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:
Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. GraphOS: Towards Oblivious Graph Processing. PVLDB, 16(13): 4324 - 4338, 2023.
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 query[1] so they fail to achieve our strong privacy requirement outlined above.

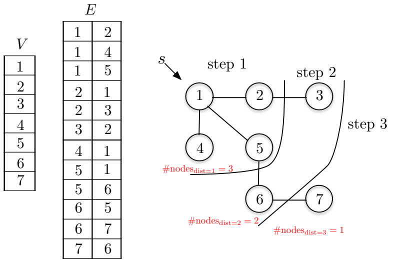
Recently, Zheng et al. [124], Eskandarian et al. [47], and Priebe et al. [91] proposed oblivious relational DBMS. These systems combine *trusted hardware* with *oblivious algorithms* to minimize the leaked information to just the size of accessed and created tables. It is important to note that trusted 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

---

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625067

[1] They 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]).



V
1
2
3
4
5
6
7

E	
1	2
1	4
1	5
2	1
2	3
3	2
4	1
5	1
5	6
6	5
6	7
7	6

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.

relational operators. For example, consider a breadth-first-search (BFS) traversal query, as shown in Figure 1. With a relational DBMS, this is executed as a sequence of joins between the vertices and the edges table, and/or self-joins of the edges table. Even if each of these joins is performed obliviously with [124], due to this multi-step approach the server is able to observe all intermediate join result sizes. Concretely, it learns the number of vertexes that have distance $1, 2, 3, \ldots$ from s, which is potentially sensitive information about the topology of the graph. Padding intermediate results to the maximum size would eliminate this leakage but with prohibitive performance downgrade (quadratic in the graph size). To some extent, this leakage is *inherent* to this approach, thus motivating the need for systems *explicitly designed for oblivious graph processing.*

Our Result. In this work, we introduce GraphOS (Graph Oblivious System)[2] an oblivious graph processing system that hides the topology of the input graph and only leaks information about its size and the result size (and type) of each query. GraphOS also relies on trusted hardware oblivious primitives but it outperforms prior state-of-art solutions in terms of performance and security. Below, we outline the novelties of GraphOS.

New doubly-oblivious primitive. As a building block for GraphOS, we propose a new *doubly-oblivious map (DOMAP)*, or in other words, a doubly-oblivious key-value store, called Omix++. It ensures that all sequences of data-structure operations are indistinguishable, even against a *hardware* adversary that can observe the memory access pattern imposed by the client-side operations (which, in our system, are performed by the trusted hardware). We stress that "standard" ORAM techniques (e.g., the classic Square-Root ORAM [54] and PathORAM [107]) do not suffice to achieve this level of security in our model, as their client-side routines may still leak information to an adversary that can observe their memory access pattern (e.g., when executed from trusted hardware at the server). See also the extended discussion in Sec 3.1 and Figure 2.

GraphOS uses Omix++ to access graph vertexes/edges without revealing the accessed element, being in the ballpark of prior plaintext 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 leafs, 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

---

[2] Inspired by the similarly pronounced greek word for graph, Γράφος.

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 DOMAP (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 DOMAP built from [83], which we call Omix, in three applications: *private contact discovery*, *key transparency logs*, and *searchable encryption*. Our results show that an Omix++ access (look-up) is overall **1.8–20**× faster, resulting in the most efficient existing DOMAP. 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 Omix 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 2^{24} entries takes 37ms computation time with Omix++ vs. 767ms with Omix.

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). Klotski [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. Shroud [79] parallelizes across multiple co-processors the Binary Tree ORAM [101]—both Shroud 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, 58, 94, 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. Obliviate [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 neighbours 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.

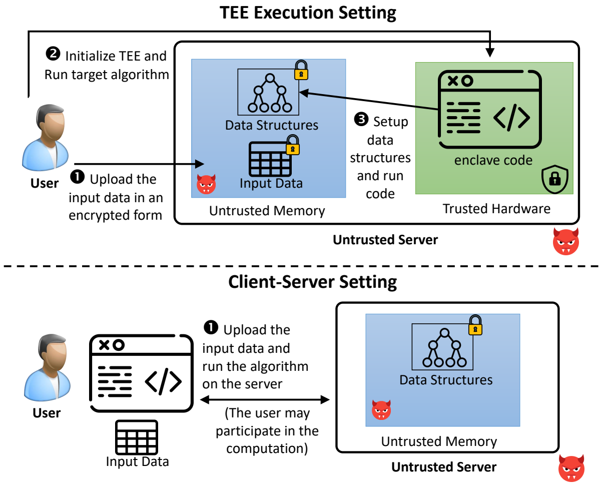


Figure 2: TEE (top) vs. Client-Server (bottom) settings. In TEE, the user uploads encrypted data and sets up the enclave. Data structures are then initialized and code is executed at the server. In Client-Server, the user may locally maintain some data and participate in parts of the computation.

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

$$Osel(c, a, b) = (c \ \& \ a) \mid (!c \ \& \ b)$$
$$Ocmp(a, b) = -((a - b) \gg (l - 1)) + ((b - a) \gg (l - 1)),$$

where $!, \&, |, \gg$ 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 described in detail in the extended version. During initialization, Setup creates a Path-ORAM 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 $O(CN \log^3 N)$ and $O(k^2 C \log^2 N)$, 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 OMAP 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($[\mathsf{bl_i}]_1^n$, $N$)
 2:     Nodes ← $[\mathsf{bl_i}]_1^n$  ▹ 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, strt, end)
12:     if (strt > end) return (-1,0)          ▹ (node leaf, node key)
13:     mid = $\lfloor (\mathsf{strt} + \mathsf{end})/2 \rfloor$
14:     curRoot ← Nodes[mid]
15:     (curRoot.leftChildKey, curRoot.leftChildPos) ←
                    CreateAVLTree(Nodes, strt, 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
```

Algorithm 2 Omix++ Find Procedure

```
 1: function Find(key, root, $N$)
 2:     (curkey, curPos) ← key and leaf position of the root node
 3:     cnt = 0; result =⊥
 4:     do
 5:         Retrieve curNode while setting a new random
            position for that and its child through DORAM.Access
            for (curkey, curPos)
 6:         Keep the new random position of the child and use it
            as the new position of the node in the next iteration
 7:         cmpRes ← Ocmp(key, curNode.Key)
 8:         (curkey, curPos) ← Evaluate cmpRes. If the target key
            is found, return a dummy pair. Otherwise, select the
            left/right child of curNode for the next step using Osel
 9:         Assign curNode.Value to result obliviously if cmpRes
            shows the equality
10:         cnt + +
11:     while cnt ≤ $1.44 * \log N$
12:     return result
13: 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 dummies 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 $2 \cdot \lceil 1.44 \cdot \log N \rceil$ 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 leafs 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 overflowed 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.

