with the same bucket id. Finally, it executes a sequential scan to send the buckets to the server.

C IMPROVED OBLIX EVICTION

Algorithm 3 shows the improved version of the OBLIX eviction which gets the eviction path $l$ as input and returns the new encrypted ORAM buckets along the path. Now, we explain the steps of this procedure in more detail.

**Initial level assignment.** In the first step, the client downloads all the blocks of path $l$ and stores them together with the blocks of the stash in Allblocks set. The invariant of Path-ORAM is that all evicted blocks must be "pushed" as low in the path as possible. Hence, the client first assigns each non-dummy block of Allblocks to the lowest possible level in path $l$ via a sequential scan and adds $C$ dummy blocks for each level (line 6). Next, Evict obliviously sorts Allblocks based on how deep they can be assigned, prioritizing real blocks over dummy ones at each level. However, blocks cannot yet be placed in their assigned buckets, since $> C$ of them may have been assigned to the same bucket, causing an overflow.

**Correcting the assignment.** To avoid the overflow, we start filling the buckets in a bottom-up fashion. If a bucket in level $l$ becomes full and more real blocks have been assigned to it, we re-assign the remaining real blocks to the upper levels. Likewise, if a bucket is not full, i.e., we have assigned all the real blocks for this level as well as all the real blocks from the lower levels, dummy blocks are used to fill it up. Dummy blocks that are not used in a specific level, have their level set to $\infty$ so that they can be discarded later (the detailed pseudocode of this step is provided in Algorithm 4).

**Bucket construction.** Finally, the algorithm executes another sequential scan to mark all $\infty$ blocks and an oblivious sort that groups all the blocks of the same bucket. The actual buckets can now be constructed via a final sequential scan. All that remains is to remove extra dummy blocks, i.e., keep only the first elements up to the fixed worst-case capacity of the stash.

D OMIX++ PROCEDURES AND SECURITY

In this section, we provide the detailed pseudocode of Find and Rebalance algorithms (Algorithm 5 and Algorithm 6-8) and explain how Rebalance works. Rebalance is used in the OMIX++ Insert operation. This routine is executed once in each level of AVL-tree traversal ($1.44 + \log N$ levels) to hide the height of the nodes need to be rebalanced. It takes as input the AVL node of the current level (node), its children (leftNode and rightNode), and two
Algorithm 7 Omix++ Rebalance-UpdateNodes Procedure

1: function UpdateNodes() // variables of Rebalance are accessible
2:   writeNode = Osel((cond1 | cond2), node, dummy)
3:   writeNode = Osel((cond3 | cond4 & doubleRotation), leftNode, writeNode)
4:   writeNode = Osel((cond4 | !cond3 & doubleRotation), rightNode, writeNode)
5:   dummyQ = !((cond1 | cond2) & cond3 | cond4 | doubleRotation)
6:   Oblx.Access(writeNode, dummyQ)
7:   doubleRotation = Osel((doubleRotation & !cond1 & !cond2 & !cond3 & !cond4), false, doubleRotation)
8:   doubleRotation = Osel((cond3 | cond4), true, doubleRotation)
9:   writeNode = Osel((cond1), leftNode, dummy)
10:  writeNode = Osel((cond2), rightNode, writeNode)
11:  writeNode = Osel((cond3 | cond4 | cond5), node, writeNode)
12:  dummyQ = !((cond1 | cond2 | cond3 | cond4 | cond5)
13:  Oblx.Access(writeNode, dummyQ)
14: end function

Algorithm 8 Omix++ Rebalance-Rotate Procedure

1: function Rotate(targetNode, opposNode, isRRot, isDummy)
2:   cond1 = (!isDummy & isRRot)
3:   cond2 = (!isDummy & !isRRot)
4:   tmpNode = load AVL node with key opposNode.leftKey or opposNode.rightKey from the cache based on
5:   cond1 and cond2
6:   opposNode.rightKey = Osel((cond1), targetNode.key, opposNode.rightKey)
7:   opposNode.rightPos = Osel((cond1), targetNode.pos, opposNode.rightPos)
8:   targetNode.leftKey = Osel((cond1), tmpNode.key, targetNode.leftKey)
9:   targetNode.leftPos = Osel((cond1), tmpNode.pos, targetNode.leftPos)
10:  opposNode.leftKey = Osel((cond2), targetNode.key, opposNode.leftKey)
11:  opposNode.leftPos = Osel((cond2), targetNode.pos, opposNode.leftPos)
12:  targetNode.rightKey = Osel((cond2), tmpNode.key, targetNode.rightKey)
13:  targetNode.rightPos = Osel((cond2), tmpNode.pos, targetNode.rightPos)
14: end function

