OBLIVIATOR: OBLIVIous Parallel Joins and other OperATORs in Shared Memory Environments

Apostolos Mavrogiannakis* *UCSC*
Xian Wang* *HKUST*
Ioannis Demertzis *UCSC*
Dimitrios Papadopoulos *HKUST*
Minos Garofalakis *ATHENA Research Center & Technical University of Crete*

Abstract

We introduce *oblivious* parallel operators designed for both non-foreign key and foreign key equi-joins. Obliviousness ensures nothing is revealed about the data besides input/output sizes, even against a strong adversary that can observe memory access patterns. Our solution achieves this by combining trusted hardware with efficient oblivious primitives for compaction and sorting, and two oblivious algorithms: (i) an oblivious aggregation tree, which can be described as a variation of the parallel prefix sum, customized for trusted hardware, and (ii) a novel algorithm for obliviously expanding the elements of a relation. In the sequential setting, our oblivious join performs 4.6×- 5.14× faster than the prior state-of-the-art solution (Krastnikov et al., VLDB 2020) on data sets of size $n = 2^{24}$. In the parallel setting, our algorithm achieves a speedup of up to roughly 16× over the sequential version, when running with 32 threads (becoming up to 80× compared to the sequential algorithm of Krastnikov et al.). Finally, our oblivious operators can be used independently to support other oblivious relational database queries, such as oblivious selection and oblivious group-by.

1 Introduction

Cloud providers, like Amazon Redshift [6], Azure SQL [14], and Google Cloud SQL [34], offer cloud-based databases. To keep important data safe, the first line of defense is *encryption*. Several "encrypted" databases, such as CryptDB [98], Monomi [114], Cipherbase [10], AzureDB/Always Encrypted [9] and methods to process encrypted data have been designed for this purpose. However, encryption does not fully protect encrypted data, as the user's memory access pattern can leak sensitive information—recent leakage-abuse attacks [4,22,41,55,59–61,64,69,70,72–75,77,84,89,99,130] show the need to protect memory access patterns.

[1]The two first authors contributed equally to this work and their names are listed here alphabetically.

Trusted execution environments (TEEs) provide a unique opportunity for "cheap" (efficient) and "fancy" (high utility) privacy-preserving computation, i.e., any kind of computation can be supported without relying on heavy cryptographic approaches such as FHE [53,54]. However, relying on trusted hardware not only falls short of fully securing memory access patterns [97,125], but also enclave side-channel attacks (e.g., Meltdown [83], Spectre [71] and Foreshadow [115]) can lead to the extraction of enclave secrets.

Obliviousness is a strong cryptographic property that can conceal memory access patterns and side channel leakages—it ensures that algorithms within trusted hardware are data-input independent versus very powerful adversaries who can observe all the memory access patterns. By leveraging cloud outsourcing together with well-designed and implemented oblivious TEEs, companies can achieve secure, privacy-focused computing, effectively minimizing the risk of sensitive data leaks while safeguarding data confidentiality and integrity.

Towards the direction of oblivious TEE-based databases (DBs), Zheng et al. [131], Eskandarian et al. [50], and Priebe et al. [100] introduced the first oblivious relational database management system (DBMS). A notable issue with these methods is their assumption that TEEs offer specialized memory, which is invisible to adversaries in terms of memory access patterns. This assumption aligns with the old-fashioned/traditional approaches of TEE-based systems, which naively considered loading everything inside the TEE as secure, thereby overlooking potential memory or side-channel attacks. We highlight the unique contribution of Opaque by Zheng et al. [131], which presents a solution for adjusting the size of this "ideal" unobservable memory, allowing it to be reduced to the smallest possible size. Recent works have formalized the non-existence of this "ideal" specialized memory as *double-obliviousness* (in [93]), or *full-obliviousness* (in [103]), which protects data privacy even against an adversary that can observe all memory access patterns without assuming any specialized memory. We use "oblivious" to refer to fully/doubly oblivious, unless stated otherwise (see Section 2 for its formal definition).

Fully/Doubly-Oblivious TEEs Approaches & Applications. Besides Opaque, which offers an oblivious distributed data analytics platform with a wide range of SQL functionality, there are other recent works that target this setting. In particular, Mishra et al. [93] introduced a suite of Doubly-Oblivious RAM and data structures tailored for TEEs. [5, 128] focused on eliminating memory access leakage in TEEs by using doubly oblivious algorithms for both code and data. Krastnikov et al. [76] proposed the first efficient doubly-oblivious non-foreign-key join schemes (denoted as KKS). The recent work of Dauterman et al. [39] introduced Snoopy, the first high-throughput and scalable oblivious key-value store. Ghareh Chamani et al. [25] improved upon [93] for use in graph databases. Sasy et al. [103, 104] introduced new efficient approaches for shuffle and compaction. Ngai et al. [96] presented the first scalable oblivious-sort and shuffle primitives. Fredrickson et al. [52] propose a new approach for anonymous communication. This line of research, besides impacting the area of oblivious databases directly, has also found other real-world applications. These include private contact discovery for Signal [108], anonymizing Google's Key Transparency [92], Private Sampling-based Query Frameworks, and large-scale monitoring of software activities [19].

One important issue with the aforementioned works is that, despite the promise of relatively cheap TEE-based computations, combining TEEs with oblivious computation raises significant scalability issues. Except for [96], which was the first to identify this challenge, all the above-mentioned works handle relatively small data sizes of less than 6 GB. In particular in the context of Oblivious DBs, the state-of-the-art oblivious non-foreign key join [76], reported results limited to maximum dataset sizes of only 32 MB (for this case, the foreign-key join of [76] takes 6.3 seconds—this is the only number we are directly using as reported in their paper).

A similar scalability limitation is evident in various other types of oblivious queries, hence the main question we ask is:

Can we design scalable and high-performant oblivious joins and other oblivious query operators?

We consider four key query types: joins (foreign key and non-foreign key), filtering (i.e., selection), and aggregation (e.g., sum and count with the group-by clause). A foreign key join (FK) between two tables, T_0 and T_1, matches rows based on a specific attribute. In this context, T_0 is assumed to be the primary table, i.e., does not contain duplicate values for the join attribute, and every row in T_1 matches exactly one row in T_0. In contrast, non-foreign key (NFK) joins do not have this property. Both tables T_0 and T_1 may contain duplicate values in their join attributes.