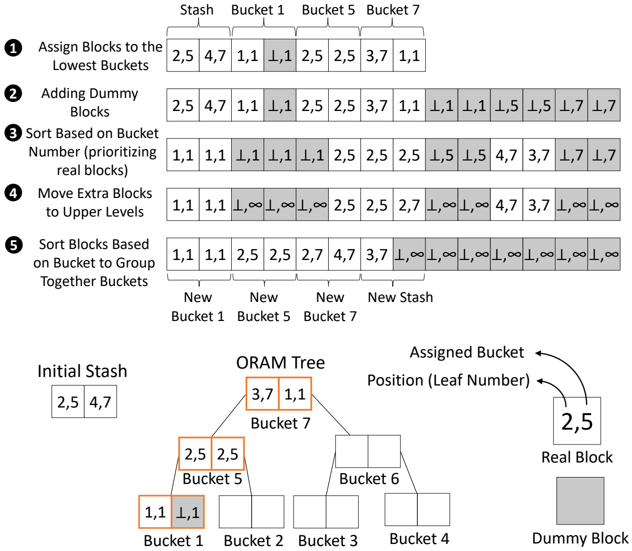


Figure 3: Improved Eviction Policy. The size of each bucket and the permanent stash is assumed to be 2; blocks' values are omitted. (1) assign real blocks of stash+path to lowest possible bucket. (2) add $C = 2$ dummy blocks for each bucket. (3) sort blocks based on assigned bucket prioritizing real ones over dummies. (4) move extra real blocks to upper levels. (5) group together blocks of buckets by another oblivious sort.

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 $O(\log N)$ 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

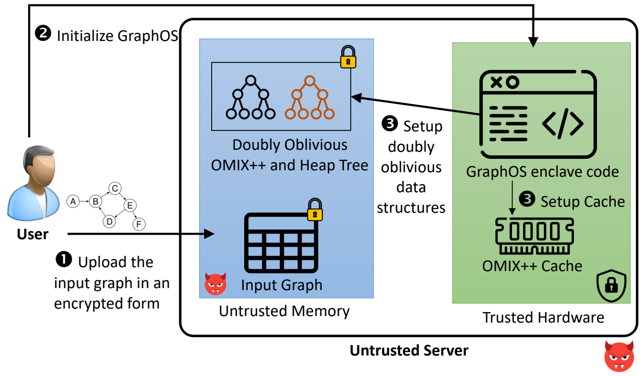


Figure 4: Architecture of GraphOS and its initialization steps. (1) upload the input graph in encrypted form. (2) setup GraphOS enclave. (3) initialize the needed data structures.

sees an even number. Although the sequence of data accesses is deterministic and *a priori* known to the adversary, observing which instruction is being fetched for each array position leaks information. 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 array positions 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. Considering 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 oblivious 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 complementary; the first eliminates data-access leakage and the second eliminates instruction-access leakage. We implemented this approach 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_{trm} with weight w, we store three key-value pairs:
  - This pair has key $(“EOut”||v_{init}, cnt)$ and value (v_{trm}, w) where “$EOut$” 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 $(“EIn”||v_{trm}, cnt)$ 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_{trm}.
  - This pair has key $(“E”, v_{init}, v_{trm})$ and value $(w, cnt_{init}, cnt_{trm})$, 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 enclave 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 iterates 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 explained 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 Initialization procedure of Omix++ for all created key-value pairs. Setup performs a loop over all edges and corresponding Omix++ Inserts $(O(C \log^2 |E| \log^2 \log |E|)$ assuming $|E| \geq |V|)$. Hence its complexity is $O(C|E| \log^3 |E|)$, 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 certain 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 edge weight, 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. 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"$||v$) with value $(0, 0)$ to Omix++. To add edge (v_{init}, v_{trm}, w), it first fetches the current number of incoming edges to v_{trm} (denoted by in_{trm}) and the number of outgoing edges from v_{init} (denoted by out_{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_{init}, v_{trm}), GraphOS has to remove the corresponding data from v_{init} and v_{trm}. It extracts the related counters of the target edge by fetching the edge counters of the initial and terminal vertices (cnt_{init} and cnt_{trm}) using key ("E", v_{init}, v_{trm}) 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"$||v$). 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 is equal to $O(C \log^2 |E| \log^2 \log |E|)$ assuming $|E| \geq |V|$ because of their constant number of DOMAP queries. On the other hand, the complexity of vertex removal is $O(|V| \cdot C \log^2 |E| \log^2 \log |E|)$ because in the worst case, the vertex is connected to all others.

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.

BFS/DFS. These two queries are graph traversals that load and unload vertexes 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 (dequeue) 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.

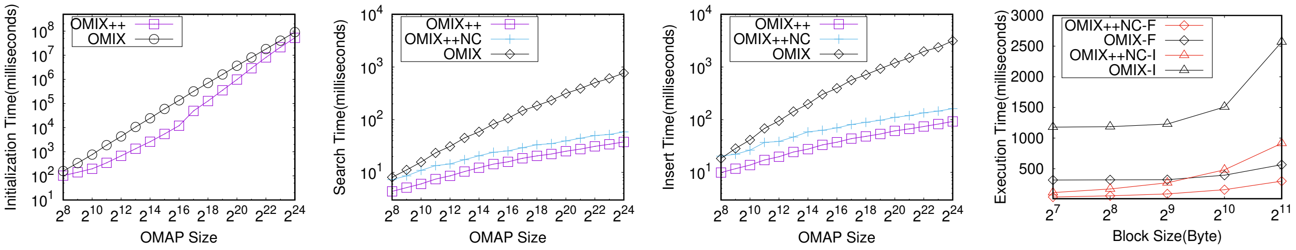
Minimum Spanning Tree. Our MST algorithm is based on the classic Kruskal [73] where edges are sorted based on their weights. Instead of running $|E|$ DOMAP queries, we do this efficiently by obliviously sorting the edges using a copy of DOMAP blocks (to avoid data corruption in DOMAP) which are then fetched sequentially (**EList**). 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 Oblix (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 $O(C|E| \log^2 |E| \log^2 \log |E|)$ while for MST it is $O(C|E| \log |V| \log^2 |E| \log^2 \log |E|)$ assuming that $|E| \geq |V|$. For comparison, Opaque's complexity for BFS/DFS and MST is $O(C|V|^2 |E| \log^2 |E|)$, $O(C|E||V|^2 \log^2 |V|)$ 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



(a) (b) (c) (d)

Figure 5: (a) DOMAP Initialization, (b) Find and (c) Insert times for variable OMAP sizes, (d) DOMAP Find (denoted by F) and Insert (denoted by I) time for variable block size in an OMAP with size 2^{23}

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 2^8–2^{18}. 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 Opaque 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 Opaque execution time for sizes which would take several days/months. In addition to Opaque, 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 Ubuntu16.04 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 4̃0% improvement in BFS/DFS execution time because we have removed the need for complex oblivious rotation. For Opaque 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 Opaque 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[3] 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 Oblix 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 Oblix setup, followed by inserting key-value pairs one-by-one. Throughout all our experiments, Omix++ setup is 1.5–11× 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).

