# **Distributed & Scalable Oblivious Sorting and Shuffling**

Nicholas Ngai UC Berkeley nicholas.ngai@berkeley.edu Ioannis Demertzis UC Santa Cruz idemertz@ucsc.edu Javad Ghareh Chamani HKUST jgc@cse.ust.hk Dimitrios Papadopoulos HKUST dipapado@cse.ust.hk

Abstract—Existing oblivious systems offer robust security by concealing memory access patterns, but they encounter significant scalability and performance challenges. Recent efforts to enhance the practicality of these systems involve embedding oblivious computation, e.g., oblivious sorting and shuffling, within Trusted Execution Environments (TEEs). For instance, oblivious sort has been heavily utilized: in Oblix (S&P'18), when oblivious indexes are created and accessed; in Snoopy's highthroughput oblivious key-value (SOSP'21) during initialization and when the input requests are deduplicated and prepared for delivery; in Opaque (NSDI'17) for all the proposed oblivious SQL operators; in the state-of-the-art non-foreign key oblivious join approach (PVLDB'20). Additionally, oblivious sort/shuffle find applications in Signal's commercial solution for contact discovery, anonymous Google's Key Transparency, Searchable Encryption, software monitoring, and differentially private federated learning with user privacy.

In this work, we address the scalability bottleneck of oblivious sort and shuffle by re-designing these approaches to achieve high efficiency in distributed multi-enclave environments. First, we propose a multi-threaded bitonic sort optimized for the distributed setting, making it the most performant oblivious sort for small number of enclaves (up to 4). For larger numbers of enclaves, we propose a novel oblivious bucket sort, which improves data locality and network consumption and outperforms our optimized distributed bitonic-sort by up to 5- $6 \times$ . To the best of our knowledge, these are the first distributed oblivious TEE-based sorting solutions. For reference, we are able to sort 2 GiB of data in 1 second and 128 GiB in 53.4 seconds in a multi-enclave test. A fundamental building block of our oblivious bucket-sort is an oblivious shuffle that improves the prior state-of-the-art result (CCS'22) by up to  $9.5 \times$  in the distributed multi-enclave setting-interestingly it is better by 10% even in the single-enclave/multi-thread setting.

Index Terms-oblivious sorting, shuffling, distributed, scalable

## 1. Introduction

Organizations outsource sensitive data to the cloud for convenience, cost-efficiency, and availability. Encryption alone may not fully protect data, as the user's access pattern can leak sensitive information [1]–[6]. For instance, a doctor's frequent access to a medication database could inadvertently reveal a patient's diagnosis. Recent leakage-abuse attacks [7]–[19] highlight the necessity of protecting memory access patterns. Hence, balancing cloud benefits and high levels of privacy becomes crucial for secure data management.

Trusted execution environments (TEEs) present a unique opportunity for privacy-preserving computation with high utility. However, relying solely on trusted hardware is not only insufficient to protect memory access patterns [20], [21], but also enclave side-channel attacks (e.g., Meltdown [22] and Spectre [23]) can lead to the extraction of enclave secrets. *Obliviousness* is a strong property that can mitigate access pattern and side channel leakages—obliviousness ensures that algorithms within trusted hardware are data-input independent and indistinguishable by powerful adversaries for any two input instances of the same size. By combining cloud outsourcing with correctly designed and implemented oblivious TEEs, organizations can securely conduct privacypreserving computation, mitigate sensitive data leakage risks, and uphold data confidentiality and integrity.

Single vs Double/Full Obliviousness. "Standard" oblivious approaches [24]–[29], operate in a model that assumes the client-side routines are executed in a fully trusted environment, typically in a machine fully controlled by the client. Therefore, they only focus on providing obliviousness on the server side (hence single-oblivious), and they do not need to worry about protecting the data privacy against an adversary who can observe the access pattern of the memory on the client side. For TEE-based solutions, where we do not want the client to be actively involved in the computation. all the oblivious routines (including the client ones) are executed on the server side requiring all server and client routines to be oblivious. Double-obliviousness (as introduced by [30], or *full-obliviousness* (as introduced in [31]) protects data privacy even against an adversary that can observe the memory access pattern imposed by the client routines, requiring both the client and server routines to be oblivious.

**Fully/Doubly-Oblivious TEE-based approaches.** Mishra et al. [30] introduced Oblix, a suite of Doubly-Oblivious ORAM (doubly-oblivious memory) and Data Structures (ODS) schemes tailored for TEEs. [32], [33] focused on eliminating memory access leakage when executing arbitrary programs in TEEs by loading both code and data onto doubly-oblivious memory. Zheng et al. [34] presented Opaque, an oblivious distributed data analytics platform offering a wide range of SQL

functionality. Krastnikov et al. [35] proposed the first efficient doubly-oblivious non-foreign-key join schemes. The recent work of Dauterman et al. [36] introduced Snoopy, the first high-throughput and scalable oblivious key-value store. The closest to ours work, Sasy et al. [31] introduced new efficient multi-thread approaches for shuffle and compaction.<u>Below, we use the term</u> **"obliviousness" for TEEs** to refer to **doubly/fully obliviousness**, unless stated otherwise.

Oblivious primitives, like oblivious sort and shuffle, form the foundation of oblivious computation, whether combined with Trusted Execution Environments (TEEs) or not. In the case of Oblivious-TEEs, Oblix [30] relies on logarithmic number of heavy oblivious sorts for creating and initializing doubly-oblivious memory and data structures. Oblivious sort is also used during the oblivious accesses. Similarly, Snoopy [36] requires a heavy obliviously-sort operation for initialization to sort all input data and distribute them in sub-ORAMs. During oblivious accesses, the loadbalancer needs to obliviously sort input requests twice. In Opaque [34], all oblivious query operators (obliviousfilter, oblivious aggregation, oblivious foreign-key join) are sort-based, necessitating multiple sorts. Furthermore, the optimized (but sequential) non-foreign key join by Krastnikov et al. [35] requires multiple sort operations, which take 96% of the total processing time of 12.8s, when joining two tables with 1M records each and 8 bytes per tuple.

**Applications.** Below, we provide broader applications where secure TEE-based approaches heavily rely on oblivious primitives, either directly or as part of existing oblivious TEE-based solutions [30], [31], [34]–[36].

• Private contact discovery for Signal: Signal [37] is an encrypted messaging system. It recently added a service for private contact discovery, i.e., users never provide Signal with plaintext access to their contacts. Signal's approach is based on a combination of Oblix [30] and Snoopy [36].

• Anonymizing Google's Key Transparency: Google's Key Transparency [38], [39] allows users to discover public keys of other users while ensuring the integrity of the retrieved key. However, the service doesn't provide anonymity, and the server learns the users' queries. To address this, Snoopy [36] can be used to provide user privacy.

Searchable encryption: Searchable encryption (SE) [40]–[50] enables efficient searching over encrypted out-sourced data without decryption. Recent works utilize TEEs to enhance efficiency [51] relying on ODS like Oblix/Omix++ [30], [52] and oblivious sort. State-of-the-art Dynamic SE (DSE) that achieve forward and backward privacy rely on ODS and oblivious sorting [44], [53], [54]
 Oblivious distributed data analytics/Oblivious Databases: As discussed above, Opaque [34] is the state-the-art solution for oblivious analytics and it solely relies on oblivious sort. It shows superiority over other approaches on three types of workloads: SQL, machine learning, and graph analytics.

• Private Sampling-based Query Frameworks (PSQF): PSQF [55] guarantees confidentiality, integrity, and anonymity for user data in the federated learning context. This is accomplished by combining differentially private queries, sampling, (oblivious) random shuffling, and TEEs. • Large-scale monitoring of software activities: Prochlo, designed by Bittau et al. [56], facilitates large-scale monitoring of software activities, including telemetry, error reporting, and demographic profiling. Integrating TEEs with oblivious shuffling provides robust privacy guarantees for user data.

Despite these numerous real-world applications, the vast majority of research works for TEE-based oblivious primitives focus only on the centralized, single-machine server scenario. This is in direct contradiction to common practices from the industry where truly large-scale problems are tackled by distributing and parallelizing the workload across multiple workers (e.g., Spark [57]/MapReduce [58] environments). When considering TEE-based solutions, distributed solutions become even more useful, as TEE enclaves usually have a limited amount of protected memory available after which costly paging kicks in, harming performance (e.g., Intel-SGX's EPC is limited to 128 MB; recent developments with SGXv2 scale this to GiBs of memory, however, similar paging techniques hamper performance at higher usage percentages [59]). Hence, developing distributed, cross-enclave, oblivious solutions becomes a necessity to scale to big-data instances. When also factoring in budget considerations, it is more favorable to consider systems that can be distributed across multiple "cheaper" machines than ones that only focus on multiple enclaves on the same "large" platform.

Specifically focusing on the most fundamental of the primitives discussed so far, oblivious sorting and shuffling, to the best of our knowledge no previous work considers fully distributed and oblivious solutions<sup>1</sup>. Focusing on single-server solutions, bitonic sort [60] is the practical and preferred choice, despite its suboptimal complexity. As far as we know, for TEEs the multi-thread implementation of bitonic sort in Snoopy [36] is the most performant available option known; as for oblivious shuffling, the recent work of Sasy et al. [31] offers the state-of-the-art and multi-thread implementation. Again, neither of these works provides a multi-enclave solution, hence they do not and cannot scale to large instances; e.g., [31] provides experimental results for up to only 4 GiB of data (for comparison, our evaluation includes results for up to 128 GiB).

Based on the above, the main question we ask is:

# Can we design scalable/high-performant distributed oblivious primitives for sorting and shuffling?

**Our contribution.** In this work, we address the scalability bottleneck of oblivious sort and shuffle by re-designing approaches that achieve high efficiency in distributed multienclave environments. Additionally, we also consider performance on multi-core single-machine environments. Hence, our solutions comprise state-of-the-art TEE-based oblivious primitives. In particular:

<sup>1.</sup> Opaque's distributed sort combines column sort and bitonic sort within the same threat model as ours. While column sort facilitates the desired distribution of the sort computation (in combination with Spark), it requires several extra passes and increased communication between enclaves compared to our more flexible MPI-based distributed bitonic and bucket sort approaches.

1. We extend bitonic sort [60] to the distributed setting, and we use chunking of swap operations—sending multiple elements at a time—to mitigate the high overhead of network communication and latency. As a baseline, this results in  $30 \times$  improvement over the naïve approach of swapping elements one at a time across the network.