Our contribution. Our primary focus is on the non-foreign-key oblivious join query, which we identify as the most fundamental problem. The new oblivious primitives that we develop for non-foreign-key joins, combined with state-of-the-art implementations for oblivious sort, shuffle, and comparison (from [96, 103]), along with new ideas, enable us not only to address non-foreign-key joins but also to significantly enhance solutions for oblivious foreign-key joins, filters, and aggregation queries. As a first step toward improving efficiency in oblivious DBs, we focus on a shared-memory setting, i.e., a single hardware enclave with an Enclave Page Cache (EPC, see Section 2) that is accessible by multiple processing units (cores). In the aforementioned test case from [76] (32 MB input/output size), our approach completes the join in 0.88 seconds with 1 thread ($\approx 7\times$ faster) and in 0.043 seconds using 32 threads ($\approx 146\times$ faster). Our detailed contributions include the following:

1. We introduce two new parallel oblivious primitives for oblivious non-foreign key join, group-by, and filter queries. These methods aim to replace slower sequential processes with faster parallel ones. The first primitive, the *Oblivious Aggregation Tree*, is an adaptation of [15] in our oblivious TEE-based setting, and operates as a parallel prefix-sum operator. It efficiently counts the occurrences of each value for a specific attribute in parallel. This method is also instrumental in duplicating values — a key step in both our non-foreign-key join algorithm and the duplication process described in [76]. The second primitive is a novel parallel oblivious expansion algorithm. This algorithm expands an input array by inserting empty spaces between elements while maintaining their original order, with the spacing determined by a second input array. Our parallel expansion algorithm avoids sorts and serves as a faster alternative to the sequential expansion method proposed in [76].
2. We demonstrate a new oblivious non-foreign-key join algorithm that is not only highly parallelizable across all execution steps, but also scalable to hundreds of GBs. Our approach is inspired by the previous state-of-the-art work from Krastnikov et al. [76], with significant improvements. In more detail, (i) we reduce the number of oblivious sorts required from five to two, which is the most resource-intensive operation, (ii) we replace the sequential scans and non-parallel distribution in [76] with more efficient parallel aggregation trees and our novel oblivious expansion, (iii) we achieve better security guarantees (see below), (iv) we achieve parallelization across all steps, enabling scalability to input sizes in the hundreds of gigabytes. This results in an evaluation size that exceeds that of [76] by over $1000\times$. These improvements make our approach superior across all experiments we performed (see Section 5)
3. We re-implemented the algorithm from [76] to improve both its efficiency and security. Specifically, we used Intel's Pin tool [68] to detect non-oblivious behavior in the C++ implementation of Krastnikov et al. under third-level compiler optimizations. We include a brief discussion of the reason behind this in Appendix A. To address this issue, we modified the algorithm to ensure obliviousness under all optimization levels while also improving its performance using our proposed oblivious primitives from Section 2.

This re-implementation, referred to as KKS* in Section 5, is experimentally compared against the original version. Our results demonstrate a $1.2\times$ speedup over the original implementation from [76].

4. We propose new parallel oblivious operators for filter, aggregation, foreign-key joins, and experimentally demonstrate their performance against the previous state-of-the-art [131]. Zheng et al., demonstrate algorithms for the aforementioned operators in a *distributed* setting. We adapt these algorithms to our shared-memory environment. Our operators outperform [131] (see Sections 4.1, 4.2 and 4.3) for the following reasons: (i) we deploy parallel algorithms in all execution steps, and (ii) reduce the number of required expensive oblivious sorts.
5. We demonstrate superior performance through extensive evaluations and have open-sourced all implementations. In detail, we evaluate the algorithms on eight synthesized and real-world datasets and benchmarks, e.g., TPC-H [112] and Big Data Benchmarks (BDB) [7]. The scaling factor of TPC-H is up to 50, and the experiment scale reached 140 GB ($2241\times$ compared to [76]). Our algorithms demonstrate better results in all tests (see Section 5). E.g., our non-foreign key join is $3.21-41.23\times$ faster than KKS. Our approach outperforms Opaque by $2.5-53\times$ on BDB queries. Our foreign key join algorithm achieves a speedup of $2.98-60\times$ against Opaque on TPC-H generated datasets. All our implementations are publicly available at [1].

Prior works. There are several prior works focusing on query execution over encrypted databases. StealthDB [116], Operon [120] and Azure SQL [9] explore encrypted databases in TEEs but use order-revealing encryption [33, 80], which leaks significantly more information. Maliszewski et al. [87] introduced a cracking-like method for radix joins on TEEs, but this exposes memory access patterns. [81] benchmarks join algorithms in various settings that do not protect against memory access patterns. Previous approaches that protect memory access patterns employ Oblivious RAMs (ORAMs) for oblivious execution [88]. For instance, Chang et al. [30] designed an index nested loop join algorithm and proposed ORAMs for joins. Avoiding the high ORAM overhead, Krastnikov et al. [76] introduced the first (sub-quadratic) oblivious non-foreign key TEE-based equi-join.

ObliDB [50] presented an oblivious database design utilizing hardware enclaves that support database queries, though it only partially protects memory access patterns. SODA [82] presents a set of oblivious algorithms for filtering and aggregation, and employs existing binary join algorithms [30, 76] in the distributed setting. Notably, their oblivious filter and aggregation, which require sequential scans, two oblivious sorts, and a non-parallel oblivious distribution, have a complexity of $O(n\log^2 n)$. Compared to this, our approach avoids expensive sorts, and is fully parallel. Our proposed algorithms for filtering, aggregation, and join can serve as drop-in replacements to further enhance their results' performance, as shown by our experimental comparison with [30,76]. We note that the distributed solution of SODA [82] introduces additional leakage compared to what we achieve here, i.e., their join execution leaks information about the values with the maximum cardinality (for each relation) participating in the join—suppressing this leakage via padding imposes a cost of $O(n^2\log^2 n)$. In Appendix B, we include a discussion of prior works that either focus on hardware enclaves and oblivious computation but not database queries specifically, or consider private database query evaluation in the MPC setting that differs from ours.