Operation	System	Time (seconds) size ($2^{12}/2^{18}$)
setup+RA	GraphOS	99 / 19566
	Opaque	0.9 / 13
look-up vertex/edge	GraphOS	0.01 / 0.02
	Opaque	1 / 1.9
add vertex	GraphOS	0.02 / 0.06
	Opaque	0.8 / 8.2
add edge	GraphOS	0.3 / 0.6
	Opaque	0.8 / 8.2
remove vertex	GraphOS	0.07 / 0.15
	Opaque	0.7 / 4.4
remove edge	GraphOS	0.3 / 0.7
	Opaque	0.7 / 4.4

mechanism we deploy, as explained in Sec 4.1. In particular, Omix++ searches are 1.8–20× faster than Omix (e.g., for $N = 2^{24}$ the former takes 37ms and the latter 767ms) and insertions are 2–34× faster (e.g., for $N = 2^{24}$ the former takes 92ms and the latter $> 3s$).

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 top major improvement 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. This follows since the underlying Oblix eviction of Omix becomes the bottleneck for large N (ignoring constants, it takes $O(\log^4 N)$ vs. $O(\log^2 Nlog^2logN)$ for Omix++). Overall, the main source of improvement of Omix++ is the individual eviction policy (also confirmed by our variable block-size experiment in Sec 6.4).

Real-world applications of Omix++. Next, we compare the performance of Omix++ with Omix in three real-world applications.

Private contact discovery in Signal. Signal [8] makes a private contact discovery by searching the given contact list inside the Signal database within the trusted hardware. To prevent access pattern leakage, a naive (baseline) solution is to do several sequential scans instead of direct accesses. We executed an experiment to measure the improvement of using Omix++ in this application. We set N (number of users) to 128M and the block size to 160 bytes. Our results show that using Omix++ improves the Signal performance 6.3× for $m = 100$ where m is the size of the user's contact list and $N = 128M$ (while Omix only provides 30% improvement). Furthermore, for the incremental contact discovery ($m = 1$), using Omix++ gives up to three orders of magnitude improvement while Omix provides two orders of magnitude improvement.

Anonymizing Google's Key Transparency. Google KT [110] provides integrity in the public-key look-up use case. To do that, it maintains a Merkle tree over all public keys and shares the root of the tree with the users. However, it does not provide anonymity and the server can identify the identity of the target user. A naive solution for providing anonymity is to do several sequential scans to hide the access pattern (we consider this solution as the baseline approach similar to [83]). A more clever solution is to use DOMAP and access these keys through this oblivious data structure. We executed an experiment and used $N = 20M$ public keys with block size 256 bytes where N is the number of keys in the Merkle tree (similar to [83]). According to our results, for small N, Omix++ approach is 126% faster than the baseline approach while Omix approach is only 9% faster (E.g., the baseline, Omix++, and Omix approaches take 904ms,56ms, and 830ms respectively). On the other hand, as N increases, our approach has a significantly lower cost. For example, for $N = 40M$, our approach is 32× faster than baseline while Omix approach is only 2× faster. E.g., the baseline approach, Omix, and Omix++ approaches take 1992ms, 996ms, and 61ms respectively.

Searchable Encryption. We compared Omix and Omix++ performance for searchable encryption [36, 52, 69, 104] using the entire Enron email dataset [30] consisting of 528K emails. After keyword extraction and filtering words that contained non-alphabetic characters, we achieved 38M key-value pairs. We initialized DOMAPs using key-values with a block size of 200 bytes. We measured the search and insertion time of the inverted index over the above key-value pairs. According to our experimental results, the search time per key-value pair using Omix++ is 17× faster than Omix. On the other hand, the insertion time of Omix++ is 25× faster than Omix.

6.2 Basic Graph Operations

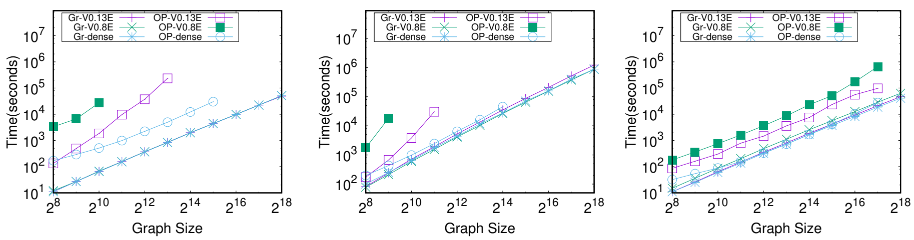
We report the performance of basic operations (setup, searching/adding/removing a vertex/edge) in GraphOS and Opaque in Table 1.

Setup Time. Overall, Opaque has a faster setup than GraphOS. E.g., it takes 13s to setup a graph with size 2^{18} for Opaque but 19566s for GraphOS. This should come as no surprise since GraphOS has to build oblivious indexes so that later it achieves more efficient query execution. On the other hand, we can postpone the oblivious index creation to query execution time (for BFS/DFS, MST, etc.), using the idea of adaptive indexing from plaintext databases [4, 64]. Somewhat surprisingly, this (initializing GraphOS on-the-fly and executing the query) is still significantly faster than executing queries in an already set-up Opaque, as we show in Sec 6.4.

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 adding/removing a vertex. Adding an edge in GraphOS takes more time than adding a vertex as it takes multiple DOMAP accesses (to update adjacent vertex information) and likewise for vertex removal.

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 2^8–2^{15}, 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



(a) (b) (c)

Figure 6: Execution time of (a) Breadth First Search/Depth First Search, (b) MST (Kruskal), (c) SSSP (Dijkstra) for variable graph sizes ($|V| + |E|$).

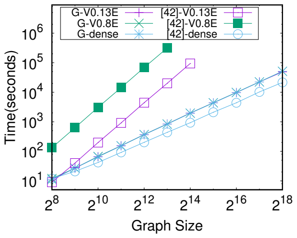


Figure 8: DFS of [78] vs. our DFS for variable graph sizes

faster. This agrees with its achieved asymptotic improvement of $O(V^2/\log^2 \log E)$ over OPAQUE. Recall that this improvement in performance is accompanied by strictly less leakage. GraphOS only reveals $|V|$ and $|E|$, whereas OPAQUE 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 $2^8 - 2^{14}$ (e.g., it takes 212s for graph size 512 while OPAQUE 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 OPAQUE in executing Dijkstra. E.g., GraphOS is 1–22× faster for sizes up to 2^{17}. Furthermore, GraphOS only reveals $|V|$ and $|E|$, whereas OPAQUE trivially reveals the number of neighbours of each vertex (again, eliminating this leakage of OPAQUE 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 OMIX++, we measured the Find/Insert time varying the block size betwenn 128-2048 bytes while fixing the size to 2^{23} (Figure 5(d)). For fairness, we disabled the path-cache of OMIX++, as this can be used in both schemes. As shown, OMIX++ clearly outperforms OMIX for all block sizes, both for Insert (I) and Find (F). Concretely OMIX++ with disabled path-cache is 1.9–10.6× faster in Find and 2.8–10.6× 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×–4.98×. 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×, whereas for very sparse ones it is close to 2.5×. The gap for our MST is 1.2×–8.5×. 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×, 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×). For more sparse graphs, ours is faster 0.8–374× 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) OPAQUE for BFS and SSSP.