Algorithm 9 GraphOS Setup

1: function Setup
2:   Client:
3:      Send encryption keys and (V,E) to the enclave
4:   Server (trusted-hardware):
5:      DOMAP tmp.Initialize([V], [L])
6:      for each v_i ∈ V do
7:         DOMAP tmp[v_i]["in"] ← 0 // incoming edge cnt
8:         DOMAP tmp[v_i]["out"] ← 0 // outgoing edge cnt
9:      end for
10:     p1 ← {}; p2 ← {}
11:    for each w_i ∈ E do
12:       b_i = w_i ← I // b_i ∈ w_i shows (initial, terminal, weight)
13:          DOMAP tmp[b_i] ← 2 // add edges
flags (restDummy, dblRotation). restDummy is set by INSERT procedure and shows whether the current level (and its corresponding node) is dummy or not. dblRotation is a flag which is used for separating double rotation conditions (Left-right and Right-left) from the single rotation ones (Left and Right).

The rebalancing algorithm, first identifies the type of rotation based on the difference between left child and right child heights. Then, it executes two rotations by calling Rotate in all cases (which is two for Left-right and Right-left).

Algorithm 10 GraphOS Add Operation for Vertex and Edge

```plaintext
1: function AddVertex(v)
2:   DOMAP [("V"||v)] ← (0, 0)
3: end function
4: function AddEdge(vinit.ิ, vterm, weight)
5:   (ininit, outinit) ← DOMAP [("V"||vinit.ิ)]
6:   (interm, outterm) ← DOMAP [("V"||vterm)]
7:   outinit++ : interm++
8:   DOMAP [("V"||vinit.ิ)] ← (ininit, outinit)
9:   DOMAP [("V"||vterm)] ← (interm, outterm)
10: DOMAP [("EOut"||vinit.ิ, outinit)] ← (vterm, weight)
11: DOMAP [("Ein"||vterm, interm)] ← (vinit.ิ, weight)
12: DOMAP [("E", vinit.ิ, vterm)] ← (weight, outinit, interm)
13: end function
```

Algorithm 11 GraphOS Remove Operation for Vertex/Edge

```plaintext
1: function RemoveEdge(vinit, vterm)
2:   (cntinit, cntterm) ← DOMAP [("E", vinit, vterm)]
3:   DOMAP [("E", vinit, vterm)] ← NULL
4:   (ininit, outinit) ← DOMAP [("V"||vinit)]
5:   (interm, outterm) ← DOMAP [("V"||vterm)]
6:   DOMAP [("EOut"||vinit, cntinit)] ←
7:   DOMAP [("Ein"||vterm, cntterm)] ←
8:   DOMAP [("Ein"||vterm, interm)] ←
9:   outinit ← outinit - 1 : interm ← interm - 1
10: DOMAP [("V"||vinit)] ← (ininit, outinit)
11: DOMAP [("V"||vterm)] ← (interm, outterm)
12: end function
13: function RemoveVertex(v)
14:   v ← DOMAP [("V"||v)]
15:   DOMAP [("V"||v)] ← NULL
16: for i = 1 to out, do
17:   in term ← DOMAP [("EOut"||v, i)]
18: end for
19: for i = 1 to in, do
20:   init ← DOMAP [("Ein"||v, i)]
21: end for
22: end function
```

that since the type of the needed rotation should not be revealed, the procedure has to execute the maximum number of needed rotations in all cases (which is two for Left-right and Right-Left). Rotate takes two AVL nodes, the direction of rotation, and a dummy flag as input and applies the needed rotation to the nodes if the dummy flag is not set. Otherwise, it executes the equivalent dummy operations.