Limitations. We recognize that using a trusted enclave might be seen as a limiting factor in this field. However, the combination of TEEs with oblivious computation can be used to improve security in TEEs and is aligned with recent initiatives to strengthen trusted hardware against side-channel attacks (e.g., Keystone project [78]). Our solution is not tied to a specific TEE vendor like Intel SGX. Our implementation specifically targets SGX but is built on OpenEnclave SDK [2], which is a hardware-agnostic library that currently offers "preview" support for other alternatives (e.g., ARM Trustzone). We have not assessed the effort needed to adapt our solution to a different TEE, but we believe that our parallel oblivious query operators will be beneficial regardless of the TEE used.

A second limitation is that, although we introduce novel parallel oblivious query operators (joins, group-by, filter), our system does not support "complex" query types, in an end-to-end oblivious manner. Instead, we evaluate complex queries by first breaking them down into subqueries. Specifically, we (obliviously) compute the subqueries of a complex query independently, using the output of each one as the input for the next. This has the drawback that it reveals the intermediate sizes of each subquery (i.e., it has additional information leakage). In our experimental evaluation (Section 5), we test Obliviator on complex queries from the TPC-H [112] and BDB [7] benchmarks.

2 Preliminaries

Notation. For a relational table T, we denote accessing its attribute j in i-th tuple as $T[i,j]$. Let $[k]$ be the range $[0,k-1]$, and $[i:j]$ the range $i,i+1,\dots,j$. We denote copying the rows of B to the end of A as $A\ //\ B$ and the columns of B next to the columns of A as $A\ ||\ B$. p is the number of processing units (cores) which is sublinear to n (specifically, we assume $p \leq n/\log^2 n$). Parallel loops and subroutines are denoted as **parallel for** and **do in parallel** blocks in which the memory locations affected by each thread are independent.

Hardware Enclave. We focus on code execution within trusted hardware enclaves (e.g., Intel SGX [86], ARM TrustZone [11], AMD enclave [38]). A hardware enclave is located on an untrusted operating system and provides enhanced security functionalities that guarantee confidential computing

Operator	Algorithm	Linear	Parallel	$n=2^{10}$	$n=2^{24}$
NFK	OBL	$O(n\log^2 n)$	$O((n\log^2 n)/p)$	0.40ms	19.89s
	KKS	$O(n\log^2 n)$	$O((n\log^2 n)/p+n\log n)$	1.31ms	97.10s
	Omix++	$O(n\cdot[C\cdot\log^2 n\log\log n])$	$O(n\cdot[C\cdot\log^2 n\log\log n])$	96.05s	N/A
FK	OBL	$O(n\log^2 n)$	$O((n\log^2 n)/p)$	0.30ms	11.11s
	OPQ	$O(n\log^2 n)$	$O((n\log^2 n)/p+n)$	0.61ms	35.73s
	Omix++	$O(n\cdot[C\cdot\log^2 n\log\log n])$	$O(n\cdot[C\cdot\log^2 n\log\log n])$	47.66s	N/A
Filter	OBL	$O(n\log n)$	$O((n\log n)/p)$	0.07ms	0.86s
	OPQ	$O(n\log^2 n)$	$O((n\log^2 n)/p)$	0.23ms	9.39s
	Omix++	$O(n\cdot[C\cdot\log^2 n\log\log n])$	$O(n\cdot[C\cdot\log^2 n\log\log n])$	19.41s	N/A
Aggregation	OBL	$O(n\log^2 n)$	$O((n\log^2 n)/p)$	0.26ms	9.96s
	OPQ	$O(n\log^2 n)$	$O((n\log^2 n)/p+n)$	0.47ms	26.46s

Table 1: Complexity comparison between prior approaches and Obliviator. For brevity and readability, we use n here to denote the maximum value of input and output sizes. In Omix++ [25], C denotes the Path ORAM's bucket size. p is the number of processing units. We expand the parallel complexity by including complexities of sub-tasks that are not parallel (marked with red) in prior works to clarify that our algorithms are fully parallel. In the last two columns, we show results in milliseconds (ms) or seconds (s) for $n=2^{10}$ and $n=2^{24}$ in single-thread mode. We did not run Omix++ on $n=2^{24}$ due to the very long time needed.

(e.g., sealing, isolation, and remote attestation). Sealing is the enclave's ability to encrypt data using its own private key. Remote attestation is a service that ensures the enclave has not been corrupted or tampered with. Isolation is the secure separation of a portion of the system's memory, called Enclave Page Cache (EPC), used to store the user's data and the code to be executed. Our implementation uses Intel SGX. With Intel SGXv1 [91], the EPC size is limited to 128MB, which imposes significant performance deterioration for applications requiring additional space; when the enclave accesses a memory address outside the EPC, it must execute a special "heavy" operation. Thus, page swaps between the enclave and the untrusted OS become the bottleneck, as shown in [49].

The second version, Intel SGXv2 [90], which is the one used in our implementation, introduced flexible and dynamic EPC memory allocation features, allowing applications to scale to much larger sizes efficiently.

Obliviousness and oblivious operations. We say an algorithm is *oblivious* if, for any two same-size inputs, its execution traces, including memory accesses, are indistinguishable. A "classic" example of an oblivious algorithm from the literature is *bitonic sort* [16] (see below) that ensures the order in which all comparisons between elements are performed in a pre-determined order, independently of their actual values. Several prior works offer similar formal definitions of this property, tailored to the trusted hardware enclave setting [39, 93, 104]. At a high level, they formulate obliviousness as the existence of a simulator that can emulate the view of an adversary observing the program's execution trace and corresponding memory accesses, only given the instance size.

In particular, consider two games: a **Real** execution and an **Ideal** simulation, where the simulator will emulate algorithms with input only the leakages that the real execution produced (e.g., input and output sizes). For an oblivious operator, no adversary must be able to distinguish between the simulation and the real execution, with more than negligible probability.

Definition 1. An oblivious operator Π is secure if, for any non-uniform probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, there exists a PPT Simulator Sim such that: $\left|Pr\left[\mathbf{Real}_{\Pi,\mathcal{A}}(\lambda)=1\right]-Pr[\mathbf{Ideal}_{Sim,\mathcal{L},\mathcal{A}}(\lambda)=1]\right|\leq negl(\lambda)$, where λ is the security parameter, and $\mathcal{L}$ denotes the leakage.