Integrating OPAQUE and GraphOS. We also evaluated an "integrated" approach of OPAQUE 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 OPAQUE 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 OPAQUE 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 $0.9 + 99 + 368 \approx 468$s, $0.9 + 99 + 4386 \approx 4486$s, and $0.9 + 99 + 356 \approx 456$s while in OPAQUE it takes $0.9{+}37328 \approx 37329$s, $0.9{+}6429 \approx 6430$s, and $0.9{+}1462 \approx 1463$s, respectively (0.9s is for OPAQUE 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.

REFERENCES

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/obfuscuro-a-commodity-obfuscation-engine-on-intel-sgx/

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX.. In *NDSS*.

[3] Nouf Al-Juaid, Alexei Lisitsa, and Sven Schewe. 2022. SMPG: Secure Multi Party Computation on Graph Databases.. In *ICISSP*. 463–471.

[4] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* ACM, 1103–1114.

[5] Abdelrahaman Aly and Mathieu Van Vyve. 2014. Securely Solving Classical Network Flow Problems. In *Information Security and Cryptology ICISC 2014 17th International Conference, Seoul, Korea, December 3-5, 2014, Revised Selected Papers (Lecture Notes in Computer Science)*, Jooyoung Lee and Jongsung Kim (Eds.), Vol. 8949. Springer, 205–221. https://doi.org/10.1007/978-3-319-15943-0_13

[6] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel privacy-preserving shortest path algorithms. *Cryptography* 5, 4 (2021), 27.

[7] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2022. Privacy-Preserving Parallel Computation of Minimum Spanning Forest. *SN Computer Science* 3, 6 (2022), 448.

[8] Signal App. 2014. https://github.com/signalapp/.

[9] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure graph analysis at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* 610–629.

[10] ARM Limited. 2004. ARM TrustZone Technology. https://developer.arm.com/documentation/102412/latest.

[11] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data.* ACM, 1383–1394.

[12] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. Optorama: Optimal Oblivious RAM. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 403–432.

[13] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on Hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11, 3 (2011), 5–9.

[14] Sumeet Bajaj and Radu Sion. 2013. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (2013), 752–765.

[15] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference.* ACM, 307–314.

[16] Marina Blanton and Siddharth Saraph. 2014. Secure and oblivious maximum bipartite matching size algorithm with applications to secure fingerprint identification. *Department of Computer Science and Engineering University of Notre Dame* (2014).

[17] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security.* 207–218.

[18] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2016. Oblivious parallel RAM and applications. In *Theory of Cryptography Conference.* Springer, 175–204.

[19] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17).*

[20] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J Carey, and Tyson Condie. 2014. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment* 8, 2 (2014), 161–172.

[21] Anrin Chakraborti and Radu Sion. 2018. ConcurORAM: High-throughput stateless parallel multi-client ORAM. *arXiv preprint arXiv:1811.04366* (2018).

[22] T.-H. Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. 2018. More is Less: Perfectly Secure Oblivious Algorithms in the Multi-server Setting. In *ASIACRYPT 2018, Proceedings, Part III (Lecture Notes in Computer Science)*, Thomas Peyrin and Steven D. Galbraith (Eds.), Vol. 11274. Springer, 158–188. https://doi.org/10.1007/978-3-030-03332-3_7

[23] TH Hubert Chan, Elaine Shi, Wei-Kai Lin, and Kartik Nayak. 2020. Perfectly oblivious (parallel) RAM revisited, and improved constructions. *Cryptology ePrint Archive* (2020).

[24] T-H Hubert Chan, Kai-Min Chung, and Elaine Shi. 2017. On the depth of oblivious parallel RAM. In *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 567–597.

[25] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2017. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 660–690.

[26] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings (Lecture Notes in Computer Science)*, Masayuki Abe (Ed.), Vol. 6477. Springer, 577–594. https://doi.org/10.1007/978-3-642-17373-8_33

[27] Binyi Chen, Huijia Lin, and Stefano Tessaro. 2016. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography Conference.* Springer, 205–234.

[28] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi (Eds.). ACM, 7–18. https://doi.org/10.1145/3052973.3053007

[29] Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2014. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II (Lecture Notes in Computer Science)*, Palash Sarkar and Tetsu Iwata (Eds.), Vol. 8874. Springer, 62–81. https://doi.org/10.1007/978-3-662-45608-8_4

[30] William W. Cohen. 2015. Enron email dataset. https://www.cs.cmu.edu/ enron/. *Carnegie Mellon University* (2015).

[31] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. 2017. The pyramid scheme: Oblivious RAM for trusted processors. *arXiv preprint arXiv:1712.07882* (2017).

[32] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[33] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).* 727–743.

[34] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. 2011. Perfectly Secure Oblivious RAM without Random Oracles. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings (Lecture Notes in Computer Science)*, Yuval Ishai (Ed.), Vol. 6597. Springer, 144–163. https://doi.org/10.1007/978-3-642-19571-6_10

[35] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.*

[36] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2020. Dynamic Searchable Encryption with Small Client Storage. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/dynamic-searchable-encryption-with-small-client-storage/

[37] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency. *CRYPTO* (2018).

[38] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2433–2450. https://www.usenix.org/conference/usenixsecurity20/presentation/demertzis

[39] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. 2016. Practical Private Range Search Revisited. In *SIGMOD.*

[40] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. 2018. Practical Private Range Search in Depth. *TODS* (2018).

[41] Ioannis Demertzis and Charalampos Papamanthou. 2017. Fast Searchable Encryption With Tunable Locality. In *SIGMOD.*

[42] Ioannis Demertzis, Rajdeep Talapatra, and Charalampos Papamanthou. 2018. Efficient searchable encryption through compression. *PVLDB* (2018).

[43] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271. https://doi.org/10.1007/BF01386390

[44] Jack Doerner, David Evans, and Abhi Shelat. 2016. Secure Stable Matching at Scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1602–1613. https://doi.org/10.1145/2976749.2978373

[45] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 523–535.

[46] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. 2022. Benchmarking the Second Generation of Intel SGX Hardware. In *Data Management on New Hardware*. 1–8.

[47] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019), 169–183. https://doi.org/10.14778/3364324.3364331

[48] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. 2015. Three-party ORAM for secure computation. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 360–385.

[49] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. 2015. A low-latency, low-area hardware oblivious RAM controller. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 215–222.

[50] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings (Lecture Notes in Computer Science)*, Emiliano De Cristofaro and Matthew K. Wright (Eds.), Vol. 7981. Springer, 1–18. https://doi.org/10.1007/978-3-642-39077-7_1

[51] Javad Ghareh Chamani. 2023. GraphOS. https://github.com/jgharehchamani/graphos.

[52] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. 2022. Dynamic searchable encryption with optimal search in the presence of deletions. In *31st USENIX Security Symposium (USENIX Security 22)*. 2425–2442.

[53] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1038–1055.

[54] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553

[55] Michael T. Goodrich and Joseph A. Simons. 2014. Data-Oblivious Graph Algorithms in Outsourced External Memory. In *Combinatorial Optimization and Applications - 8th International Conference, COCOA 2014, Wailea, Maui, HI, USA, December 19-21, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhao Zhang, Lidong Wu, Wen Xu, and Ding-Zhu Du (Eds.), Vol. 8881. Springer, 241–257. https://doi.org/10.1007/978-3-319-12691-3_19

[56] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure two-party computation in sublinear (amortized) time. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 513–524. https://doi.org/10.1145/2382196.2382251

[57] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2.