2. Unfortunately, bitonic sort (even after the chunking optimization) still demonstrates high overhead network latencies and poor data locality. Targeting better solutions for the distributed setting, we focus on Asharov et al.'s bucket oblivious sort [61], which is based on the idea of first applying an oblivious random permutation (ORP, also called oblivious sourt. Our distributed bucket sort, called DBUCKET, utilizes a new structure for the bucket routing network which improves the data locality and communication overhead of [61]'s ORP to O(N). Our bucket ORP, including the oblivious compaction optimization of Sasy et al. [31], gives up to  $9.52 \times$  speedup in a 64-enclave setting compared to the state-of-the-art [31] random shuffling.

**3.** We similarly extended our distributed bucket ORP to distributed bucket oblivious sort. By leveraging a comparison-based sort that has a similar  $O(N \log N)$  runtime and O(N) communications cost, the result is an asymptotically optimal oblivious sort in the distributed setting which is up to  $6.63 \times$  faster than the bitonic sort with 64 enclaves.

**4.** Orthogonal to our algorithmic and implementation-level improvements to the above algorithms, we additionally describe efficient oblivious primitives that leverage low-level assembly instructions and greatly outperform the standard bit-manipulation-based oblivious comparison and CMOV-based oblivious swap, resulting in an up to  $1.71 \times$  speedup due to these assembly optimizations alone.

**5.** We design and implement an encrypted MPI layer that provides confidentiality and integrity even against network adversaries that can replay messages, by utilizing the attested TLS feature provided by TEEs to derive shared secrets between enclaves used to encrypt and authenticate messages. We additionally implement a sliding window design inspired by DTLS [62] and ensure unique pairwise keys to efficiently achieve replay resistance against network adversaries.

6. We discussed above that our new scalable oblivious sorting and shuffling techniques significantly can impact prior oblivious TEE-based approaches by replacing their oblivious primitives with ours. Surprisingly, we found that some prior choices for these TEE-based approaches were made due to the lack of scalable oblivious primitives. For instance, Snoopy's [36] multiple load balancers and the number of sub-ORAMs were chosen to avoid heavy oblivious sort operations. However, with our scalable oblivious primitives, many of their prior choices are suboptimal. In our experimental evaluation, we propose Snoopy++, a simplified version of Snoopy, which outperforms Snoopy while requiring fewer resources. Snoopy++ utilizes one oblivious sort, one sequential scan, and one oblivious compaction on the input data and the collected requests. This highlights the efficiency and superiority of our scalable oblivious techniques in practical

implementations. While Snoopy, with a database size of 16M 128-B elements, is only able to reach a throughput of  $\sim$ 4,000 lookup requests per second with an average latency of 1,000 ms with 32 enclaves, Snoopy++ is able to process over 700,000 lookup requests per second using the same hardware within 0.86 seconds.

Limitations. We acknowledge that relying on a trusted enclave may be a mitigating factor for all works in this area, however, the high level of achieved security may be necessary for certain applications. It is also worth noting that our solution is not dependent explicitly on a particular TEE vendor (e.g., Intel-SGX), but can be based on any trusted enclave that meets the required properties (e.g., AMD Enclave and ARM TrustZone). Our implementation specifically targets SGX but is built on OpenEnclave SDK<sup>2</sup>, which is a hardware-agnostic library that currently offers "preview" support for other alternatives (e.g., ARM Trustzone). We have not evaluated the development effort to switch to another TEE but we believe the locality-aware design of our algorithms can be beneficial for distributed settings regardless of the used TEE. Finally, our proposed approach is also aligned with recent efforts of making trusted hardware robust against side-channel attacks (e.g., Keystone project [63]). Finally, we want to highlight that our distributed oblivious sorting and shuffling algorithms can be of independent interest for any distributed solution even without TEEs.

# 2. Preliminaries

**Secure Enclaves.** Our proposed oblivious algorithms can be implemented using any TEE that provides isolation, sealing, and remote attestation, such as Intel-SGX [64], AMD enclave [65], or ARM [66] TrustZone. As a proof of concept, we implemented our solution using Intel-SGX [64]. In Intel-SGX, isolation is achieved by reserving a portion of the system's memory, known as the Enclave Page Cache (EPC), to store the user's code and data and maintain it in encrypted form. The total size of the EPC memory of SGXv2 is in the order of tens/hundreds of GiBs. Sealing allows the enclave to persistently store its data outside the secure environment, while remote attestation ensures the integrity of the code.

**Obliviousness.** A memory/algorithm/data-structure is oblivious if and only if for any two same-sized sequences of polynomially many operations, their resulting access patterns (i.e., the sequence of memory accesses while performing the operations) is computationally indistinguishable for anyone but the client, assuming an honest but curious adversary that sees all memory accesses and network communications. In the case of TEE-based solutions, the adversary can observe both the clients' and servers' memory access patterns (full/double obliviousness [30], [55]).

**Oblivious Compaction.** Compaction takes as input an array where each element is either "marked" or "unmarked" and outputs a permutation of the array such that all marked elements are ordered at the beginning of the array, followed

<sup>2.</sup> https://github.com/openenclave/openenclave

by all unmarked elements. While an oblivious sort may be used to trivially implement oblivious compaction by sorting on the marked bit, it is much more efficient to utilize explicit oblivious compaction due to the relaxed constraints of the latter. All known oblivious compaction operators are constructed as compaction networks, using a deterministic set of oblivious conditional swaps to achieve oblivious compaction. For reference, we provide the high-level compaction algorithm (ORCompact) of the state-of-the-art work, Sasy et al. [31]. ORCOMPACT utilizes a recursive helper function, OROFFSETSET(A, z), which compacts marked elements in A, of length N, to an offset z.

- 1) OROFFCOMPACT(A, z): Let m be the count of in the left half of A,  $A_{\text{left}}$ . Then, recursively call OROFFCOMPACT  $(A_{\text{left}}, z)$  and ORCOMPACT  $(A_{\text{right}}, z + m)$ . Finally, obliviously swap  $A_{\text{left}}[i]$  and  $A_{\text{right}}[i]$  for  $i \in \{1, ..., \frac{N}{2}\}$ , conditioned on  $(z \mod \frac{N}{2} + m) \oplus (z \ge N) \oplus (i \ge (z + m) \mod \frac{N}{2})$ .
- 2) ORCOMPACT(A): Call OROFFCOMPACT(A, 0).

This is a compaction network with O(N) oblivious swaps for  $O(\log N)$  recursive layers, so the running time of ORCompact is  $O(N \log N)$ . We refer the reader to the original paper for more details.

**Oblivious Random Permutation.** Oblivious random permutation (ORP, also known as oblivious shuffling) is an oblivious algorithm that results in a random permutation of elements in an array. While it is relatively trivial to assign a random key to each element in the array and apply an oblivious sort based on the random keys in order to achieve ORP, this approach comes with overhead that can be avoided by using a "native" ORP algorithm, as observed in [31].

The current state-of-the-art ORP algorithm in a nondistributed setting is Sasy et al.'s ORShuffle (Oblivious Recursive Shuffle) algorithm [31], which utilizes the ORCompact algorithm by the same authors as a building block:

- 1) Mark exactly half of the elements in A.
- 2) Call ORCOMPACT(A) to compact a random half of the elements into the left half of the array and the other half of the elements to the right half.
- 3) Recursively call ORSHUFFLE  $(A_{left})$  and ORSHUFFLE  $(A_{right})$ .

ORShuffle performs  $O(\log N)$  invocations of ORCompact, so its running time is  $O(N \log^2 N)$ .

Bucket ORP, introduced by Asharov et al. [61], takes a slightly different approach. By expanding the input array into an array of size 2N, padding with N dummy elements, and assigning each element a random key, it can achieve a sort based on the randomly assigned keys using a bucket routing network with only  $O(\log N)$  layers, leveraging the uniformly randomly distribution of the generated keys. At a high level, bucket ORP starts with an oblivious random bin assignment step, using randomly generated keys to assign each element to a bucket and routing elements towards that bucket using a butterfly routing network, followed by a bucket permutation

step to randomly permute the elements within each bucket and removing the dummy elements.

Let  $A_j^{(i)}$  refer to the *j*'th bucket of elements of size Z in layer *i* of the routing network. Z is a configurable security parameter. Bucket ORP uses a helper function  $(A'_0, A'_1) \leftarrow$ MERGESPLIT  $(A_0, A_1, i)$ , which takes as input two buckets  $A_0$  and  $A_1$  and outputs buckets  $A'_0$  and  $A'_1$ , such that all elements with a 0 in the *i*'th bit of the random key are in  $A'_0$ and all elements with a 1 in the *i*'th bit of the random key are in  $A'_1$ . This MERGESPLIT can be implemented using a bitonic sort in time  $O(Z \log^2 Z)$ .

- 1) Generate a random key and assign it to each element.
- 3) At layer  $\log E$ , bucket  $A_j^{(\log E)}$  contains all elements with keys starting with the bitstring *j*. Obliviously sort each bucket according to the remaining bits in the random key to complete the ORP.

Because there are  $O(\log N)$  layers, with  $O\left(\frac{N}{Z}\right)$ MERGESPLIT operations per layer, each of complexity  $O(Z \log Z)$ , the total complexity of this sort is  $O(N \log N \log Z)$ . For a fixed security parameter Z (Z = 512 in our implementation, as suggested in [61]), the running time of bucket ORP is  $O(N \log N)$ .

Oblivious Sort. There are two main approaches for oblivious sort. The first constructs an oblivious sort by following a predefined sequence of oblivious swaps, known as a sorting network. Batcher's bitonic sort [60] is the most commonly used such algorithm. It sorts two halves of the array recursively, one in reverse order, resulting in a bitonic sequence comprising an increasing half and a decreasing half of the array. This is followed by a bitonic merge, i.e., a deterministic sequence of swaps that sorts the entire sequence with a  $O(N \log N)$  complexity, for N elements. The running time of bitonic sort is  $O(N \log^2 N)$ , which matches the complexity of most practical sorting networks, including Batcher's odd-even merge sort. Although there are theoretical constructions that achieve  $O(N \log N)$  complexity, such as the AKS [67] sorting network, large constant factors make them impractical for reasonable workloads. The second approach, known as the bucket oblivious sort [61], builds on bucket ORP, pairing it with an efficient non-oblivious comparison-based sort, to achieve  $O(N \log N)$  time. Since the behavior of any comparison-based sort depends only on the initial order of elements, the ORP step makes the non-oblivious sort oblivious to the original input data.

**Threat Model.** We adopt a similar threat model as the one proposed by prior works that combine oblivious primitives with trusted hardware, such as [30], [31], [34]. We assume an attacker who can observe all memory accesses and has

control over the server's software stack and operating system. However, the attacker cannot steal any information from the secure processor, including the processor's secret keys. Moreover, the attacker cannot access the plaintext values of data and code loaded in the secure processor's protected enclave portion of memory, although they can observe the accessed memory locations<sup>3</sup>. The protected memory is encrypted using the processor's secret key. We also consider any enclave side-channel leakage, rollback attacks [68], and denial-of-service attacks to be outside the scope of our work, following the practices of previous works. Examples of such side-channel leakage include cache-timing, power analysis, or other timing attacks [69]–[74]. However, there are several complementary techniques, such as [32], [33], [75]–[79], that can potentially mitigate these attacks.