Throughout the paper, we assume the existence of some fundamental oblivious operators, namely, *oblivious comparison* and *oblivious swap*. The former takes two inputs x,y and returns $1,0,-1$ if $x>y, x=y, x<y$, respectively. Meanwhile, the latter takes two inputs x,y and a single bit c and swaps them only if $c=1$. Prior works discuss different ways to build such oblivious operators, combining low-level assembly and bitwise manipulation operators, e.g., [96, 103]. In this paper, we deploy a constant-time, branch-less bitwise XOR-based oblivious swap, as explained in [96], denoted as OSwap.

Oblivious Compaction. Given an offset z and an input array A of n elements, where some of the elements are marked, a compaction algorithm will permute the table in such a way that all marked elements will appear starting from position z (traditionally $z=0$, to move all marked elements in the beginning of the array). There are works [47, 57] that propose oblivious compaction algorithms with linear complexity [47, 57], but introduce large constants, restricting their practicality. Due to their inefficiency, prior works have relied on oblivious sorting for compaction [76, 131]. However, Sasy et al. [103] demonstrated that their state-of-the-art compaction algorithm (exploiting the fact that compaction is simpler than sorting), is both asymptotically and practically faster than oblivious sorting methods. For instance, with an input size of $n=2^{24}$, oblivious sorting takes 10.80s on a single thread, whereas compaction completes in just 1.22s. Moreover, the compaction algorithm by [103] is fully parallelizable, achieving an asymptotic cost of $O((n\log n)/p)$, where p represents the number of processing units. In this work, we adopt and utilize the parallel oblivious compaction method by Sasy ORCOMPACT, which executes the following three steps:

1. Split the array into two equal halves A_l, A_r.
2. Let m be the number of marked entries in A_l. Recursively call ORCOMPACT(A_l, z), and ORCOMPACT(A_r, $z+m$).
3. Then, swap $A_l[i]$ and $A_r[i]$, where $i \in \{1,\ldots,\frac{n}{2}\}$, based on a condition c.

Note that [103] also describes a version of ORCOMPACT for non-power-of 2 inputs. Additionally, ORCompact has an order-preserving property among marked items. For instance, if i and j elements are marked, and $i < j$, then $i' < j'$ will also be true after compaction, where i', j' are the new positions of element i and j, respectively. This property will be useful in Section 4.1 when we want to split the merged table T.

Oblivious Shuffle. Given an input array of n elements, an *oblivious shuffle* algorithm will randomly permute it without revealing any information about its elements' values. Moreover, it is infeasible for an adversary to correlate input positions with output ones. At a first glance, oblivious sort can be used to randomly permute an array, by generating/sorting random positions for each element. However, as shown in [13, 96, 104], oblivious shuffling is marginally more efficient in practice but it remains asymptotically better than the best single-server oblivious sort (see below). [13] shows that an oblivious sort can be viewed essentially as an oblivious shuffle followed by a non-oblivious sort. Prior works have aimed to develop more efficient oblivious shuffling algorithms (see [13, 63, 96, 103, 104]). The recent approach proposed in [104] partially offloads the required permutation computation to an offline phase. This strategy provides a significant advantage in settings where such offline precomputation is feasible, making oblivious shuffling particularly beneficial compared to oblivious sorting. However, in our setting, we do not assume or utilize any offline phase, so we can not deploy this optimization. (Nonetheless, in scenarios allowing offline precomputation, our second oblivious filter approach would outperform the first). We deploy the oblivious shuffle introduced in [96], which results in a complexity of $O((n\log n)/p)$ as explained below. In detail, [96] performs oblivious random bin assignment based on butterfly networks. For an array with n elements, a butterfly network contains $\log n$ layers with variable number of buckets and each bucket has capacity Z. In layer k, we are shuffling elements between buckets B_i and B_{i+2^k}. The destination bucket for each element is chosen based on the $i-$th $least$ significant bit. To improve performance, [96] chooses the output buckets to be the same as the input buckets, which makes the algorithm memory-friendly (see [96] for more details). Butterfly networks are inherently scalable. In layer k, each assignment to a new bucket is independent: Pairs of buckets (separated by a distance of 2^k) can be distributed among parallel processing units, with each block assigned to buckets in layer $k+1$.

Oblivious Sort. Finally, given an array of n elements, *oblivious sort* generates as output a sorted version of the array. Bitonic sort is the most practical oblivious sorting algorithm in our single-server/multi-threaded setting (shared-memory setting), as demonstrated by previous work [96]. Although bitonic sort has a worse asymptotic complexity $O((n\log^2 n)/p)$ than alternatives like bucket sort [63, 96] ($O((n\log n)/p)$), it still outperforms them in practice on a single server with multiple threads. For example, sorting 2^{24} elements on 32 processors takes 0.64 seconds with bitonic sort, compared to 2.7 seconds with bucket sort. Consequently, we adopt bitonic sort in Obliviator, following prior works such as Opaque [131] and KKS [76]. The bitonic sort introduced in [96], is based on sorting networks and ensures the order in which elements are compared is deterministic and agnostic to the values in the array. Moreover, bitonic sort is scalable due to the independent comparisons and swaps performed at each level of the sorting network. We can assign to each processing unit n/p number of pairs to perform the computation.

We consider a version of OSORT that takes a second array as input, which serves as an "external" key for the sorting process. Instead of sorting based on the original array's entries, the sorting is performed according to this external key.

Threat Model & Leakage. We adopt the same threat as prior TEE-based approaches [25, 76, 93, 103, 104, 131], where we assume a powerful attacker who can observe the server's network traffic of encrypted data, manage the software stack outside the enclave, and control the entire operating system. However, they cannot breach the secure processor or access the processor's secret key. In this threat model, the attacker can observe memory accesses performed on the untrusted memory and those performed inside the enclave. In addition, the attacker can observe data on the memory bus, in the main memory, and in secondary storage (e.g., HDDs, SSDs). Beyond access to data in memory, the attacker has access to code traces. Attackers can use this information to implement software side-channel attacks. Other forms of side-channel leakage, such as those arising from power consumption analysis, timing attacks, and related methods [21, 58, 65, 83, 105, 115, 121], fall outside the scope of this paper. While prior works [32, 36, 62, 106, 107] provide techniques to defend against such attacks, these methods are complementary to ours and can be applied alongside our techniques to improve security further.