[58] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency smoothing for encrypted data stores. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2451–2468.

[59] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX*.

[60] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 299–312.

[61] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable Collaborative Analytics System on Private Database with Malicious Security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1740–1753.

[62] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. 2020. MOSE: Practical Multi-User Oblivious Storage via Secure Enclaves. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. 17–28.

[63] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. 2019. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *Proceedings on Privacy Enhancing Technologies* 2019, 1 (2019).

[64] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proc. VLDB Endow.* 4, 9 (2011), 585–597. https://doi.org/10.14778/2002938.2002944

[65] Alekh Jindal, Samuel Madden, Amol Deshpande, and Michael Stonebraker. 2014. Graph Analytics on Relational Databases. *NEDB* (2014).

[66] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. 2014. Vertexica: Your relational friend for graph analytics! *Proceedings of the VLDB Endowment* 7, 13 (2014), 1669–1672.

[67] Seny Kamara and Tarik Moataz. 2018. SQL on structurally-encrypted databases. In *ASIACRYPT International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 149–180.

[68] Seny Kamara and Tarik Moataz. 2019. SQL on Structurally-Encrypted Databases. *ASIACRYPT* (2019).

[69] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *ACM CCS 2012*. 965–976.

[70] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[71] Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *ASIACRYPT International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 506–525.

[72] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[73] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[74] Russell WF Lai and Sherman SM Chow. 2017. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS International Conference on Applied Cryptography and Network Security*. Springer, 478–497.

[75] Peeter Laud. 2015. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *Proc. Priv. Enhancing Technol.* 2015, 2 (2015), 188–205. https://doi.org/10.1515/popets-2015-0011

[76] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[77] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 87–101.

[78] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 359–376. https://doi.org/10.1109/SP.2015.29

[79] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. 2013. Shroud: Ensuring private access to large-scale data in the data center. In *11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*. 199–213.

[80] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2010. Graphlab: A new parallel framework for machine learning. In *Conference on uncertainty in artificial intelligence (UAI)*, Vol. 20.

[81] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).

[82] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *CCS*.

[83] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 279–296.

[84] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*.

[85] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In *CCS*.

[86] Kartik Nayak and Jonathan Katz. 2016. An Oblivious Parallel RAM with O(log^2 N) Parallel Runtime Blowup. *IACR Cryptol. ePrint Arch.* (2016), 1141. http://eprint.iacr.org/2016/1141

[87] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 377–394.

[88] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. 2018. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*.

[89] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 85–100.

[90] Technology preview: Private contact discovery for signal. accessed:2023-03-02. https://signal.org/blog/building-faster-oram/.

[91] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.

[92] Vijaya Ramachandran and Elaine Shi. 2020. Data oblivious algorithms for multicores. *arXiv preprint arXiv:2008.00332* (2020).

[93] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptol. ePrint Arch.* 2014 (2014), 997.

[94] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–217.

[95] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_02B-4_Sasy_paper.pdf

[96] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 38–54.

[97] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24.

[98] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 505–516.

[99] Elaine Shi. 2020. Path oblivious heap: Optimal and practical oblivious priority queue. In *SP*.

[100] Elaine Shi. 2020. Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 842–858. https://doi.org/10.1109/SP40000.2020.00037

[101] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O ((logN) 3) worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*. Springer, 197–214.

[102] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.. In *NDSS*.

[103] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *AsiaCCS*.

[104] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *IEEE SP 2000*. 44–55.

[105] Intel® Software Guard Extensions SSL. 2011. https://github.com/intel/intel-sgx-ssl.

[106] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 253–267.

[107] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 299–310.

[108] Tomas Toft. 2011. Secure data structures based on multi-party computation. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 291–292.

[109] Shruti Tople, Yaoqi Jia, and Prateek Saxena. 2019. Pro-oram: Practical read-only oblivious {RAM}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 197–211.

[110] Google's Key Transparency. 2011. https://github.com/google/keytransparency.

[111] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 289–300.

[112] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[113] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: Secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

[114] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: Architecting Efficient Outsourced Databases using Incremental MPC and Differential Privacy. In *Proceedings of the 2022 International Conference on Management of Data*. 818–832.

[115] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*.

[116] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 850–861.

[117] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: oblivious RAM for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 191–202.

[118] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 215–226.

[119] Peter Williams, Radu Sion, and Alin Tomescu. 2012. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 977–988.

[120] Konstantinos Xirogiannopoulos and Amol Deshpande. 2017. Extracting and analyzing hidden graphs from relational databases. In *SIGMOD*.

[121] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting square-root ORAM: efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 218–234.

[122] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. 2020. Klotski: Efficient obfuscated execution against controlled-channel attacks. In *ASPLOS*.

[123] Wenting Zheng. 2017. Opaque. https://github.com/ucbrise/opaque.

[124] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.

Algorithm 4 Improved Oblix Eviction AssignBlocksToBuckets

```
function AssignBlocksToBuckets(Allblocks, $i$, $cnt$, $level$, $curBuckID$, $N$)
    $block \leftarrow$ Allblocks$[i]$
    $cond1 = Osel((cnt - (\log N + 1 - level)C \geq C), 1, 0)$
    $cond2 = Osel((block.BucketID == curBuckID), 1, 0)$
    $cond3 = Osel((block$ is dummy or $cnt \geq C(\log N + 1)),$
            $1, 0)$
    $tmpBucketID$ = bucket id of level $(\log N + 1) -$
            $\lfloor (cnt/C) \rfloor$ in path $l$
    $nextBucketID$ = bucket id of $level - 1$ in path $l$
    $block$.bucketID = $Osel((cond1 \& cond2 \& cond3),$
            $\infty$, $block$.bucketID)
    $block$.bucketID = $Osel((cond1 \& cond2 \& !cond3),$
            $tmpBucketID$, $block$.bucketID)
    $cnt = Osel((cond1 \& cond2 \& !cond3), cnt + 1,$ cnt)
    $cnt = Osel((!cond1 \& cond2), cnt + 1, cnt)$
    $level = Osel((cond1 \& !cond2), level - 1, level)$
    $i = Osel((cond1 \& !cond2), i - 1, i)$
    $curBuckID = Osel((cond1 \& !cond2),$
            $nextBucketID, curBuckID)$
end function
```