After that, the procedure updates the rotated AVL nodes by performing some OBLIX accesses. OBLIX Access takes a node for write and a flag that shows whether the access is dummy or not. Although the actual rebalancing only appears in few levels of the AVL-tree, the algorithm should treat all levels in a similar way to avoid extra information leakage. It is important to note that the naive padding of needed OBLIX accesses in each level would lead to three accesses per level because the Left-right and Right-left rotations update three different nodes. We propose an optimization and reduce the needed OBLIX updates to two per level. To do that, we update the current level node and one of its children (depending on the traversal path) but postpone the third node update to the upper level (that is identified by dblRotation).

The following theorem characterizes the security of OMMX++.

**Theorem 1.** OMMX++ is secure according to the DOMAP security definition [83].

**Proof.** To prove the security of OMMX++, we construct a simulator that only gets the memory size as input and provides the same interface as INITIALIZE, FIND, and INSERT in the real scheme. In INITIALIZE procedure, the simulator runs INITIALIZE of OBLIX simulator and sets the root info to null. To implement FIND/INSERT procedures, the simulator runs OBLIX simulator for ACCESS procedure for $2^k \cdot \lceil \log N \rceil$ times. After each ACCESS call, the simulator sequentially scans the whole stash. Clearly, the adversary cannot distinguish the real scheme from the simulator because i) OBLIX simulator is indistinguishable from the real OBLIX scheme, ii) the adversary sees the same number of memory accesses due to the padding and sequential scans.

### E. GRAPHOS ALGORITHMS

Here we provide the pseudocode for the GraphOS algorithms. Setup (Algorithm 9) stores the graph in a DOMAP instance which can then be used to access/insert/delete vertices/edges, as outlined in Sec 5.1. The pseudocode of these operations is provided in Algorithm 10 and Algorithm 11. We also provide the pseudocodes of the complex graph queries, i.e., BFS/DFS (Algorithm 12), Minimum Spanning Tree (Algorithm 13) and Single Source Shortest Path (Algorithm 14).

The MST algorithm is more complex than BFS/DFS because we need to avoid cycle creation in the tree construction. To do that, in the generic version of the algorithm a helper function is used to extract the root of the sub-trees corresponding to source and termination vertices. This reveals some information about the structure of the graph. To avoid this, we merged the outer and inner loops of the algorithms and created a state counter (st) which determines whether the algorithm is executing the related codes of the outer-loop (st=1) or the inner-loop (st=2). In the latter case, it uses a second state variable i to determine the source or termination vertex. As the given pseudocode shows, there are no conditional branches and no execution trace depends on secret data.

The given Dijkstra algorithm, is the same as [78]; interested readers are referred to that paper for details.
Algorithm 13 Kruskal Algorithm in GraphOS

1: function Execute(DOMAP, EList)
2:     Obliviously sort edges in EList based on their weights
3:     for i = 1 to |V| do
4:         DOMAP["Root", i] ← i
5:     end for
6:     i = 1; st = 1
7:     for j = 1 to 2 * |E| + log|V| do
8:         index ← Osel(i < (|E| + 2), i, |E| + 2)
9:         (init, trm, weit) ← EList[cell(index/2)]
10:        vertex ← Osel(st == 1 & i%2 == 1, init, vertex)
11:        vertex ← Osel(st == 1 & i%2 == 0, trm, vertex)
12:        tmp = DOMAP["Root", vertex]
13:        tmp = DOMAP["Root", vertex]
14:        tmp = DOMAP["Root", vertex]
15:        tmp = DOMAP["Root", vertex]
16:        newRoot ← Osel(st == 2, tmp, newRoot)
17:        mapKey ← Osel(st == 2, vertex, -1)
18:        DOMAP["Root", mapKey] ← newRoot
19:        curRoot ← Osel(st == 2, newRoot, curRoot)
20:        vertex ← Osel(st == 2, curRoot, vertex)
21:        tmp = DOMAP["Root", vertex]
22:        curRoot ← Osel(st == 2, tmp, curRoot)
23:        initRoot ← Osel(st == 1 & i%2 == 1, vertex, initRoot)
24:        tmpRoot ← Osel(st == 1 & i%2 == 0, vertex, tmpRoot)
25:        mapKey ← Osel(st == 1 & i%2 == 0 & initRoot != tmpRoot && initRoot != -1)
26:        DOMAP["Root", mapKey] ← tmpRoot
27:        i ← Osel(st == 1, i + 1, i)
28:     end for
29: end function