Output Size Leakage. Any information that can be derived by the adversary during or after the execution of our algorithm is referred to as leakage $\mathcal{L}$ similar to [26, 42, 44–46, 94]. Our threat model only allows leaking the output size $\mathcal{L} = D_{out}$ (same to prior works [40, 52, 76, 131]). This leakage is significant when operating on complex queries, as mentioned in Section 1, since we break down complex queries and perform each subquery individually.Most of our operators naturally mitigate such leakage if truncation is avoided. In particular, filtering and aggregation can hide the result size if we avoid truncation (as mentioned in Sections 3 and 4) and keep the original input size of the table, which is public information. Additionally, the size of a foreign key join will always be bound to that of the foreign table (see Section 1). However, in the case of non-foreign key joins, an alternative approach can be used–such as adjustable padding [43], or worst-case padding to hide the result sizes. This is a property of non-foreign key joins, where the output size exceeds the input size. As a result, an additional computation overhead will be added, especially when using worst-case padding, since in the worst case a binary join can result in $O(n^2)$ size. Adjustable padding mitigation, as outlined in [43], reduces leakage by concealing the exact output size, revealing only the next power of a chosen parameter x, determined by the client. For example, if $x = 2$, the adversary would observe sizes rounded up to the nearest power of 2 limiting the precision of the leaked information and reducing overall exposure.

3 Oblivious Building Blocks

Here, we discuss two oblivious algorithms that will be used in our oblivious database operators in Section 4. In detail, Section 3.1 provides a doubly-oblivious version of [15], and Section 3.2 describes our novel parallel expansion algorithm.

3.1 Oblivious Aggregation Tree

The aggregation tree described below is a variation of the prefix sum (e.g., [20, 35, 66]), differing primarily in that it performs computations (such as addition or duplication) in segments instead of the entire array. While the problem of parallel prefix sums is well-studied in research, we cannot naively adopt a prefix sum algorithm as we need one that is oblivious/data-independent and ensures the segment lengths remain concealed. More recent MPC approaches (e.g., [15]) operate on different security assumptions and threat models. E.g., in the MPC setting, a part of the computation is performed locally, which is assumed to be trusted. However, these assumptions do not align with our threat model; thus, we adopt [15] to our security guarantees and ensure the algorithm's doubly-obliviousness. Additionally, we are the first to observe the necessity of parallelizing scans and additions/duplications (see Sections 4 and 5). In particular, we observe that due to the extensive study of *parallel* oblivious sorts [96], oblivious shuffle [96, 103, 104], and compaction [103], linear scans become substantial. For instance, for $n = 2^{22}$ in a non-foreign key join, the 32-thread oblivious sort and linear scan require 0.15s and 0.22s, respectively. In the remainder of this part, we explain our oblivious aggregation tree assuming the computation is a summation over the array elements. We stress that the algorithm can naturally support other aggregation types including count, min, max, and duplication.

The algorithm expands the array elements with an additional tag bit to indicate each segment's start, denoted $B[i]$ for the i-th element. $B[i] = 0$ if the i-th element is the first element of a segment, $B[i] = 1$ in all other cases.

Our algorithm supports three modes of operation: PREFIX, SUFFIX, FULL. In detail, PREFIX computes the rolling sum

```
AGGTREE$_{\langle \text{TYPE, OP} \rangle}(D,B)$ :
WHERE TYPE = {PREFIX,SUFFIX,FULL}, AND OP = {+, DUP}
Initialize data structure $A$
parallel for $i \in [n]$ do                                   ▷Initialization
    $sx_i = D[i] \wedge B[i]$
    $A[0,i] \leftarrow \langle 0, D[i], 0, sx_i, B[i], B[i] \rangle$
end
/* We denote with subscripts the left and right children*/
for $h \leftarrow [1 : \log n]$ do                               ▷Upstream
    parallel for $i \leftarrow [1 : \frac{n}{2^h}]$ do
        $\langle lpx_i, px_i, rsx_i, sx_i, \ell_i, c_i \rangle \leftarrow A[h,i]$
        $lpx_i \leftarrow px_0$, $rsx_i \leftarrow sx_1$
        $l_i \leftarrow l_0, c_i \leftarrow c_0 \wedge c_1$
        $px \leftarrow \neg c_1 \cdot px_1 + c_1 \cdot (px_0 \bullet px_1)$
        $sx \leftarrow \neg c_0 \cdot sx_0 + c_0 \cdot (sx_0 \bullet sx_1)$
    end
for $h \leftarrow [\log n : 1]$ do                               ▷Downstream
    parallel for $i \leftarrow [1 : n/2^h]$ do
        $\langle lpx_i, px_i, rsx_i, sx_i, \ell_i, c_i \rangle \leftarrow A[h,i]$
        $px_0 \leftarrow px_i$
        $px_1 \leftarrow l_1 \cdot \neg c_0 \cdot lpx_i + l_1 \cdot c_0 \cdot (lpx_i \bullet px_i)$
        $sx_0 \leftarrow c_1 \cdot (rsx_i \bullet sx_i) + \neg c_1 \cdot rsx_i$
        $sx_1 \leftarrow sx_i$
        $A[h,i]_0 \leftarrow \langle px_0, sx_0 \rangle$
        $A[h,i]_1 \leftarrow \langle px_1, sx_1 \rangle$
    end
Initialize output array $Res$ of size $n$.
parallel for $i \leftarrow [n]$ do
    $Res[i] = B[i] \cdot px \bullet D[i] \bullet sx$
end
return $Res$
```

Figure 1: Oblivious Aggregation Tree.

of entries. For instance, Figure 2 shows the algorithm running on an input array D, with tag bits array B. The PREFIX value of the i-th element in the array will be the sum of all previous elements inside the same segment, e.g., the prefix of the third tuple is 10. SUFFIX is similar to PREFIX, but instead of calculating the rolling sum of all previous entries inside the segment, it calculates the sum of all the subsequent entries inside the segment, e.g., the first tuple of the table will have suffix equal to $7 + 9 + 2 = 18$.