Algorithm 3 Improved Oblix Eviction Procedure

```
function Evict($l$)
    Pblocks $\leftarrow$ Download blocks of path $l$
    Sblocks $\leftarrow$ Download all the blocks from the stash
    Allblocks $\leftarrow$ Pblocks $\cup$ Sblocks.
    Assign non-dummy blocks to lowest level in path $l$
    Add $C$ dummy blocks to each level and set their level
    Obliviously sort Allblocks by their assigned level
     giving priority to the non-dummy blocks in each level
    $level = \log N + 1$
    $curBuckID$ = bucket id of the lowest level in path $l$
    for $i = 1$ to |Allblocks| do
        Assign Allblocks$[i]$ to $curBuckID$ if it is not full,
         Otherwise:
            If Allblocks$[i]$ is non-dummy, assign it to its closest
               non-full bucket in the upper levels.
            If Allblocks$[i]$ is dummy, mark it as $\infty$
        Decrement $level$ and update $curBuckID$
    end for
    Perform a sequential scan over Allblocks and mark 0
     all the blocks with level $\infty$ and 1 the remaining ones
    Perform an oblivious sort on Allblocks
    for $i = 1$ to $(\log N + 1)$ do
        currentBucket = Allblocks$[C \cdot i \ldots C \cdot (i + 1)]$
        Store currentBucket at the server in level $i$
    end for
    Store remaining blocks in the stash up to its capacity
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:

- $(\sigma; T) \leftarrow$ Initialize$(1^\lambda, N)$: Given a security parameter λ 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. Finally, a data structure for storing the overflowing blocks and temporary blocks is initialized and called stash (S). The encrypted tree T is sent to the server, while $\sigma = (M, S, sk)$ is stored locally.
- $(\sigma, T(M[y]); T') \leftarrow$ Access$(r/w, y, null/val, M, S, sk; T)$: 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.
  To write (denoted by w) the val for the given index y, the client repeats the above procedure except that it updates the value of the found block with val.
- $(\sigma, S'; T') \leftarrow$ 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) Initialize,

---

[4] This non-standard notation means that for any $f(N) \in \omega(1)$, the access cost is $O(C \log^2 N f(N))$; in the context of the applications we examine here, for all practical purposes one can consider $f(N)$ as constant.

Algorithm 5 Omix++ Find Procedure

```
function Find(key, root, $N$)
    curkey = root.key,lastPos = root.pos
    newPos $=\xleftarrow{\$} [1, N]$, result=“”, upperBound = $1.44 * \log N$
    root.pos = newPos, cnt = 0, dummyState = 0
    do
        newChildPos $=\xleftarrow{\$} [1, N]$; cnt + +
        isDummy = $Osel((\text{dummyState} == 1), true, false)$
        head ←Oblix.Access(lastPos,curKey,newPos,
                newChildPos,key, isDummy)
        $cond1 = Osel((\text{dummyState} == 1), true, false)$
        $cond2 = Osel((\text{head}.key > \text{key}), true, false)$
        $cond3 = Osel((\text{head}.key < \text{key}), true, false)$
        $cond4 = Osel((\text{head}.key == \text{key}), true, false)$
        lastPos = $Osel((cond1)$, random position, lastPos)
        lastPos = $Osel((!cond1\ \&\ cond2)$, head.$leftPos$,
                lastPos)
        lastPos = $Osel((!cond1\ \&\ !cond2\ \&\ cond3)$,
                head.$rightPos$, lastPos)
        curkey = $Osel((!cond1\ \&\ cond2)$, head.$leftKey$,
                dummy)
        curkey = $Osel((!cond1\ \&\ !cond2\ \&\ cond3)$,
                head.$rightKey$, curkey)
        newPos = $Osel((cond1)$, random position, newPos)
        newPos = $Osel((!cond1\ \&\ (cond2 \mid cond3))$,
                newChildPos, newPos)
        result = $Osel((!cond1)$, head.$value$, result)
        dummyState = $Osel((!cond1\ \&\ !cond2\ \&\ !cond3$
                and $cond4)$, dummyState + 1, dummyState)
    while cnt $\leq$ upperBound
    return result
end function
```

(ii) Insert, and (iii) Find. Initialize constructs a Path-ORAM tree and stores an empty root node in a random leaf. In each Insert/Find operation, the AVL-tree is traversed for finding the requested node (a node access in the AVL-tree corresponds to one Path-ORAM access). After each access, nodes are mapped to new random positions and are re-encrypted freshly before being stored on the server. In Insert, rebalancing of the AVL-tree may be needed. The asymptotic complexity of this Find/Insert is $O(C \log^2 N)\omega(1)$.

B OBLIX ALGORITHMS

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

- Initialize(N, $[bl_i]_1^n$): Oblix proposes an initialization mechanism that gets the maximum number of blocks N and an initial list of n blocks of data $[bl_i]$. It constructs a Path-ORAM tree such that each node stores a bucket with a constant number of $[bl_i]$ 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 $[bl_i]$ 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.
- $bl \leftarrow$ 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 ∞ 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 ∞ blocks and an oblivious sort that groups together 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

```
function UpdateNodes( )        ▹ variables of Rebalance are accessible
    writeNode = $Osel((cond1 \mid cond2)$, node, dummy)
    writeNode = $Osel((cond3 \mid (!cond4\ \&\ dbleRotation)$,
        leftNode, writeNode)
    writeNode = $Osel((cond4 \mid (!cond3\ \&\ dbleRotation)$,
        rightNode, writeNode)
    $dumyQ \leftarrow !(cond1 \mid cond2 \mid cond3 \mid cond4 \mid$
        $dbleRotation)$
    Oblix.Access(writeNode,$dumyQ$)
    $dbleRotation = Osel((dbleRotation\ \&\ !cond1\ \&\ !cond2$
        $\&\ !cond3\ \&\ !cond4), false, dbleRotation)$
    $dbleRotation = Osel((cond3 \mid cond4), true,$
        $dbleRotation)$
    writeNode = $Osel((cond1)$, leftNode, dummy)
    writeNode = $Osel((cond2)$, rightNode, writeNode)
    writeNode = $Osel((cond3 \mid cond4 \mid cond5)$, node,
        writeNode)
    $dumyQ \leftarrow !(cond1 \mid cond2 \mid cond3 \mid cond4 \mid cond5)$
    Oblix.Access(writeNode,$dumyQ$)
end function
```

Algorithm 8 Omix++ Rebalance-Rotate Procedure

```
function Rotate(targetNode, opposNode, $isRRot$, $isDummy$)
    $cond1 = (!isDummy\ \&\ isRRot)$
    $cond2 = (!isDummy\ \&\ !isRRot)$
    tmpNode ← load AVL node with key opposNode.$leftKey$
        or opposNode.$rightKey$ from the cache based on
        $cond1$ and $cond2$
    opposNode.$rightKey = Osel((cond1)$, targetNode.$key$,
        opposNode.$rightKey$)
    opposNode.$rightPos = Osel((cond1)$, targetNode.$pos$,
        opposNode.$rightPos$)
    targetNode.$leftKey = Osel((cond1)$, tmpNode.$key$,
        targetNode.$leftKey$)
    targetNode.$leftPos = Osel((cond1)$, tmpNode.$pos$,
        targetNode.$leftPos$)
    opposNode.$leftKey = Osel((cond2)$, targetNode.$key$,
        opposNode.$leftKey$)
    opposNode.$leftPos = Osel((cond2)$, targetNode.$pos$,
        opposNode.$leftPos$)
    targetNode.$rightKey = Osel((cond2)$, tmpNode.$key$,
        targetNode.$rightKey$)
    targetNode.$rightPos = Osel((cond2)$, tmpNode.$pos$,
        targetNode.$rightPos$)
    update height of targetNode and opposNode based on
     $cond1$ and $cond2$
end function
```

Algorithm 9 GraphOS Setup