While we take care to ensure the obliviousness of source code, we make no guarantees that the compiler has preserved full obliviousness across the entire program in the final enclave image. Assembly-level obliviousness verifiers are a more recently introduced work [31] which may be used to achieve higher confidence of the final obliviousness.

#### 3. Distributed Bitonic Sort

Here we explain the instantiation of our distributed bitonic sort. To the best of our knowledge, ours is the first instantiation of bitonic in the distributed setting across multiple TEE enclaves, hence we need to address certain challenges to improve its practical performance. As a starting baseline for the oblivious sort, we implement Batcher's bitonic sort [60] using the standard recursive algorithm in a distributed setting. Each enclave is responsible for a slice of the total data set, and swaps for elements in different enclaves are implemented by the two elements' enclaves exchanging elements and using an oblivious assignment on both enclaves to choose the correct element.

**Choice of Sorting Network.** Of note, we did not choose Batcher's odd-even mergesort [60], despite its constant-factor improvement in the number of swaps and an equivalent level of algorithmic parallelizability. This was because it presents performance issues in a distributed setting when swaps across two enclaves incur network overhead, which can be orders of magnitude slower than simple enclave memory accesses. The merge phase of the odd-even mergesort recursively subdivides the elements into odd and even slices, which, by nature, remain spread across every single enclave in every recursive level. In a setting with N elements and E enclaves, this means each one of the  $\log N$  levels of the recursive sort and merge steps incurs swaps across different enclaves, as illustrated in Figure 1a.

3. We assume the adversary may even observe accesses within a single page or within a single cache line. Other works have noted that such sub-cache-line-memory bus attacks have not been observed in the real world [31], and an adversary tapping into the memory bus in the real world may not be able to observe accesses below the level of a cache line, but we adopt a liberal view of the adversary's view of access patterns to enclave memory anyway to obey the strictest notions of obliviousness.

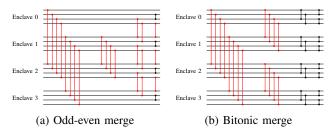


Figure 1: Recursive merge of an odd-even merge sorter and a bitonic merge sorter. For odd-even merge, all  $O(\log N)$  levels have at least one swap across two enclaves. For bitonic merge, after  $O(\log E)$  levels, all swaps happen within a single enclave.

By contrast, sort and merge phases of the bitonic sort both recursively subdivide elements into left and right *halves* meaning that after only  $O(\log E)$  levels, all swaps happen within a single enclave, improving the overall communications from  $O(\log N)$  to  $O(\log E)$ , as shown in Figure 1b.

Mitigating Network Latency through Swap Chunking. A naïve implementation of the bitonic sort in a distributed setting swaps elements across enclaves one at a time, incurring a full round trip of communication for each element, resulting in a total of  $O(N \log^2 E)$  round trips across the network. Because the sequence of swaps is deterministic, however, the sequence of swaps is fully predictable, and it is easy to predict what sets of elements will need to be sent to a given enclave. The bitonic sort typically swaps consecutive runs of adjacent elements, we optimize our bitonic sort baseline by sending chunks of up to a configurable parameter C elements at a time to the remote enclave for a swap of elements across enclaves, which reduces the number of round trips around the network the enclaves must wait by a factor of C. Note that for C = 1, this approach is equivalent to simply sending elements one at a time. This decreases the effect of network latency and increases throughput, since enclaves can send a chunk of elements all at once before waiting to receive the remote enclave's chunk of elements.

**Bitonic Sort Shortcomings.** Ultimately, while it is not difficult to achieve a near-optimal implementation of the bitonic sort in a distributed setting, several inherent factors of sorting networks render their use non-ideal in this setting. First, it is difficult to avoid a communications overhead of  $O(N \log^2 E)$  with sorting networks, due to their deterministic nature necessitating several swaps before an element's position may be narrowed down to within a single enclave. In non-distributed settings, the cost of swapping is generally low, and this is a worthwhile tradeoff to make for bitonic sort's high parallelizability. In the distributed setting, however, cross-enclave swaps quickly become the performance bottleneck.

#### 4. Distributed Bucket Oblivious Sort

We now describe in detail our distributed sorting algorithm, motivated by the shortcomings of the distributed bitonic sort that we identified above. Our algorithm, which

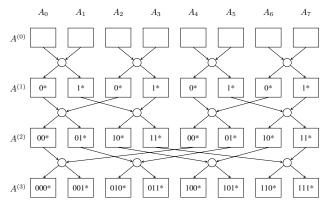


Figure 2: Original MERGESPLIT butterfly network from [61] implementing oblivious random bin assignment. Each MERGESPLIT writes to buckets in layer  $A^{(i+1)}$  that are different from those in layer  $A^{(i)}$ . At layer *i*, the elements are sorted by the *i most* significant bits of the random key.

we call DBUCKET sort, is an extension and variation of the bucket oblivious sort described by Asharov et al. [61], in order to optimize its behavior in a distributed setting. Bucket sort is fundamentally an algorithm with two parts: an ORP step followed by a non-oblivious, comparisonbased sorting step. Both these steps provide us with an opportunity to optimize their behavior for the distributed setting. Hence, DBUCKET has a similar two-step structure: a distributed bucket ORP (DBUCKET ORP) followed by an efficient, non-oblivious, comparison-based sort. First, we briefly mention an optimization originally described by Sasy et al. [31], that uses ORCompact instead of bitonic sort for the implementation of MERGESPLIT. Then we explain in detail our proposed optimizations for the distributed setting, focusing on expediting low-level memory behavior and asymptotically reducing inter-enclave communication.

**MERGESPLIT** Using Oblivious Compaction. The MERGESPLIT operation requires an algorithm that obliviously places all "marked" elements in the first output bucket and all "unmarked" elements in the second output bucket. Asharov et al.'s original bucket oblivious sort uses bitonic sort for this purpose. Sasy et al. [31] describes the use of oblivious recursive compaction as a drop-in replacement to sort all marked elements into the first output bucket and unmarked elements into the second one. This represents an improvement of  $O(Z \log^2 Z)$  to  $O(Z \log Z)$  for the MERGESPLITOperation; we adopt this optimization here.

**Memory-Friendly Oblivious Random Bin Assignment.** Figure 2 shows the original the MERGESPLIT butterfly network from [61]. Notice that indices of the buckets taken as inputs into the MERGESPLIT operation are different from indices of the output buckets. Namely, the *j*'th group of buckets in the *i*'th level of the network calls

$$\left(A_{2j}^{(i+1)}, A_{2j+1}^{(i+1)}\right) \leftarrow \operatorname{MergeSplit}\left(A_{j'+j}^{(i)}, A_{j'+j+2^i}^{(i)}, i\right),$$

where  $j' = \lfloor \frac{j}{2^i} \rfloor \cdot 2^i$ . Thus, we must use an additional buffer of size N in which output buckets are written because output

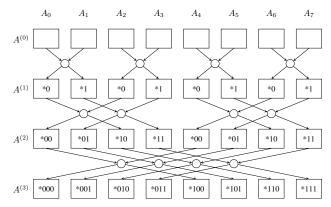


Figure 3: Our modified MERGESPLIT butterfly network implementing oblivious bin assignment. Each MERGESPLIT operation writes to the same buckets in layer  $A^{(i+1)}$  as in layer  $A^{(i)}$ . At layer *i*, the elements are sorted by the *i least* significant bits of the random key.

indices do not directly match input ones.

We eliminate this overhead with the following simple modification: The outputs are instead written to  $(A_{j'+j}^{(i+1)}, A_{j'+j+2i}^{(i+1)})$ . Because the input and output bucket indices are the same, we are able to perform the MERGESPLIT operation fully *in-place*. Since the input buckets in  $A^{(i)}$  are no longer needed once their corresponding buckets in  $A^{(i+1)}$  are produced, we can reuse the same buffer of memory for both the input and the output. Figure 3 shows the resulting butterfly network after our modification. Clearly, since the mapping is still deterministic, this does not affect the oblivious property of the algorithm, as we elaborate next.

The original construction results in elements in  $A^{(i)}$  semisorted by the *i* most significant bits of the ORP key. To maintain a similar property, our modified network performs MERGESPLIT according to the *i*'th *least* significant bit. Notice that the effect of this modification is that elements at layer *i* of the butterfly network are sorted by the *i least* significant bits of the of their ORP keys, rather than the *i most* significant bits. Because the ORP keys are random, it is clear that this modified butterfly network maintains the correctness of the random bin assignment and ORP, which in turn maintains the obliviousness of the subsequent comparison-based sorting algorithm.

Linear-Communication Distributed Oblivious Random Bin Assignment. When focusing on the distributed setting, a potentially major performance bottleneck is communication across enclaves. In an enclave-protected setting, all communications between different enclaves must additionally be encrypted and authenticated before being transmitted over the network, to defend against the adversary's ability to read and modify network traffic. While some communication is obviously necessitated for distributed ORP, it is crucial to minimize it, to the extent that this is possible.

Along these lines, one clear observation from Figure 3 is that any level of MERGESPLIT operation that occurs across enclaves incurs communication of size O(N). For a cluster

comprised of E enclaves, the butterfly network must swap elements across enclaves for the first  $O(\log E)$  layers, for a total of  $O(N \log E)$  communications cost taken by the oblivious random bin assignment operation, which can be a significant overhead in practice.

Next, we describe an optimized bucket ORP that reduces this communication cost down to O(N)! At a high level, we leverage the fact that we can predict the final position of any given bucket from the first log E bits of the ORP key, which is produced after the first log E layers of the bucket routing network. By rearranging the positions of buckets after the first log E layers, we can ensure that the remaining log(N/E) layers all occur *locally within an enclave*. This rearrangement step requires only O(N) communication, representing an asymptotic improvement over the  $O(N \log E)$  cost of the original butterfly network design.