Algorithm 14 Dijkstra Algorithm in GraphOS

1: function Execute(DOMAP, start)
2:     MinHeap ← ObliviousMinHeap.Initialize(|V|)
3:     for i = 1 to |V| do
4:         DOMAP["Dist", i] ← ∞
5:     end for
6:     DOMAP["Dist", start] ← 0
7:     MinHeap.AddNewNode(start, 0)
8:     innerLoop ← false; u = -1; distu = -1
9:     cnt = 1; weit = -1
10:    for i = 1 to 2 * |V| + |E| do
11:        mapKey ← Osel(innerLoop, v, dummy)
12:        u ← Osel(innerLoop, u, -1)
13:        distu ← Osel(innerLoop, curDistu, -1)
14:        tmp ← DOMAP["Dist", mapKey]
15:        distv ← Osel(innerLoop, tmp, distv)
16:        maxValue ← Osel(innerLoop & distu + weit < distv, distu + weit, distv)
17:        DOMAP["Dist", mapKey] ← maxValue
18:        OP ← Osel(innerLoop == false, EXTRACT_MIN, dummy)
19:        OP ← Osel(innerLoop & distu + weit < distv, ADD_NODE, OP)
20:        (u, distu) ← MinHeap.execute(OP, u, distu + weit)
21:        cnt ← Osel(innerLoop, cnt + 1, cnt)
22:        mapKey ← Osel(innerLoop == false & u ≠ -1, ("Dist", u, dummy))
23:        tmp ← DOMAP(mapKey)
24:        curDistu ← Osel(innerLoop == false & u ≠ -1, tmp, curDistu)

F EVALUATION OF DISTRIBUTED GRAPHOS

To implement the distributed version of GraphOS, we used the idea of [38] with an adjustable leakage for an ORAM, where a DORAM/DOMAP can be partitioned into smaller DORAMs/DOMAPs. In particular, an adjustable ORAM reveals α bits of the memory access patterns in order to partition an ORAM with size N into 2α smaller
Figure 7: Distributed GraphOS execution time for variable graph size \((V + E)\) and different machine numbers (2M means running the algorithm on 2 machines) for (a) Breadth First Search, (b) Single Source Shortest Path (Dijkstra).

ORAMs with size \(N/2^\alpha\) and improve efficiency. [38]’s partitioning is based on a Pseudorandom Permutation (PRP) which ensures that all small ORAMs have the same size. They also propose the concept of OMAP with adjustable leakage. Similarly, instead of storing all key-value pairs in one OMIX++, our proposed distributed GraphOS partitions them into multiple OMIX++’s which can be stored on different machines. When comparing our distributed GraphOS with OPAQUE for small values of \(\alpha (\leq \log \log N)\), we notice that our approach does not leak structural information about the input graph in contrast to OPAQUE (as we discussed in the previous sections). Obviously, all algorithms are not designed to be run in parallel, e.g., DFS cannot be run in a parallel way, because we need to finish all the operations on one vertex of the graph before moving to the next vertex. On the other hand, BFS and SSSP (Dijkstra) can benefit from such a distributed system.

In Figure 7, we report the GraphOS performance in a distributed setting. We use multiple threads, each using a separate CPU core to simulate different machines. We compare this with an “idealized” version of distributed OPAQUE that achieves perfect parallelization, e.g., running a graph query with 2 machines would reduce its execution time by half. This is clearly unachievable but it can still serve as a measure of how GraphOS would fare as a distributed system. We only consider BFS and SSSP queries, as they can clearly benefit from parallel execution.

Compared with idealized distributed OPAQUE, using 4 threads our system is up to \(2.2 \times\) faster for BFS in the largest size we were able to run \((|V| + |E| = 2^{14})\) (e.g., 3757s for GraphOS vs. 9402s for OPAQUE). On the other hand, GraphOS becomes faster in SSSP only for large sizes; i.e., \(1.1 \times\) faster for size \(2^{15}\). Using distributed GraphOS for sparser graphs does not seem to provide considerable improvement as the amount of parallelism these queries can leverage is limited.