```
function Setup
Client:
    Send encryption keys and $(V,E)$ to the enclave
Server (trusted-hardware):
    DOMAP $_{tmp}$.Initialize($1^\lambda$, $|V|$, $\perp$)
    for each $v_i \in V$ do
        DOMAP $_{tmp}[v_i||$“$in$”$] \leftarrow 0$        ▹ incoming edge cnt
        DOMAP $_{tmp}[v_i||$“$out$”$] \leftarrow 0$       ▹ outgoing edge cnt
    end for
    $p_1 \leftarrow \{\}; p_2 \leftarrow \{\}$
    for each $v_i v_j w_{ij} \in E$ do
                        ▹ $v_i v_j w_{ij}$ shows (initial, terminal, weight)
```

Algorithm 6 Omix++ Rebalance Procedure

-*restDummy*: shows whether the current level is dummy or not
-*dbleRotation*: shows whether a left-right or right-left rotation has already appeared in any level or not
-leftNode & rightNode: are children of node in the current level

```
function Rebalance(node,leftNode,rightNode,$restDummy$,
$dbleRotation$)
    $balance$ = leftNode.$height$ − rightNode.$height$
    $cond1 = Osel(!restDummy\ \&\ balance > 1\ \&$ key <
        node.$leftKey, true, false)$                ▹ Left Left Rotate
    $cond2 = Osel(!restDummy\ \&\ balance < -1\ \&$ key >
        node.$rightKey, true, false)$               ▹ Right Right Rotate
    $cond3 = Osel(!restDummy\ \&\ balance > 1\ \&$ key >
        node.$leftKey, true, false)$                ▹ Left Right Rotate
    $cond4 = Osel(!restDummy\ \&\ balance < -1\ \&$ key <
        node.$rightKey, true, false)$               ▹ Right Left Rotate
    tmpKey = $Osel((!cond1\ \&\ !cond2\ \&\ cond3)$,
        leftNode.$rightKey$, dummy)
    tmpKey = $Osel((!cond1\ \&\ !cond2\ \&\ !cond3\ \&\ cond4)$,
        rightNode.$leftKey$, tmpKey)
    use cache to load AVL node with key tmpKey into
    leftRightNode or rightLeftNode based on $cond1$-$cond4$
    targetNode = $Osel((!cond1\ \&\ !cond2\ \&\ cond3)$,
        leftNode, dummy)
    targetNode = $Osel((!cond1\ \&\ !cond2\ \&\ !cond3\ \&$
        $cond4)$, rightNode, targetNode)
    opposNode = $Osel((!cond1\ \&\ !cond2\ \&\ cond3)$,
        leftRightNode, dummy)
    opposNode = $Osel((!cond1\ \&\ !cond2\ \&\ !cond3\ \&$
        $cond4)$, rightLeftNode, opposNode)
    $dummy = !(cond3 \mid cond4)$
    Rotate(targetNode, opposNode, $cond4$, $dummy$)
    update position of left/right nodes and their values in parent
    update position of leftRight.$leftPos$/rightLeft.$rightPos$ and
    their values in parent
    targetNode = $Osel((cond1 \mid cond2$ or $cond3 \mid cond4)$,
        node, dummy)
    opposNode = $Osel((cond1)$, leftNode, dummy)
    opposNode = $Osel((!cond1\ \&\ cond2)$, rightNode,
        opposNode)
    opposNode = $Osel((!cond1\ \&\ !cond2\ \&\ cond3)$,
        leftRightNode, opposNode)
    opposNode = $Osel((!cond1\ \&\ !cond2\ \&\ !cond3\ \&$
        $cond4)$, rightLeftNode, opposNode)
    $dummy = !(cond1 \mid cond2 \mid cond3 \mid cond4)$
    Rotate(targetNode, opposNode, $cond1 \mid cond3$, $dummy$)
    UpdateNodes()
end function
```

flags (*restDummy*, *dbleRotation*). *restDummy* is set by Insert procedure and shows whether the current level (and its corresponding node) is dummy or not. *dbleRotation* 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 procedure. Note 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.

Algorithm 10 GraphOS Add Operation for Vertex and Edge

```
function AddVertex($v$)
    DOMAP [("$V$"||$v$)] ← $(0, 0)$
end function
function AddEdge($v_{init}$,$v_{trm}$,weight)
    ($in_{init}$, $out_{init}$) ←DOMAP [("$V$"||$v_{init}$)]
    ($in_{trm}$, $out_{trm}$) ←DOMAP [("$V$"||$v_{trm}$)]
    $out_{init}$++ ; $in_{trm}$++
    DOMAP [("$V$"||$v_{init}$)] ← ($in_{init}$, $out_{init}$)
    DOMAP [("$V$"||$v_{trm}$)] ← ($in_{trm}$, $out_{trm}$)
    DOMAP [("$EOut$"||$v_{init}$, $out_{init}$)] ← ($v_{trm}$, weight)
    DOMAP [("$EIn$"||$v_{trm}$, $in_{trm}$)] ← ($v_{init}$, weight)
    DOMAP [("$E$", $v_{init}$, $v_{trm}$)] ← (weight, $out_{init}$, $in_{trm}$)
end function
```

Algorithm 11 GraphOS Remove Operation for Vertex/Edge

```
function RemoveEdge($v_{init}$,$v_{trm}$)
    ($cnt_{init}$, $cnt_{trm}$) ←DOMAP [("$E$", $v_{init}$, $v_{trm}$)]
    DOMAP [("$E$", $v_{init}$, $v_{trm}$)] ← NULL
    ($in_{init}$, $out_{init}$) ←DOMAP [("$V$"||$v_{init}$)]
    ($in_{trm}$, $out_{trm}$) ←DOMAP [("$V$"||$v_{trm}$)]
    DOMAP [("$EOut$"||$v_{init}$, $cnt_{init}$)] ←
                DOMAP [("$EOut$"||$v_{init}$, $out_{init}$)]
    DOMAP [("$EIn$"||$v_{trm}$, $cnt_{trm}$)] ←
                DOMAP [("$EIn$"||$v_{trm}$, $in_{trm}$)]
    $out_{init}$ ← $out_{init} - 1$ ; $in_{trm}$ ← $in_{trm} - 1$
    DOMAP [("$V$"||$v_{init}$)] ← ($in_{init}$, $out_{init}$)
    DOMAP [("$V$"||$v_{trm}$)] ← ($in_{trm}$, $out_{trm}$)
end function
function RemoveVertex($v$)
    ($in_v$, $out_v$) ←DOMAP [("$V$"||$v$)]
    DOMAP [("$V$"||$v$)] ← NULL
    for $i = 1$ to $out_v$ do
        trm ← DOMAP [("$EOut$"||$v$, $i$)]
        RemoveEdge ($v$, trm)
    end for
    for $i = 1$ to $in_v$ do
        init ← DOMAP [("$EIn$"||$v$, $i$)]
        RemoveEdge (init, $v$)
    end for
end function
```

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 *dbleRotation*).

The following theorem characterizes the security of Omix++.

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

Proof. To prove the security of Omix++, 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 $2 * \lceil 1.44 \cdot \log N \rceil$ 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