In more detail, the DBUCKET ORP's oblivious random bin assignment now consists of the following three steps:

- Run log E layers of the MERGESPLIT butterfly network. This results in the buckets in layer A<sup>(log E-1)</sup> being semi-sorted by the least log E bits of the ORP key—in other words, the elements in bucket A<sup>(log E-1)</sup> satisfy orpkey mod E = j mod E.
   Route buckets to their final destination enlaye by
- Route buckets to their final destination enclave by rearranging the buckets of layer A<sup>(log E)</sup>. Precisely, bucket A<sup>(log E-1)</sup> is stored in any bucket in enclave j mod E. We will refer to the arrangement of buckets in this layer after this step as A'<sup>(0)</sup>.
- 3) Perform the remaining  $\log(N/E)$  layers of the MERGESPLIT butterfly network with input layers  $A'^{(0)}$  to  $A'^{(\log(N/E))}$  and ignoring the first *E* bits of the ORP key. In other words, the butterfly network is "restarted" with  $A'^{(0)}$  as the initial state. Because only  $\log(N/E)$  layers are performed, all MERGESPLIT operations will be performed locally, without incurring any communications cost.

For steps (1) and (3), all MERGESPLIT operations are performed locally, without incurring any communication  $\cos t^4$  and step (2) needs just O(N) communications cost. A visualization of this new, three-step butterfly network, with the additional rearranging step, is given in Figure 4.

**Distributed Non-Oblivious Sort.** After the ORP step, the remaining step is to sort the array according to the sort key of each element using a standard, non-oblivious sorting algorithm. The choice of algorithm is left to the implementer's discretion—the only fundamental constraint is that it must be comparison-based since its behavior will depend on the order of the randomly permuted elements. The "classic" idea in distributed sorting of partitioning data according to the distribution of the sort keys will not work here, since those partitioning schemes algorithms are not comparison-based and may leak information about the distribution of sort keys. Instead, we rely on the guaranteed random distribution of

inputs to the non-oblivious sorting algorithm to implement an efficient *samplesort*:

- 1) Locally sort the elements in enclave 0 with a comparison-based sorting algorithm.
- 2) For  $i \in [1, ..., E 1]$  use the key in  $\frac{iN}{E^2}$  th element in enclave 0 as the key for partitioning elements across enclaves. Broadcast the chosen keys to all other enclaves.
- Use Quickselect [80] in all other enclaves to quickly partition elements based on the received partitions.
- 4) All enclaves send and receive the partitioned elements to each other. This guarantees that elements in enclave *i* are strictly greater than all elements in enclave *j*, for j < i and strictly less than all elements in enclave *j*, for j > i.
- 5) Locally sort elements within enclaves (in parallel).

As a samplesort implementation, assuming an efficient local sort algorithm (we use Quicksort in our implementation), the total time complexity of this non-oblivious sort is  $O(N \log N)$ . Additionally, Step (2) incurs a negligible O(1)communication cost, and Step (3) incurs O(N) communication, so the total communication cost of this algorithm is O(N), similar to that of our distributed ORP. We also note that the above works assuming no duplicate elements in order to avoid leakage during comparison, after the preceding ORP (this is not explicitly discussed in [61]). We avoid this by appending a new random value randKey to each element and then sorting according to the tuple of (key, randKey).

## 5. Secure Inter-Enclave Communication

In the world of distributed algorithms, the Message Passing Interface (MPI) [81] is a standardized interface to implement distributed protocols using message-passing. Numerous MPI implementations of standard algorithms, like bitonic sort, exist and the interface is generally one that application developers are familiar with. MPI generally do not provide encryption/authentication routines, since they are optimized for extremely low latencies and high performance and assume all nodes in a cluster and the network fabric are trusted. This is not the case for our threat model, rendering standard MPI unsuitable for our use. While enclave technologies like Intel-SGX provide a mechanism to establish mutually attested TLS channels to secure communication between two enclaves [82], it is difficult to implement distributed algorithms efficiently with this stream-oriented primitive. While it is technically possible to implement an MPI interface within the attested TLS connection, the streamoriented protocol presents several issues:

 Because OS-level thread scheduling is not available from within enclaves, messages must be decrypted only when a thread enters communication subroutines. This gives decreased performance as idle CPU time cannot be used to maintain TLS connections as in "standard" user-space TLS implementations, which use multithreading to schedule TLS session management concurrently with the main process thread.

<sup>4.</sup> Step (1) requires that  $N \geq E^2$  for all MERGESPLIT operations to be run locally. If  $N < E^2$ , some MERGESPLIT operations must occur across enclaves in order to perform log E layers, but this seems unavoidable.

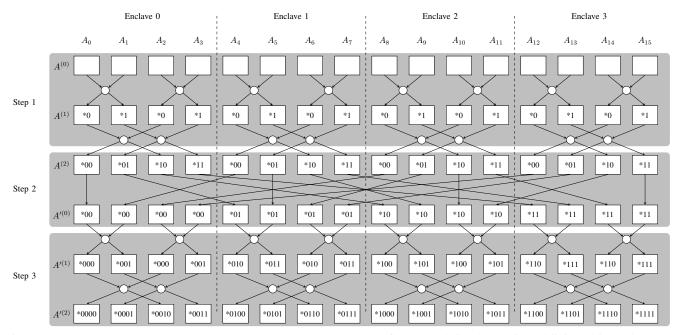


Figure 4: DBUCKET ORP's 3-stage, rearranged MERGESPLIT butterfly network implementing oblivious bin assignment. Steps 1 and 3 are performed entirely within a single enclave, and the rearrangement step in Step 2 requires only O(N) communication overhead.

- Since TLS is a stream-oriented protocol, any data that is decrypted and not ready for immediate use by the process (e.g., when messages may be sent and received in different orders) must be buffered within enclave memory. This consumes precious enclave memory, even though it is equally secure to buffer the encrypted form of these messages in host memory.
- TLS's stream-oriented nature also limits the parallelizability of encryption/decryption operations. A performant distributed application may utilize multiple threads for multiprocessing within a single enclave, but TLS connections are not designed for multiple consumers from the data stream at the same time, and TLS's encryption/decryption operations become a multithreading bottleneck within the enclave.

On the other hand, standard attested TLS is also unsuitable for our use case. It seems that a more "powerful" inter-enclave communications primitive is necessary, and it is clear that data must be encrypted and authenticated before sent out onto the network. Ideally, this *encrypted MPI layer* provides three properties:

- 1) **Security.** Remain secure against active network attackers (and, in the TEE setting, an untrusted host OS) able to read, tamper with, and replay messages.
- Ease of use. Expose an API to the programmer that is semantically identical to the underlying MPI routines, which are commonly understood by application developers.
- 3) **Performance.** Incur minimal overhead and maintain parallelizability equivalent to that of standard MPI.

While there is a preexisting line of work focused on encrypting MPI messages [83]-[87] which maintains ease of use and performance, these solutions have generally incomplete security, ranging from catastrophic use of insecure cryptographic schemes ([83]-[85]) to vulnerability to replay attacks ([86], [87]). [86] and [87] both discuss in greater detail the security vulnerabilities in many of these past works, and both works mention replay attacks as an attack that is out of scope for their respective designs. By contrast, attested TLS's stream primitive provides sufficient security but loses the ease of use and performance properties of MPI, namely in terms of parallelizability as discussed earlier. We note that the more recent Intel SGXv2 architecture provides multi-socket functionality utilizing NUMA for crossenclave communication between CPUs [88]. However, this is based on an entirely different paradigm-shared-memory vs. shared-nothing-and is currently limited to only inter-socket communication within a single machine, meaning it is also unsuitable for our distributed computing model.

Because of these solutions' shortcomings, we designed our own novel encrypted MPI layer using modern cryptographic techniques in order to fulfill all three desired properties, which we describe further in this section. At a high level, to achieve replay security we use unique, unidirectional, pairwise keys (against cross-channel replay attacks), we include the tag in the authenticated data (against cross-tag replay attacks) and implement an efficient sliding window algorithm to check counter uniqueness (against direct replay attacks within a channel).

Attested Pairwise TLS Key Derivation. The protocol begins with a pairwise, mutually attested TLS handshake

between each pair of enclaves in the network [82], [89]. Any tampering during this step is detectable by the enclaves, and the output is a shared secret between each enclave pair. In standard TLS, the bytestream comprising the data would be encrypted using keys derived from this; in our protocol, the TLS portion is completed without any subsequent input data. Instead, we just use the shared secrets from the TLS handshakes to produce a set of unidirectional encryption keys for each pair of enclaves. We denote the key used to communicate from enclave *i* to enclave *j* as  $K_{ij}$  (note that  $K_{ij} \neq K_{ji}$ , i.e., different keys are used for different directions within the same TLS handshake).

Authenticated Encryption with Replay Resistance. Once pairwise keys  $K_{ij}$  have been derived from the attested TLS handshake, any message sent may be encrypted with them. The steps to encrypt a message M are as follows:

- 1) Generate a random IV value.
- Fetch and increment a monotonically increasing counter value c which is unique for each message sent for a given enclave, to mitigate replay attacks.
- 3) Encrypt the value  $c \| M$  using AES-GCM [90] under the key  $K_{ij}$  to produce  $C_1$ , using the randomly generated IV. The AES-GCM routine should be called with the MPI tag of the message as the additional authenticated data.
- 4) Produce the ciphertext  $C = IV ||C_1$ .

AES-GCM is an authenticated mode of encryption, i.e., it also protects the integrity of the ciphertext. Additionally, the MPI tag of the message is included as additional authenticated data in the AES-GCM routine in order to prevent an attacker from changing the MPI tag of the message in-flight to tamper with the behavior of the algorithm.

Upon receipt of ciphertext C, the recipient enclave performs the steps in reverse order to decrypt it:

- 1) Extract IV and ciphertext  $C_1$  from message C.
- 2) Decrypt  $C_1$  under the key  $K_{ij}$  using AES-GCM, with the extracted IV, and key  $K_{ij}$ , and the MPI tag value as the additional authenticated data. If decryption fails, return an error.
- 3) Extract the counter value c and the message M from the resulting decryption.
- Verify the counter value c has not been previously seen by the decryption routine. A technique to efficiently implement this is given next.

**Efficient Counter Uniqueness Verification via Sliding Window.** Borrowing on ideas from the DTLS specification [62]<sup>5</sup>, we implement a sliding window algorithm to efficiently check the uniqueness of message counters for a given channel.

As is common, our sliding window assumes counter values will typically be received in an order "close to" monotonically increasing. As such, as sequences of consecutive counter values are received, we can increment a head counter to mark all values below the head as implicitly "received." Similarly, since all counter values above a certain threshold won't be received for a while, we denote a tail counter to mark all values above the tail as implicitly "not received." All values between head and tail will be explicitly tracked using an efficient bitfield data structure, with one bit allocated for each counter value between the head and the tail. The process of verifying the uniqueness of a value becomes:

- If c < head, c has already been seen.
- If c > tail, c has not yet been seen.
- If head ≤ c ≤ tail, c has been seen if and only if the bit window[c - head] has been set.