Finally, FULL is computed as the sum of PREFIX and SUFFIX. Observe that to gain the final array output, we sum this result with the corresponding entry's data $D[i]$. Besides summation, we also consider oblivious aggregation for *duplication*. In this case, the operator copies the value of the first element of each segment to the entire segment, see Figure 2.

Implementation Details. We consider a binary tree where leaves are mapped to entries of D. Each node corresponds to an aggregation tuple $\langle lpx, px, rsx, sx, \ell, c \rangle$, where lpx and rsx denote its left subtree's prefix value and its right subtree's suffix value, respectively. Fields px and sx denote the current

D	B	Prefix	Suffix	Full	Dup	Res
3	0	0	18	18	3	21
7	1	3	11	14	3	21
9	1	10	2	12	3	21
2	1	19	0	19	3	21
4	0	0	8	8	4	12
8	1	4	0	4	4	12
5	0	0	0	0	5	5

Figure 2: Example of running Oblivious Aggregation Tree over array D. Dup denotes the result when performing duplication, whereas Res denotes the result of performing a full aggregation with addition.

tuple's prefix and suffix values. Then, c is the bitwise AND result of the node's children's tag bits, and ℓ is the tag bit of the left-most leaf in the sub-tree. In practice, ℓ and c are bits that control how we update the values during both phases of the aggregation tree. For example, in Figure 3, the tuple of the root of the right subtree is $\langle 21, 5, 0, 0, 0, 0 \rangle$. At the same time, the values of the children are denoted with subscripts. For instance, px_0 and px_1 are the prefix values of the left child and right child, respectively (likewise for all values in the tuple).

The pseudocode of our oblivious aggregation tree is shown in Figure 1. We mark with different colors the calculations performed for the three different types {PREFIX, SUFFIX, FULL}. We denote the aggregation operation as $\bullet$. In our work, it refers to addition or duplication, i.e., $\bullet \in \{+, \text{DUP}\}$ where DUP copies previous values to current positions as shown in Figure 2. The algorithm initializes a data structure A to store the tree nodes. For simplicity, we denote A as a two-dimensional array where the first dimension is the tree layer (with 0 being the leaf layer), and the second is the index of a node within a layer. $A[0,[n]]$ refers to the entire 0-row of A, i.e., all the tree leaves.

The algorithm consists of three phases: initialization, upstream, and downstream. For brevity, we will discuss the PREFIX type, the other two types follow similarly. During *initialization* (lines 1-4), we set the tuples of all leaves. Then, in the *upstream phase*, we update the set of internal nodes based on the tuple values of their children. We iterate with a double for-loop, layer-by-layer, and left-to-right (lines 6-13). To provide some intuition, if $c_1 = 1$, we know that all leaf nodes in the right subtree are in the same segment with some (or maybe all) the leaves in the left subtree; thus, we should update $px = px_0 \bullet px_1$. For example, in Figure 3, and focusing on node 21 (i.e., the node with $px = 21$, on level 1 of the right tree), the right subtree's control bit $c_1 = 1$, which means that all leaves in the right subtree are in the same segment with some leaves in the left subtree (in our example, with all the leaves). Thus, the value of the internal node will be updated to $px = 21$. In the other case, we propagate the prefix value of the right child to the internal node $px = px_1$. The other values are trivially updated, lpx will be equal to the prefix value of the left subtree px_0, c will be updated as $c = c_0 \wedge c_1$, and ℓ will be ℓ_0 since ℓ contains the tag bit of the left-most leaf.

In the *downstream phase*, we propagate the prefix and suffix values calculated during the upstream to the children based on their tag bits, starting from the root layer (lines 15-24). For each internal node, we first update value px_0 and then px_1. More precisely, if $\ell_1 = 0$, this means that the right subtree's left-most child is the start of a segment; thus, px_1 should be zero. In the other case, we check c_0 to see if all the leaves in the left subtree belong in the same segment. For example, in Figure 3 (b), focusing on node 21 (i.e., the node with $px = 21$, on level 1 of the left tree), its value will be updated to 5 since for the left child, we propagate the value of the parent. At the same time, node 10 will also be updated to $px = 5$, but node 11 (which is the right child of 21) will be updated to $px = 10$, since $\ell_1 = 1$ as the leaves of the right subtree are in the same segment with the leaves of the left subtree. Finally, we return the desired result in array *Res* which we populate with an iteration over all tree leaves.

Efficiency. Our algorithm, in sequential mode, clearly takes $O(n)$ since it initializes data structures of size $O(n)$ and traverses the binary tree with n leaves twice. Assuming p parallel processors, following classic results for parallel tree traversal [117], the algorithm can be run in $O(n/p)$ in the exclusive-read/write model. At a high level, all computations performed within a layer are independent of each other (they access different nodes) so each layer can be partitioned into p segments (if it has more than p nodes, else each node is processed separately) and processed in parallel.

Obliviousness and Security. This algorithm has no inherent leakage. The only information accessible to an adversary is the output size which matches the (known) input size. Thus, $\mathcal{L} = \perp$. Intuitively, the obliviousness of our algorithm is easy to prove as it performs no data-dependent branches or loops. It accesses data in all phases in a deterministic order, independent of the array values. Indeed, its memory access pattern can be fully calculated from n, and it performs simple arithmetic and logical operations or assignments; hence, its simulation is trivial (see our extended version for formal proof).

3.2 Oblivious Expansion

In this section, we propose our novel oblivious expansion algorithm. This algorithm is crucial for non-foreign key joins (see Section 4). Krastnikov et al. [76] present an algorithm that cannot be parallelized due to its iterative nature, where each iteration depends on the previous one. Attempting parallelization could lead to memory collisions and incorrect results. Furthermore, as mentioned in Section 4, expansion operates in output size granularity. In the worst case, the output size will be large enough ($O(n^2)$) to significantly reduce performance. As a result, a *parallel* expansion algorithm is required. For instance, in Figure 6(a), the algorithm will expand the array T so that each of a_1, a_2, and a_3 are two positions apart (for the expanded version of T, see the right side of Figure 6(b)). Looking ahead, this type of expansion will be very useful in the next section when we build oblivious joins.