```
function EXECUTE(DOMAP, EList)
    Obliviously sort edges in EList based on their weights
    for $i = 1$ to $|V|$ do
        DOMAP $[(\text{“}Root\text{”}, i)] \leftarrow i$
    end for
    $i = 1$; st = 1
    for $j = 1$ to $2 * |E| * log|V|$ do
        $index \leftarrow Osel(i < (|E| * 2), i, |E| * 2)$
        $(init, trm, weit) \leftarrow$ EList[ceil($index/2$)]
        $vertex \leftarrow Osel($st $== 1$ & $i\%2 == 1, init, vertex)$
        $vertex \leftarrow Osel($st $== 1$ & $i\%2 == 0, trm, vertex)$
        $tmp$ =DOMAP $[(\text{“}Root\text{”}, vertex)]$
        $curRoot \leftarrow Osel($st $== 1, tmp, curRoot)$
        st $\leftarrow Osel(curRoot \neq vertex, 2, 1)$
        $tmp$ = DOMAP $[(\text{“}Root\text{”}, curRoot)]$
        $newRoot \leftarrow Osel($st $== 2, tmp, newRoot)$
        $mapKey \leftarrow Osel($st == 2, vertex, -1)
        DOMAP $[(\text{“}Root\text{”}, mapKey)] \leftarrow newRoot$
        $curRoot \leftarrow Osel($st $== 2, newRoot, curRoot)$
        $vertex \leftarrow Osel($st $== 2, curRoot, vertex)$
        $tmp \leftarrow$ DOMAP $[(\text{“}Root\text{”}, vertex)]$
        $curRoot \leftarrow Osel($st $== 2, tmp, curRoot)$
        $init_{root} \leftarrow Osel($st $== 1$ & $i\%2 == 1,$
                    $vertex, init_{root})$
        $trm_{root} \leftarrow Osel($st $== 1$ & $i\%2 == 0,$
                    $vertex, trm_{root})$
        $mapKey \leftarrow Osel($st $== 1$ & $i\%2 == 0$ &
                    $init_{root} \neq trm_{root}, init_{root}, -1)$
        DOMAP $[(\text{“}Root\text{”}, mapKey)] \leftarrow trm_{root}$
        $i \leftarrow Osel($st $== 1, i + 1, i)$
    end for
end function
```

Algorithm 14 Dijkstra Algorithm in GraphOS

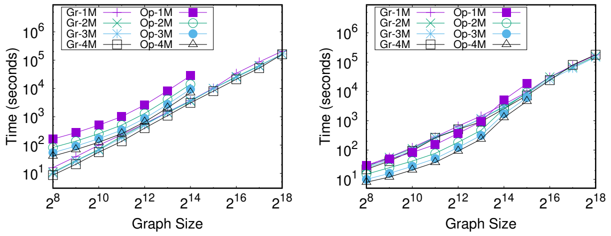
```
function EXECUTE(DOMAP, $start$)
    MinHeap $\leftarrow$ ObliviousMinHeap.Initialize($|V|$)
    for $i = 1$ to $|V|$ do
        DOMAP $(\text{“}Dist\text{”}, i) \leftarrow \infty$
    end for
    DOMAP $(\text{“}Dist\text{”}, start) \leftarrow 0$
    MinHeap.AddNewNode($start$, 0)
    innerLoop $\leftarrow$ false ; $u = -1$ ; distu = $-1$
    cnt = 1 ; weit = $-1$
    for $i = 1$ to $2 * |V| + |E|$ do
        $mapKey \leftarrow Osel($innerLoop$, v, dummy)$
        $u \leftarrow Osel($innerLoop$, u, -1)$
        distu $\leftarrow Osel($innerLoop, curDistu, $-1)$
        $tmp \leftarrow$ DOMAP $(\text{“}Dist\text{”}, mapKey)$
        distv $\leftarrow Osel($innerLoop$, tmp,$ distv$)$
        $mapValue \leftarrow Osel($innerLoop & distu + weit < distv,
                                distu + weit, distv)
        DOMAP $(\text{“}Dist\text{”}, mapKey) \leftarrow mapValue$
        $OP \leftarrow Osel($innerLoop $== false, EXTRACT\_MIN,$
                                $dummy\,)$
        $OP \leftarrow Osel($innerLoop & distu + weit < distv,
                                $ADD\_NODE, OP)$
        $(u,$ distu$) \leftarrow$ MinHeap.execute($OP$,$v$,distu + weit)
        cnt $\leftarrow Osel($innerLoop, cnt + 1 , cnt)
        $mapKey \leftarrow Osel($innerLoop $== false$ & $u \neq -1,$
                            $(\text{“}Dist\text{”}, ++u), dummy)$
        $tmp \leftarrow$ DOMAP $(mapKey)$
        curDistu $\leftarrow Osel($innerLoop $== false$ & $u \neq -1,$
                                $tmp,$ curDistu)
```

Algorithm 12 BFS Algorithm in GraphOS

```
function EXECUTE(DOMAP, start)
        ▷ BFS-specific parts are in red and DFS-specific ones in blue
    Qcnt = 1; curQCnt = 1; source $\leftarrow start$
    DOMAP $[(\text{“}InQ\text{”},$ Qcnt$)] \leftarrow start$
    DOMAP $[(\text{“}Visited\text{”},$ source$)] \leftarrow true$
    Qcnt $\leftarrow$ Qcnt + 1 ; Qcnt $\leftarrow$ Qcnt
    outerL = $true$
    while curQCnt $\neq$ Qcnt; Qcnt $\neq 0$ do
        tmp $\leftarrow$ DOMAP $[(\text{“}InQ\text{”},$ curQCnt; Qcnt$)]$
        source $\leftarrow Osel($outerL, tmp, source$)$
        curQCnt $\leftarrow Osel($outerL, curQCnt + 1, curQCnt$)$
        Qcnt $\leftarrow Osel($outerL, Qcnt $-$ 1, Qcnt$)$
        cnt $\leftarrow Osel($outerL, 1, cnt$)$
        trm $\leftarrow$ DOMAP $[(\text{“}EOut\text{”},$ source, cnt$)]$
        outerL $\leftarrow Osel($trm $== NULL, true, false)$
        tmp $\leftarrow$ DOMAP $[(\text{“}Visited\text{”},$ trm$)]$
        visited $\leftarrow Osel($outerL, visited, tmp$)$
        mostInner $\leftarrow$ visited = $NULL$ & outerL = $false$
        tmp $\leftarrow Osel($mostInner, trm$, dummy)$
        DOMAP $[(\text{“}InQ\text{”},$ Qcnt; Qcnt + 1$)] \leftarrow$ tmp
        tmp $\leftarrow Osel($mostInner$, true, dummy)$
        DOMAP $[(\text{“}Visited\text{”},$ trm$)] \leftarrow$ tmp
        Qcnt $\leftarrow Osel($mostInner, Qcnt + 1, Qcnt$)$
        cnt $\leftarrow Osel($outerL, cnt, cnt + 1$)$
    end while
end function
```

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 2^{α} smaller



(a)

(b)

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 α ($\leq \log\log N$), 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× 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× 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.