## 6. Assembly-Level Oblivious Primitives

A fundamental building block of most oblivious algorithms and data structures is oblivious swapping and assignment. In this section, we describe ways to optimize the performance of these two basic operations in modern hardware. Swap and assignment operators have traditionally been implemented using the x86-specific CMOV instruction [91]. This instruction performs a branchless conditional move from the source to the target operand depending on register flags set by an earlier comparison operation, such as a CMP or TEST, and this can be composed into an oblivious assignment operator or an oblivious swap operator.

CMOV-based swapping, however, is no longer the most performant option for swapping on modern hardware. Works such as Snoopy [36] utilize x86-64 AVX2 instructions like VPBLENDVB to implement a "CMOV-like" operation with AVX2's 256-bit registers, which achieves faster performance than CMOV on AVX2 but architecturally constrained to the x86-64 AVX2 platform and difficult to translate to other platforms or architectures, resulting in less portable code. Instead, we propose using an XOR-based swap as our oblivious swap operator. XOR-based swapping is not new in any sense but, to the best of our knowledge, ours is the first XOR-based open-source implementation for oblivious swapping, hence we need to elaborate on the details. By transforming the boolean condition variable into a mask of 1's or 0's using arithmetic operations and using it to mask the second XOR operations within the traditional XOR swap, we are able to produce a branchless version of the conditional swap with native, portable C code. Likewise, this can be used to construct an oblivious assignment operator, again masking the XOR operation to either set the destination operand to the source operand or leave it unchanged.

When using this in a loop to produce an oblivious swap of any length, modern compilers are able to automatically loop unroll and vectorize the XOR-based swap on any target platform. In practice, the XOR-based swap compiles down to 256bit VPXOR instructions on AVX2 platforms. The resulting binary, as we will show in our evaluation, is approximately equivalent in performance; however, we argue that the source code portability benefits and automatic compatibility with future vector architectures favor the use of XOR-based swap.

<sup>5.</sup> In fact, we originally implemented the communications channel using DTLS directly, but it didn't give us enough flexibility to include the MPI tag as additional authenticated data or to parallelize the encryption routines for multiple messages sent in parallel.

For oblivious swapping applications, the same XOR-based oblivious swap source code may automatically be compiled to support targets with even newer vector architectures, such as AVX-512 or RISC-V's vector extension's assembly-level variable-length vector registers [92], with no additional effort by the application developer. We omitted AVX-512 testing of our XOR-based swap due to limitations of the Open Enclave SDK we used, which does not support AVX-512 at this time. We note, however, that there is no 512-bit equivalent of the VPBLENDVB instruction, while AVX-512 has the 512-bit VPXORQ instruction as part of the base instruction set. This would likely cause the XOR-based swap to outperform the VPBLENDVB-based swap outright on platforms supporting AVX-512. A simplified version of the C code used to implement the oblivious swap and assignment operators is available in the Appendix.

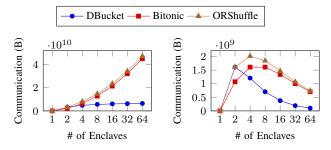
### 7. Experimental Evaluation

In this section, we compare the performance of our proposed oblivious primitives with the prior state-of-the-art results. We present a set of microbenchmarks to demonstrate the impact of each major optimization we introduced. Additionally, we propose a new, simpler version of Snoopy and conduct a comparison with the original approach.

Experimental Setup. We evaluate our DBUCKET oblivious sort using the DCsv3-series of Azure confidential computing virtual machines which use Intel-SGXv2 [93]. The Azure virtual machines utilize Intel Xeon Platinum 8370C CPUs with hyperthreading/SMT disabled, running Ubuntu 20.04. We use the Standard DC8s v3 machines with 64 GiB of memory and 32 GiB of EPC memory per machine. They are all connected to the same local network with an average RTT between any two machines of 1.25 ms (utilizing the proximity placement group feature of Azure). We ran all experiments, including competitors/baselines, on the same machines. While [31] experimented with both SGX-v1 and SGX-v2, we repeated their tests on both versions and chose to only focus on v2 due to its superior performance. Furthermore, the ORShuffle of [31] did not support distributed execution so we redesigned and re-implemented it from scratch (plus we verified our version has nearly identical performance to that of [31] for a single machine).

All algorithms in this paper are realized with  $\sim 10,000$  lines of C code. We use version 0.18.5 of the Open Enclave SDK [94], version 2.16.4 of the mbedTLS library [95], and version 3.3.2 of the MPICH library [96].

We use 128-byte elements in all aspects of this evaluation unless otherwise stated. To generate the results, our enclave program includes a data set generation routine that generates elements with a configurable size, each with a fixed, 64-bit sort key. This generation time is not included in any ORP or sorting time measurements—though the time taken to generate the random keys for DBUCKET ORP *is* included since it is part of the algorithm itself. In all figure legends, we will use the DBUCKET descriptor to refer both to DBUCKET ORP and DBUCKET sort, depending on the context.



(a) Total Communication (b) Per-Enclave Communication Figure 5: Number of bytes sent across the network for fixed size N and increasing number of enclaves E. Total summed across all enclaves and per-enclave values are shown.

**Baselines/Competitors.** Our baseline for distributed sorting will be our distributed bitonic sort with the chunking optimization described in §3, where the chunking factor is C = 4096 (unless otherwise specified). In our evaluation, our bitonic sort marginally outperforms Snoopy's bitonic sort implementation by up to 8% for multiple threads.

**Measuring Network Communication.** We demonstrate the improved network communication overhead of DBUCKET sort by measuring the total number of bytes sent over the network across all enclaves. Figure 5a measures the total number of bytes sent over the network for a fixed data set size and increasing number of enclaves for DBUCKET, distributed bitonic, and distributed ORShuffle-based sort. We observe that the total network communication increases approximately proportional to  $O(N \log^2 E)$  for bitonic sort and the ORShuffle-based sort, DBUCKET sort's communication remains approximately constant as the number of enclaves increases, due to its O(N) communication overhead.

Figure 5b shows the per-enclave communications cost of executing each sort. The per-enclave communication is approximately  $\frac{N \log E}{E}$  for bitonic and ORShuffle-based sort and  $\frac{N}{E}$  for DBUCKET sort. This means that, for bitonic and ORShuffle-based sort, the per-enclave communication reaches a maximum at 4–8 enclaves before decreasing with greater numbers of enclaves, thus incurring a performance penalty for these algorithms with fewer than 16 enclaves.

**Distributed Oblivious Random Permutation.** In Figure 6, we compare our DBUCKET ORP against ORShuffle [31] in a distributed environment. Figure 6a is conducted with a data set size of 2 GiB and shows that DBUCKET ORP provides a marginal  $1.04 \times$  speedup over ORShuffle in the single-enclave setting. This gap, however, widens in a multi-enclave environment, providing a  $3.04 \times$  speedup over ORShuffle with 8 enclaves and  $9.52 \times$  speedup with 64 enclaves. DBUCKET ORP scales well to large numbers of enclaves, with a  $21.1 \times$  speedup when the amount of hardware is increased from 1 enclave to 64 enclaves, compared to only a  $2.32 \times$  speedup for ORShuffle with the same increase in hardware. Figure 6b demonstrates that the result holds with a much larger, 128-GiB data set size—too large to fit in a single enclave. With 64 enclaves, DBUCKET ORP

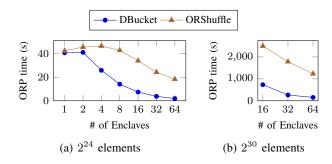


Figure 6: ORP time as the number of enclaves increases, with 1 thread per enclave and 128-B elements.

outperforms ORShuffle by  $7.96 \times$ , bringing an operation that would take 20 minutes with ORShuffle down to just 2 and a half minutes. The performance of ORShuffle decreases for 2, 4, and 8 enclaves due to the high per-enclave communications cost introduced for these less-distributed settings.

It is worth noting that our results for DBUCKET ORP in the single-enclave setting differ greatly from the results presented in [31], whose results indicate that ORShuffle outperforms bucket ORP (which the paper refers to as "BORPCompact"). We investigated this discrepancy and found that this is due to two primary reasons: (1) the paper's experimental evaluation used a buggy implementation of bucket ORP, which took  $O(N \log^2 N)$  in its implementation compared to an ideal  $O(N \log N)$  implementation of bucket ORP, and (2) our assembly-level implementation improvements provide a substantial speedup for DBUCKET ORP implementation. These two factors combined mean that even in a non-distributed setting with 1 enclave, bucket ORP (to which DBUCKET ORP reduces to in a single-enclave setting) outperforms Sasy et al.'s ORShuffle when implementing ORP.

Figures 7a and 7b show that the performance speedup gained by increasing the number of threads allocated to each enclave is preserved even in the distributed environment. Even in a 32-enclave environment, increasing the number of threads per enclave from 1 thread to 8 thread provides a  $6.8 \times$  performance speedup despite the need to communicate data across enclaves. Due to the expensive encryption/decryption operations needed to communicate data securely between enclaves across the untrusted network fabric, near-ideal scaling is achievable using DBUCKET ORP when increasing the number of threads per enclave. Because of maximum vCPU core quotas set in our Azure testing environment, we were unable to repeat this experiment with 64 enclaves, each with 8 threads, as it would exceed our allocation.

**Distributed Oblivious Sorting.** Figures 8 and 9 compare similar metrics for an oblivious sort implemented using ORP, resulting in DBUCKET sort and ORShuffle sort, as well as the bitonic sort. While in Figure 8a, the bitonic sort outperforms both DBUCKET and ORShuffle sort, the performance of DBUCKET sort quickly outperforms bitonic sort with just 4 enclaves, all the way up to a  $6.63 \times$  speedup over bitonic sort with 64 enclaves. This speedup factor is similarly maintained in the larger data set in Figure 8b, with a  $5.38 \times$  speedup

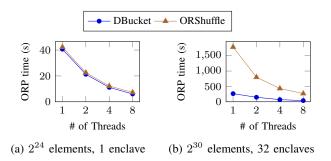


Figure 7: ORP time as the number of threads increases, for single- and multi-enclave environments, with 128-B elements.

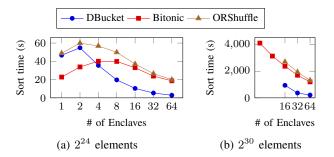


Figure 8: Sort time as the number of enclaves increases, with 1 thread per enclave and 128-B elements.

factor over bitonic sort when sorting a 128-GiB dataset with 64 enclaves.

Similarly, Figure 9 shows that, again, DBUCKET sort maintains a near-ideal speedup factor when increasing the number of threads per enclave even in highly distributed setting. With 32 enclaves, increasing from 1 thread per enclave to 8 threads per enclave increases performance by  $6.32\times$ . Overall, DBUCKET sort provides an improvement in sorting our 128-GiB data set from 221 seconds to just 58 seconds over the bitonic sort.