In summary, our oblivious expansion algorithm begins by

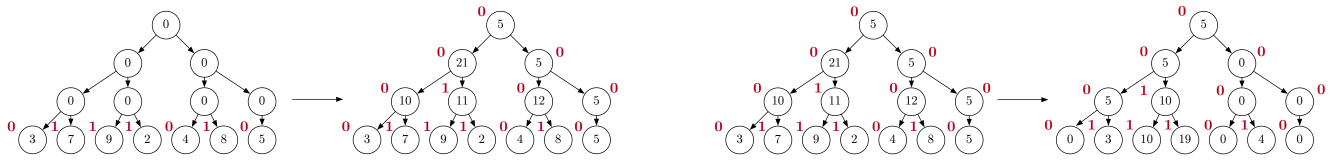


Figure 3: Visualization of our oblivious aggregation tree upstream (left) and downstream (right) phases, when executing a $\text{AGGTREE}_{\langle \text{PREFIX},+\rangle}$. Each node contains it its value px, and next to its tag bit c with a red color. For internal nodes, c is updated as $c = c_1 \wedge c_0$, where c_0 and c_1 are the tag bits of their left and right children, respectively. Note that the suffix values are never changed when performing a PREFIX; thus, we omit them for clarity. We omit ℓ since it can be derived by the leaves' tag bits.

```
T ← OExpand(T, M)
 1: $Q \leftarrow \text{AGGTREE}_{\langle\text{Prefix},\,+\rangle}(M, 0||1^{n-1})$
 2: $D_{out} \leftarrow M[n-1] + Q[n-1]$
 3: Pad $T$ until $n = D_{out}$
 4: $n_1 \leftarrow 2^{\lfloor \log_2 n \rfloor}$; $n_2 \leftarrow n - n_1$
 5: $m \leftarrow \sum_i (Q[i] > n_2)$, parallel for $i \in [n_2]$
 6: parallel for $i \in [n_2]$ do
 7:     $b \leftarrow (m \leq i)$
 8:     OSwap$(T[i], T[i+n_1], b)$
 9:     OSwap$(Q[i], Q[i+n_1], b)$
10: end
11: do in parallel
12:     OExpand$(T[0:n_2-1], M[0:n_2-1])$
13:     OExpandSub$(T, Q, n_2, n-1, (n_1 - n_2 + m) \bmod n_1)$
14: end
15: return $T$
```

Figure 4: OExpand. Oblivious expand the array. Given as input an array T, an array M that denotes the number of copies we need to generate for each element, we get the expansion of array T on arbitrary input sizes, as seen in Figure 6(b).

```
OExpandSub(T, Q, s, e, z)
 1: $mid \leftarrow \lfloor \frac{s+e}{2} \rfloor$, $n = e - s + 1$
 2: $m \leftarrow \Sigma_i (Q[i] < mid)$, parallel for $i \in [n]$
 3: if $n = 2$ then
 4:     OSwap$(T[s], T[e], m \oplus z)$
 5:     OSwap$(Q[s], Q[e], m \oplus z)$
 6: else if $n > 2$ then
 7:     $ii \leftarrow (m + z) \mod (n/2)$
 8:     parallel for $i \in [s : mid-1]$ do
 9:         $b \leftarrow (s + ii \leq i)$
10:         OSwap$(T[i], T[i+n/2], b)$
11:         OSwap$(Q[i], Q[i+n/2], b)$
12:     end
13:     do in parallel
14:       OExpandSub$(T, Q, s, mid-1, z \mod mid)$
15:       OExpandSub$(T, Q, mid, e, (z+m) \mod mid)$
16:     end
```

Figure 5: OExpandSub. This function takes as input an array T, a column with the destination indexes of each entry Q, starting and ending indexes s, e, where the range $[s:e]$ is a power of 2, and an offset z. Outputs the expanded array T.

calculating the destination indices for each element and storing them in a separate array, Q. If the required size, D_{out}, exceeds $|T|$, the algorithm pads T to match D_{out} (Figure 6(b), left side). Otherwise, it removes the necessary number of rows from the end of T. Next, it redistributes the entries of T such that the space between entry i and $i+1$ corresponds to the required number of empty entries, determined as $M[i] - 1$.

Our pseudocode for algorithm OEXPAND, is shown in Figure 4. To compute the destination index for each entry, it utilizes an oblivious aggregation tree, configuring the tag bits of the array as $B = 0||1^{n-1}$ (line 1). Hence, the result will contain the prefix sum of the whole array (i.e., $\text{AGGTREE}_{\langle\text{Prefix},\,+\rangle}$), as shown in Figure 6 (a), which is stored in an array Q. It then calculates the output size D_{out} (line 2), and pads table T with empty entries until it has length D_{out} (Figure 6 (b)). For the next part, the algorithm splits T into two parts and processes them separately. Let n_1 be the largest power of 2 such that $n_1 \leq n$, and set $n_2 = n - n_1$. The "chunks" $T[0:n_2-1]$ and $T[n_2:n-1]$ will be processed separately.

Before it proceeds, it may have to move some entries from the first chunk to the second, depending on their $Q[i]$ entry, i.e., target index (line 5). To hide how many entries were swapped, it performs oblivious swaps (lines 6-10). Finally, for the first chunk $T[0:n_2-1]$, it calls OEXPAND recursively. (Note that if the input size is a power of 2, lines 6-11 are skipped.) For the second chunk, since we know its length is a power of 2, we can go ahead and start distributing its entries via the subroutine OEXPANDSUB. This follows the same methodology, however, instead of splitting the array into the highest power of 2, it splits the array into two equal parts and progresses recursively until the processing range of T is down to 2 elements (lines 3-5). Similar to the above, it checks before the next iteration level whether it needs to swap some of the entries from the first to the second half (lines 7-12).