We observe similar scaling characteristics for the bitonic sort as we do for the ORShuffle operation in the previous section: The bitonic sort is less suited to distributed applications due the high cost of introducing swaps across the network. In this case, the cost is even more extreme, and a speedup over a single-enclave bitonic sort is not observed until 64 enclaves are added, compared to just 16 for the ORShuffle-basd sort and 4 for DBUCKET sort.

Finally, in Figure 10, we show that the performance characteristics are preserved across a variety of block sizes in our testing. Figure 10a shows a single-enclave setting, where the maximum element size we were able to test for  $N = 2^{20}$  elements was an element size of 1 KiB, resulting in a 16 GiB dataset. While the bitonic sort is still by far the most performant in the single-enclave setting, bucket sort is able to provide similar performance to ORShuffle sort across all tested environments. In fact, for larger element sizes, bucket sort outperforms ORShuffle sort due to the improved efficiency of the XOR-based oblivious swap operator over

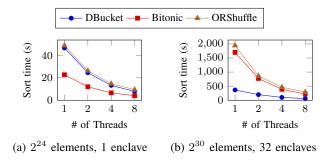


Figure 9: Sort time as the number of threads increases, for single- and multi-enclave environments, with 128-B elements.

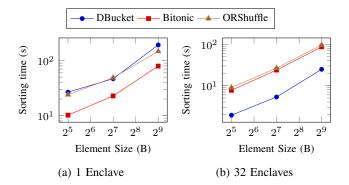


Figure 10: Sort time as the element size increases, for a fixed number of elements  $N = 2^{24}$ , in both single-enclave and multi-enclave environments.

the CMOV-based swap. In the distributed experiment shown in Figure 10b, the performance advantage of DBUCKET sort is maintained across all block sizes.

**Communication Chunk Sizes.** We additionally investigated the performance impact of varying the size of chunking that was performed for inter-enclave communication for both the bitonic sort and DBUCKET sort. As discussed in §3, we expect an increase in the chunk size to decrease the effect of network latency and improve overall performance, which is supported by the data in Figure 11. While the sorting time strictly decreases as the chunk size is increased, we observe diminishing returns as the chunk size is increased past approximately  $2^9$  elements, resulting in our choice of chunk size of  $2^{12}$  for this evaluation.

We similarly explored the impact of chunking the communication of elements and buckets across the network in DBUCKET sort. In the case of DBUCKET sort, the minimal chunk size must be at least the size of one bucket, so we evaluated the effect of increasing the number of buckets in a chunk. We observe that the impact on overall performance is minimal mainly because of the improved communication overhead of DBUCKET sort.

**Microbenchmarks.** We performed a series of ORP benchmarks to isolate the performance impact of each of our major optimizations: using XOR-based oblivious swap, implementing the MERGESPLIT operation with ORCompact instead

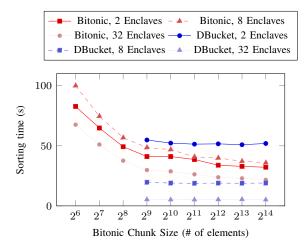


Figure 11: Sorting time of bitonic and DBUCKET sort, as the size of the chunk of elements sent over the network at once is increased. Bucket sort may not go below a chunk size of Z = 512.

bitonic sort, and rearranging bucket sort's butterfly network to achieve O(N) communication overhead. Figure 12 shows 4 sets of experiments to sort the same 2-GiB dataset with different numbers of enclaves, with each combination of the 3 aforementioned optimizations enabled.

The ORCompact optimization has the greatest relative impact on performance in less-distributed environments, and its relative speedup decreases as the number of enclaves increases, from a  $3.28 \times$  speedup with 1 enclave to a more modest  $1.25 \times$  speedup with 32 enclaves. This makes sense, since the the MERGESPLIT operations are performed entirely locally, and the overall ORP time is dominated by local operations for less-distributed environments.

On the other hand, rearranging the bucket routing network to achieve O(N) communication overhead—logically—has by far the greatest performance impact on highly distributed settings, with a lesser impact on less-distributed settings. In the 32-enclave setting, this optimization provided a  $2.87 \times$  speedup over the naïve implementation on bucket sort achieved using that optimization alone, while in the 2-enclave setting, it provided a smaller  $1.24 \times$  speedup. Obviously, there is no effect on the 1-enclave setting for this optimization.

The XOR-based swapping has a less significant impact on the ORP performance overall, exhibiting behavior similar to ORCompact since its operation largely improves the performance of local operations. It provides a  $1.71 \times$  speedup in the 1-enclave setting while providing a mere  $1.19 \times$ speedup in the 32-enclave setting.

**Snoopy++.** Below, we provide a simplified version of Snoopy [36] which is based on our scalable oblivious sort operator. In particular, in our Snoopy++:

- 1) We construct an array from all keys associated with their values and all requests containing desired keys.
- 2) We sort the array according to (key, b), where b is a bit that is 0 for data items and 1 for requests. This

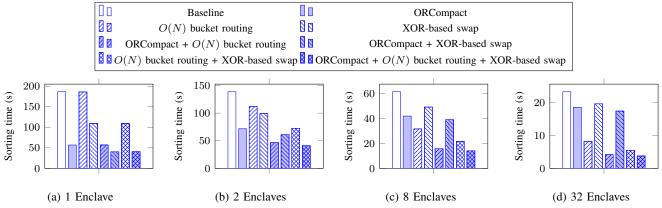


Figure 12: Microbenchmarks of DBucket sort isolating different performance speedups achieved using different optimizations discussed in §4 and §6.

results in an array where all requests are positioned immediately after the data item containing their desired values.

- Linearly scan across the entire array, obliviously copying data values into the request immediately following the data item if the keys match.
- Compact the array such that all requests are positioned at the end of the array once again.
- 5) We return the values stored in the requests at the end of the array.

In keeping with the idea of targeting request latency as a constraint rather than as a metric, we first establish the minimum latency for any request to be served by Snoopy++, which is the time it takes to perform the sort, linear scan, and compaction steps of the oblivious join.

Focusing on a 32-enclave setting, Snoopy++ requires just 0.86 seconds. Assume that N is the number of elements and R is the number of requests being processed during the join operation. Note that, assuming  $N \ll R$ , the 0.86-second join time is *independent* of R, meaning that Snoopy++ is able to achieve extremely high throughputs while maintaining a latency of no more than 0.86 seconds. Precisely, assuming a sort size of  $2^{24}$  (as DBUCKET must be padded to a power of 2 anyway), Snoopy++ can handle  $2^{24} - N$  requests per 0.86-second epoch. If N = 16 million, this means Snoopy++ could handle over 700,000 requests per epoch! Comparing this against Snoopy's maximum throughput with the same hardware (32 sub-ORAMs and 1 load balancer), Snoopy is able to reach only 4,093 requests per second for the same database size. We provide Snoopy's numbers for this experiment in the appendix.

### 8. Related Work

Here we provide additional relevant existing works, focusing on oblivious compaction, oblivious shuffling, and other approaches for oblivious execution of algorithms.

**Oblivious Compaction.** Goodrich [97] proposes an orderpreserving tight compaction algorithm that can output all n elements, including the marked items, using as few as  $(\log_2 n - 2)n$  swaps. This algorithm can also be parallelized. Asharov et al. [29] propose a linear-time tight compaction algorithm, but Dittmer and Ostrovsky [98] show that this algorithm has a considerably big constant. Mitchell and Zimmerman [99] propose an algorithm for compaction, but Falk and Ostrovsky [100]'s analysis indicates that this algorithm requires more swaps than the solution of Sasy et al. [31] unless n is considerably large.

**Oblivious Shuffling.** Melbourne Shuffle [101] and Stash Shuffle [56] are two shuffling algorithms designed for different settings. Melbourne Shuffle is designed for cloud storage with minimal client storage overheads. Stash Shuffle, on the other hand, is designed to be efficient for the TEE setting and uses a stash of size  $O(\sqrt{n})$  to hold items that cannot be distributed in a given round to their intended output buckets. There are also probabilistic sorting networks [102] which can achieve a smaller size. While random sorting networks appear to be asymptotically efficient [103], their practical performance has not been demonstrated [104].

**MPC and obliviousness.** In multi-party computation (MPC), one or more parties secret-share their data across multiple servers, assumed to be *non-colluding*, and the latter communicate to evaluate subsequent queries [105]–[113]. The vast majority of these works focus on challenges arising from the communication and interactive nature of MPC [114]–[119], proposing trade-offs (e.g., optimizing the circuit size). However, the oblivious nature of these approaches could inspire the designing of oblivious algorithms for TEEs.

**Concurrent Works.** [120], [121] are independent and concurrent works on oblivious sort and shuffle within TEEs, which appeared online after our submission. However, they do not focus on the distributed/parallel settings that we target, hence they do not achieve comparable scalability to ours.

#### 9. Conclusion

Oblivious primitives play an important role in the performance of systems with strong privacy. Oblivious sorting and shuffling are two fundamental and widely used primitives in this setting. As distributed computing becomes increasingly important for scaling computations, such oblivious primitives must also be lifted in the distributed setting, without compromising security. In this work, we proposed the first distributed and scalable solutions for oblivious sorting and shuffling, dubbed DBUCKET sort and DBUCKET ORP. As we show, they exhibit near-ideal scaling characteristics in both multi-threading and multi-enclave cases, e.g., with 64 enclaves they achieve a  $9.5 \times$  speedup over the previous state of the art for shuffling and outperform bitonic sort by  $6.3 \times$ .

#### Acknowledgement

This work was supported in part by the Hong Kong Research Grants Council under grant GRF-16200721.

#### References

- M. A. Abdelraheem, T. Andersson, and C. Gehrmann, "Inference and record-injection attacks on searchable encrypted relational databases," *IACR Cryptol. ePrint Arch.*, p. 24, 2017.
- [2] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd* ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 668–679.
- [3] M. Giraud, A. Anzala-Yamajako, O. Bernard, and P. Lafourcade, "Practical passive leakage-abuse attacks against symmetric searchable encryption," in *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) -Volume 4: SECRYPT*, 2017, pp. 200–211.
- [4] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov, "Breaking web applications built on top of encrypted data," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1353–1364.
- [5] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in 19th Annual Network and Distributed System Security Symposium, NDSS, 2012.
- [6] C. Liu, L. Zhu, M. Wang, and Y. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," *Inf. Sci.*, vol. 265, pp. 176–188, 2014.
- [7] D. Poulioi and C. V. Wright, "The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer* and Communications Security, 2016, pp. 1341–1352.
- [8] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in 25th USENIX Security Symposium, 2016, pp. 707–720.
- [9] J. L. Dautrich Jr and C. V. Ravishankar, "Compromising privacy in precise query protocols," in *Proceedings of the 16th International Conference on Extending Database Technology*, 2013, pp. 155–166.
- [10] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson, "Learning to reconstruct: Statistical learning theory and encrypted database attacks," in *IEEE Symposium on Security and Privacy*, 2019, pp. 1067–1083.
- [11] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 315–331.
- [12] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted databases: New volume attacks against range queries," in *Proceedings ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 361–378.
- [13] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1329–1340.

- [14] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Data recovery on encrypted databases with k-nearest neighbor query leakage," in *IEEE Symposium on Security and Privacy*, 2019, pp. 1033–1050.
- [15] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution," in *IEEE Symposium on Security and Privacy*, 2020, pp. 1223–1240.
- [16] M. Lacharité, B. Minaud, and K. G. Paterson, "Improved reconstruction attacks on encrypted data using range query leakage," in 2018 IEEE Symposium on Security and Privacy, SP 2018, 2018, pp. 297–314.
- [17] E. A. Markatou and R. Tamassia, "Full database reconstruction with access and search pattern leakage," in *Information Security* -22nd International Conference, ISC, ser. Lecture Notes in Computer Science, vol. 11723, 2019, pp. 25–43.
- [18] E. M. Kornaropoulos, N. Moyer, C. Papamanthou, and A. Psomas, "Leakage inversion: Towards quantifying privacy in searchable encryption," in *Proceedings of the 2022 ACM SIGSAC Conference* on Computer and Communications Security, 2022, pp. 1829–1842.
- [19] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Responsehiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks," in 2021 IEEE Symposium on Security and Privacy (SP), IEEE, 2021, pp. 1502–1519.
- [20] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE Symposium on Security and Privacy*, 2015, pp. 640–656.
- [21] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma, "Observing and preventing leakage in mapreduce," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1570–1581.
- [22] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018, pp. 973–990.
- [23] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.
- [24] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," J. ACM, vol. 43, no. 3, pp. 431–473, May 1996.
- [25] K. Chung, Z. Liu, and R. Pass, "Statistically-secure ORAM with õ(log<sup>2</sup>n) overhead," in *Advances in Cryptology - ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 8874, 2014, pp. 62–81.
- [26] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly secure oblivious RAM without random oracles," in *Theory of Cryptography* - 8th Theory of Cryptography Conference, ser. Lecture Notes in Computer Science, vol. 6597, 2011, pp. 144–163.
- [27] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," in *Privacy Enhancing Technologies - 13th International Symposium, PETS*, ser. Lecture Notes in Computer Science, vol. 7981, 2013, pp. 1–18.
- [28] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "PanORAMa: Oblivious RAM with logarithmic overhead," in *FOCS*, 2018.
- [29] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "Optorama: Optimal Oblivious RAM," in Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2020, pp. 403–432.
- [30] M. T. Goodrich, "Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data," in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, 2011, pp. 379–388.
- [31] S. Sasy, A. Johnson, and I. Goldberg, "Fast fully oblivious compaction and shuffling," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, 2022, pp. 2565–2579.
- [32] P. Zhang, C. Song, H. Yin, D. Zou, E. Shi, and H. Jin, "Klotski: Efficient obfuscated execution against controlled-channel attacks," in ASPLOS, 2020.

- [33] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "OBFUSCURO: A commodity obfuscation engine on Intel SGX," in *Network and Distributed System Security Symposium*, 2019.
- [34] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Mar. 2017, pp. 283–298.
- [35] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," arXiv preprint arXiv:2003.09481, 2020.
- [36] E. Dauterman, V. Fang, I. Demertzis, N. Crooks, and R. A. Popa, "Snoopy: Surpassing the scalability bottleneck of oblivious storage," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, 2021, pp. 655–671.
- [37] Signal, https://github.com/signalapp/, 2014.
- [38] Google, "Key transparency," 2011.
- [39] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: bringing key transparency to end users," in USENIX Security Symposium, 2015, pp. 383–398.
- [40] R. W. Lai and S. S. Chow, "Forward-secure searchable encryption on labeled bipartite graphs," in ACNS International Conference on Applied Cryptography and Network Security, Springer, 2017, pp. 478–497.
- [41] M. Chase and S. Kamara, "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, ser. Lecture Notes in Computer Science, vol. 6477, 2010, pp. 577–594.
- [42] I. Demertzis and C. Papamanthou, "Fast searchable encryption with tunable locality," in SIGMOD, 2017.
- [43] I. Demertzis, R. Talapatra, and C. Papamanthou, "Efficient searchable encryption through compression," *PVLDB*, 2018.
- [44] I. Demertzis, J. Ghareh Chamani, D. Papadopoulos, and C. Papamanthou, "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, 2020.
- [45] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," *CRYPTO*, 2018.
- [46] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical Private Range Search Revisited," in *SIGMOD*, 2016.
- [47] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou, "Practical private range search in depth," *TODS*, 2018.
- [48] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: attack mitigation for encrypted databases via adjustable leakage," in 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020, 2020, pp. 2433–2450.
- [49] S. Kamara and T. Moataz, "Sql on structurally-encrypted databases.," ASIACRYPT, 2019.
- [50] J. G. Chamani, C. Papamanthou, D. Papadopoulos, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in CCS, 2018.
- [51] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with SGX," in *Proceedings of the* 12th European Workshop on Systems Security, 2019, pp. 1–6.
- [52] J. G. Chamani, I. Demertzis, D. Papadopoulos, C. Papamanthou, and R. Jalili, "Graphos: Towards oblivious graph processing," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, pp. 4324–4338, 2023.
- [53] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage.," in NDSS, vol. 14, 2014, pp. 23–26.
- [54] J. Ghareh Chamani, D. Papadopoulos, M. Karbasforushan, and I. Demertzis, "Dynamic searchable encryption with optimal search in the presence of deletions," in *31st USENIX Security Symposium* (USENIX Security 22), 2022, pp. 2425–2442.

- [55] S. Sasy and O. Ohrimenko, "Oblivious sampling algorithms for private data analysis," *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [56] A. Bittau, Ú. Erlingsson, P. Maniatis, et al., "Prochlo: Strong privacy for analytics in the crowd," in *Proceedings of the 26th symposium* on operating systems principles, 2017, pp. 441–459.
- [57] M. Armbrust, R. S. Xin, C. Lian, et al., "Spark SQL: Relational data processing in Spark," in Proceedings of the 2015 ACM SIGMOD international conference on management of data, ACM, 2015, pp. 1383–1394.
- [58] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [59] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig, "Benchmarking the second generation of intel sgx hardware," in *Data Management on New Hardware*, 2022, pp. 1–8.
- [60] K. E. Batcher, "Sorting networks and their applications," in Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, ser. AFIPS '68 (Spring), 1968, pp. 307–314.
- [61] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket oblivious sort: An extremely simple oblivious sort," in *Symposium on Simplicity in Algorithms (SOSA)*, pp. 8–14. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611976014.2.
- [62] E. Rescorla and N. Modadugu, *Datagram Transport Layer Security Version 1.2*, RFC 6347, Jan. 2012.
- [63] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference* on Computer Systems, 2020, pp. 1–16.
- [64] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13, 2013.
- [65] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," White paper, 2016.
- [66] ARM Limited, ARM TrustZone Technology, https://developer.arm. com/documentation/102412/latest, 2004.
- [67] M. Ajtai, J. Komlós, and E. Szemerédi, "An O(nlogn) sorting network," in *Proceedings of the Fifteenth Annual ACM Symposium* on Theory of Computing, ser. STOC '83, 1983, pp. 1–9.
- [68] J. Van Bulck, M. Minkin, O. Weisse, et al., "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 991–1008.
- [69] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *11th USENIX Workshop on Offensive Technologies* (WOOT 17), 2017.
- [70] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *Proceedings of the 10th European Workshop on Systems Security*, ACM, 2017, p. 2.
- [71] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017, pp. 299–312.
- [72] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in CHES, 2017.
- [73] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware*, and Vulnerability Assessment, Springer, 2017, pp. 3–24.
- [74] W. Wang, G. Chen, X. Pan, *et al.*, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *CCS*, 2017.
- [75] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjà vu," in *Proceedings of Asia Conference on Computer and Communications Security*, 2017, pp. 7–18.
- [76] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in USENIX Security Symposium, 2016, pp. 857–874.

- [77] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in USENIX, 2017.
  [78] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating
- [78] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.," in NDSS, 2017.
- [79] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *AsiaCCS*, 2016.
- [80] C. A. R. Hoare, "Algorithm 65: Find," Commun. ACM, vol. 4, no. 7, pp. 321–322, Jul. 1961.
- [81] M. P. I. Forum, "Mpi: A message-passing interface standard, version 4.0," Message Passing Interface Forum, Tech. Rep., Jun. 2021.
- [82] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij, "Integrating remote attestation with transport layer security," Intel Labs, Tech. Rep., 2018.
- [83] X. Ruan, Q. Yang, M. I. Alghamdi, S. Yin, and X. Qin, "Esmpich2: A message passing interface with enhanced security," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 3, pp. 361–374, 2012.
- [84] M. A. Maffina and R. S. RamPriya, "An improved and efficient message passing interface for secure communication on distributed clusters," in 2013 International Conference on Recent Trends in Information Technology (ICRTIT), 2013, pp. 329–334.
- [85] V. Rekhate, A. Tale, N. Sambhus, and A. Joshi, "Secure and efficient message passing in distributed systems using one-time pad," in 2016 International Conference on Computing, Analytics and Security Trends (CAST), 2016, pp. 393–397.
- [86] A. Naser, M. Gavahi, C. Wu, V. T. Hoang, Z. Wang, and X. Yuan, "An empirical study of cryptographic libraries for MPI communications," in 2019 IEEE International Conference on Cluster Computing (CLUSTER), 2019, pp. 1–11.
- [87] A. Naser, C. Wu, M. S. Lahijani, et al., Cryptmpi: A fast encrypted MPI library, 2020. arXiv: 2010.06471 [cs.DC].
- [88] S. Johnson, R. Makaram, A. Santoni, and V. Scarlata, "Supporting Intel® SGX on multi-socket platforms," Intel Corporation, Tech. Rep., 2021.
- [89] E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.3, RFC 8446, Aug. 2018.
- [90] J. A. Salowey, D. McGrew, and A. Choudhury, AES Galois Counter Mode (GCM) Cipher Suites for TLS, RFC 5288, Aug. 2008.
- [91] "Intel® 64 and IA-32 architectures software developer manuals," Apr. 6, 2023.
- [92] "The RISC-V instruction set manual, volume I: User-level ISA, document version 2019121," Dec. 2019.
- [93] Intel, 3rd Gen Intel® Xeon® scalable processors brief, 2021.
- [94] Open Enclave SDK, https://openenclave.io/sdk/, 2023-04-29.
- [95] Mbed TLS, https://www.trustedfirmware.org/projects/mbed-tls/, 2023-04-29.
- [96] MPICH, https://www.mpich.org/, 2023-04-29.
- [97] M. T. Goodrich, "Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data," in *Proceedings of the 23rd Annual ACM Symposium on Parallelism* in Algorithms and Architectures, 2011, pp. 379–388.
- [98] S. Dittmer and R. Ostrovsky, "Oblivious tight compaction in O(n) time with smaller constant," in *International Conference on Security* and Cryptography for Networks, Springer, 2020, pp. 253–274.
- [99] J. C. Mitchell and J. Zimmerman, "Data-oblivious data structures," in 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.
- [100] B. H. Falk and R. Ostrovsky, "Secure merge with O(nloglogn) secure operations," in 2nd Conference on Information-Theoretic Cryptography (ITC 2021), 2021.
- [101] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal, "The melbourne shuffle: Improving oblivious storage in the cloud," in Automata, Languages, and Programming: 41st International Colloquium, ICALP, Springer, 2014, pp. 556–567.
- [102] T. Leighton, Y. Ma, and T. Suel, "On probabilistic networks for selection, merging, and sorting," in *Proceedings of the seventh* annual ACM symposium on Parallel algorithms and architectures, 1995, pp. 106–118.

- [103] M.-W. Shih, "Securing Intel SGX against side-channel attacks via load-time synthesis.," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2020.
- [104] A. Czumaj, "Random permutations using switching networks," in Proceedings of the forty-seventh annual ACM symposium on Theory of Computing, 2015, pp. 703–712.
- [105] A. Aly and M. V. Vyve, "Securely solving classical network flow problems," in *Information Security and Cryptology ICISC*, ser. Lecture Notes in Computer Science, vol. 8949, 2014, pp. 205–221.
- [106] M. Blanton and S. Saraph, "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.
- [107] M. Blanton, A. Steele, and M. Alisagari, "Data-oblivious graph algorithms for secure computation and outsourcing," in *Proceedings* of the 8th ACM SIGSAC symposium on Information, computer and communications security, 2013, pp. 207–218.
- [108] J. Doerner, D. Evans, and A. Shelat, "Secure Stable Matching at Scale," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1602–1613.
- [109] J. Doerner and A. Shelat, "Scaling ORAM for secure computation," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 523–535.
- [110] S. Faber, S. Jarecki, S. Kentros, and B. Wei, "Three-party ORAM for secure computation," in *International Conference on the Theory* and Application of Cryptology and Information Security, Springer, 2015, pp. 360–385.
- [111] M. T. Goodrich and J. A. Simons, "Data-Oblivious Graph Algorithms in Outsourced External Memory," in *Combinatorial Optimization and Applications*, ser. Lecture Notes in Computer Science, vol. 8881, 2014, pp. 241–257.
- [112] S. D. Gordon, J. Katz, V. Kolesnikov, et al., "Secure two-party computation in sublinear (amortized) time," in ACM Conference on Computer and Communications Security, 2012, pp. 513–524.
- [113] M. Keller and P. Scholl, "Efficient, oblivious data structures for MPC," in ASIACRYPT International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2014, pp. 506–525.
- [114] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure graph analysis at scale," in *Proceedings of the* 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 610–629.
- [115] M. Anagreh, P. Laud, and E. Vainikko, "Parallel privacy-preserving shortest path algorithms," *Cryptography*, vol. 5, no. 4, p. 27, 2021.
- [116] M. Anagreh, P. Laud, and E. Vainikko, "Privacy-preserving parallel computation of minimum spanning forest," *SN Computer Science*, vol. 3, no. 6, p. 448, 2022.
- [117] N. Al-Juaid, A. Lisitsa, and S. Schewe, "SMPG: Secure multi party computation on graph databases.," in *ICISSP*, 2022, pp. 463–471.
- [118] F. Han, L. Zhang, H. Feng, W. Liu, and X. Li, "Scape: Scalable collaborative analytics system on private database with malicious security," in 2022 IEEE 38th International Conference on Data Engineering (ICDE), IEEE, 2022, pp. 1740–1753.
- [119] C. Wang, J. Bater, K. Nayak, and A. Machanavajjhala, "IncShrink: Architecting efficient outsourced databases using incremental MPC and differential privacy," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 818–832.
- [120] S. Sasy, A. Johnson, and I. Goldberg, "Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks," in CCS, 2023.
- [121] T. Gu, Y. Wang, B. Chen, A. Tinoco, E. Shi, and K. Yi, "Efficient oblivious sorting and shuffling for hardware enclaves," *Cryptology ePrint Archive*, 2023.

## Appendix A.

# Assembly-Level Oblivious Primitive Code Listings

Here we provide code for implementing different oblivious algorithms in different settings. Figure 13, Figure 15, and Figure 16 show how to implement an oblivious comparison, oblivious swap, and oblivious assignment without relying on any oblivious instruction. On the other hand, Figure 14 provides an example of the traditional, CMOV-based oblivious swap that is still often used today in oblivious applications.

Figure 13: C code to realize an oblivious, constant-time comparison on architecture without an oblivious, constant-time comparison instruction.

cmpl %rax, \$0; mov (% rsi), %r12; mov (% rsi), %r14; mov (% rdi), %r13; cmovl %r13, %r12; cmovl %r14, %r13; mov %r12, (% rsi); mov %r13, (% rdi);

Figure 14: CMOV-based x86 assembly implementing an oblivious swap. RAX holds the swap flag, RSI and RDI hold the swap buffers, and R12–R14 are temporary registers.

Figure 15: C code to realize an XOR-based oblivious swap. If cond == 1, mask ==  $0 \times ff$ ; else, mask ==  $0 \times 00$ . Thus, if cond == 1 the operands \*a and \*b are swapped. Else, \*a is XORed with \*b twice, and operands are unchanged.

# Appendix B. Snoopy and Snoopy++ Evaluation Figures

Figures 17 and 18 present a visualization of the data we collected in our evaluation of Snoopy and Snoopy++. Figure 17 shows the performance of the oblivious join tracks closely with that of DBUCKET sort itself. Figure 18 shows that Snoopy's request throughput increases as the number of enclaves is increased, though it does not approach the request throughput achievable in the right conditions for Snoopy++.

Figure 16: C code to realize an XOR-based oblivious assignment. If cond == 1, mask ==  $0 \times 100$ . Thus, if cond == 1 the operands \*dest is set to the value of src. Else, \*dest is XORed with src twice, and the operands are unchanged.

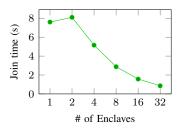


Figure 17: Snoopy++ times for an increasing number of enclaves on a database with 16M elements of size 128 B, with each enclave utilizing 8 threads.

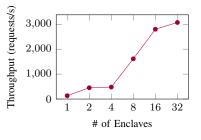


Figure 18: Snoopy request throughput as the number of enclaves acting as subORAMs increases, for a database with 16M elements of size 128 B.

# Appendix C. Distributed Bucket ORP Construction

Finally, we provide the pseudocode of DBUCKET ORP. As we mentioned in Section 4, our DBUCKET sort has a two-step structure: a distributed bucket ORP (DBUCKET ORP) followed by an efficient, non-oblivious, comparisonbased sort. In Figure 19, we present the detailed steps of DBUCKET ORP.

1:  $e_i$  represents the enclave of rank *i*, where  $i \in [1, E]$  and *E* is a power of 2. 2: self is the rank of the current enclave. 3: Z is the number of elements in a bucket. 4: A represents the globally indexed array of elements across all enclaves. 5: A[i] represents the *i*'th element of A, where  $i \in [1, N]$  and N is a power of 2. 6:  $A_j^{(i)}$  represents the *i*'th bucket in layer *j* of the bucket routing network. 7: do in parallel 8: for all  $i \in \text{local portion of } A$  in  $e_{\text{self}}$  do 9: Assign a large random value to A[i].orp\_id 10: end for 11: end parallel 12: for all  $i \in \{0, ..., \log E - 1\}$  do  $\triangleright$  Step 1: Butterfly network for log *E* levels. for all  $j \in \{0, ..., \frac{N}{2Z} - 1\}$  do MERGESPLIT $(A_{j'+j}^{(i)}, A_{j'+j+2^i}^{(i)}, \text{mut}A_{j'+j}^{(i+1)}, \text{mut}A_{j'+j+2^i}^{(i+1)})$ , where  $j' = \lfloor \frac{j}{2^i} \rfloor \cdot 2^i$ end for 13: do in parallel 14: 15: 16: 17: end parallel 18: end for 19: do in parallel ▷ Step 2: Rearrange buckets. Send bucket  $A_i^{(\log E-1)}$  to enclave  $j \mod E$ . 20: 21: end parallel 22: Let  $A'^{(0)}$  refer to the state of buckets in  $A^{(\log E-1)}$  after rearrangement. for all  $i \in \{0, ..., \log(N/E) - 1\}$  do do in parallel 23:  $\triangleright$  Step 3: Butterfly network for  $\log(N/E)$  levels. 24: for all  $j \in \{0, ..., \frac{N}{2} - 1\}$  do MERGESPLIT $(A'^{(i)}_{j'+j}, A'^{(i)}_{j'+j+2^i}, i + \log E, \operatorname{mut} A'^{(i+1)}_{j'+j}, \operatorname{mut} A'^{(i+1)}_{j'+j+2^i})$ , where  $j' = \lfloor \frac{j}{2^i} \rfloor \cdot 2^i$ 25: 26: end for 27: end parallel 28: 29: end for 30: Non-obliviously sort A. function MERGESPLIT $(A_0, A_1, i, \mathsf{mut} A'_0, \mathsf{mut} A'_1)$ 31: 32: Let  $A_{01}$  be a view of length 2Z of the the elements in  $A_0$  followed by the elements of  $A_1$ 33:  $c \leftarrow$  the number of real elements in  $A_{01}$  with 0 in bit i of its orp\_id 34: If c > Z, abort. Obliviously assign Z - c dummy elements in  $A_{01}$  to  $A'_0$  and the remaining dummy elements to  $A'_1$ . Obliviously compact elements in  $A_{01}$  such that elements assigned to  $A'_0$  come before those assigned to  $A'_1$ . 35: 36:  $A'_0$  receives all elements in the first half of  $A_{01}$ .  $A'_1$  receives all elements in the second half of  $A_{01}$ . 37: 38:

39: end function

#### Figure 19: Pseudocode describing the implementation of DBucket ORP