Efficiency. The algorithm recursively divides the input into chunks and processes them. This process is repeated up to a logarithmic number of times. During each iteration, the entire array is processed either through scans and swaps, or using our linear-time oblivious aggregation tree. Therefore, the overall complexity of the algorithm is $O(n \log n)$. Furthermore, the operations in each iteration access independent parts of the array and are fully parallelizable. With p processors, the

T	M
a_1	2
a_2	2
a_3	2
b_1	1

T	M	Q
a_1	2	0
a_2	2	2
a_3	2	4
b_1	1	6

$k_i \longrightarrow$ i-th copy of key k

(a) Using AggTree$_{\langle\text{Prefix}, +\rangle}$ to calculate the destination index for each element $q \in Q$. Each tuple of T is of the form k_i, where k denotes the key and i denotes the i-th copy of k.

T	M	Q
a_1	2	0
a_2	2	2
a_3	2	4
b_1	1	6
-	-	-
-	-	-
-	-	-

T	Q
a_1	0
-	-
a_2	2
-	-
a_3	4
-	-
b_1	6

(b) Appending empty entries until we reach D_{out}. Then, obliviously distributing the elements based on Q. Finally, we duplicate the elements to the empty entries below.

Figure 6: Visualization of Oblivious Expansion.

runtime reduces to $O((n \log n)/p)$.

Obliviousness and Security. The only observable leakage by the adversary is the output size; thus, $\mathcal{L} = D_{out}$. Recall that we deploy expansion for non-foreign key joins (see Section 4.1), hence this leakage can be suppressed by padding to the maximum output size, i.e., to $O(n^2)$ (or by applying adjustable padding, as in [43]). Intuitively, our algorithm is secure based on the following observations. (i) It entails oblivious subroutines for oblivious aggregation trees and oblivious swaps. (ii) It scans data in a predetermined order, independent of their actual values. (iii) The sizes of chunks are inferred just from the size of the input and the output (already observable by the adversary). (iv) The if-branch in OExpandSub is based on the size of the chunk known to the adversary at each repetition. A formal proof is given in our extended version.

4 Oblivious Join and Other Operators

Having introduced all the necessary oblivious algorithms we will use as sub-routines, we are now ready to present our main algorithms for oblivious parallel relational database operators, starting from oblivious joins, which is our main focus.

4.1 Oblivious Join

Description. Given two input relation tables, T_0 and T_1, a join constructs an output table that combines tuples from the input tables based on equality between values of a specific attribute j, which we will refer to as the join attribute. For instance, the query below will output a table that contains the tuples from T_0, and T_1 that have the same values on the attribute key. Figure 7 (a) demonstrates an example of the structure of the tables, and 7 (c) shows the result.

```
SELECT * FROM T0 JOIN T1 ON T0.key = T1.key;
```

T_1
$a_{1,1}$
$a_{2,1}$
$b_{1,1}$
$c_{1,1}$

T_0
$a_{1,0}$
$a_{2,0}$
$a_{3,0}$
$b_{1,0}$

T	M_0	M_1	B	C
$a_{1,0}$	3	2	0	0
$a_{2,0}$	3	2	1	1
$a_{3,0}$	3	2	1	1
$a_{1,1}$	3	2	1	0
$a_{2,1}$	3	2	1	1
$b_{1,0}$	1	1	0	0
$b_{1,1}$	1	1	1	0
$c_{1,1}$	0	1	0	0

$k_{i,j} \longrightarrow$ tuple from table j

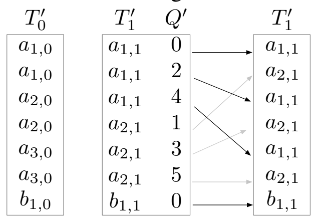
$\quad \llcorner\!\longrightarrow$ i-th copy of key k

(a) Given two relational tables T_0, T_1, append T_1 on T_0, sort based on the key and the table id. Calculate the multiplicity factors. Array B denotes the chunks that have equal value in T, whereas array C denotes the chunks with the same value and table id.

T'_0	M'_0	M'_1	C'
$a_{1,0}$	3	2	0
$a_{2,0}$	3	2	1
$a_{3,0}$	3	2	1
$b_{1,0}$	1	1	0

T'_1	M''_0	M''_1	C''
$a_{1,1}$	3	2	0
$a_{2,1}$	3	2	1
$b_{1,1}$	1	1	0
$c_{1,1}$	0	1	0

(b) Split tables back to their original form and obtain T'_0, T'_1.



T'_0
$a_{1,0}$
$a_{1,0}$
$a_{2,0}$
$a_{2,0}$
$a_{3,0}$
$a_{3,0}$
$b_{1,0}$

T'_1	Q'
$a_{1,1}$	0
$a_{1,1}$	2
$a_{1,1}$	4
$a_{2,1}$	1
$a_{2,1}$	3
$a_{2,1}$	5
$b_{1,1}$	0

T'_1
$a_{1,1}$
$a_{2,1}$
$a_{1,1}$
$a_{2,1}$
$a_{1,1}$
$a_{2,1}$
$b_{1,1}$

(c) Expand T'_0, T'_1 and duplicate the real tuples to the empty tuples among them. Finally, align T'_1 with T'_0 by computing an oblivious sort based on the ending index of each tuple.

Figure 7: Visualization of the oblivious join algorithm.

Note that the nature of such a query can reveal a substantial amount of information regarding the input tables' values, even under encryption. For instance, an adversary that simply observes the memory access pattern occurring during the query execution, learns which tuples of input table T_0 share the same join key with those of T_1. Prior work by Demertzis et al. [43] showed that this information suffices to launch devastating leakage-abuse attacks, recovering values from the (encrypted) database with considerable success rates. This makes developing oblivious join operators extremely important. We next explain the design of our oblivious joins for two cases: (i) non-foreign key joins and (ii) foreign key joins. As mentioned in the introduction, a non-foreign key join allows two tables to have duplicate join attributes while a foreign key join does not. We design our algorithms to be scalable and highly parallelizable.

Non-Foreign Key Join. Given tables T_0, T_1, the output of a non-foreign key join based on attribute j will be a table T, where every tuple $t \in T$ will be $t = t_0 || t_1$, where $t_0 \in T_0, t_1 \in T_1$, and $t_0.j = t_1.j$. Figure 7 demonstrates a join on tables T_0 and T_1 where each tuple $k_{i,j}$ contains the join attribute k, a number that denotes the copy i and a table id j. Let $n = |T_0| + |T_1|$. For simplicity, in our exposition, we assume each tuple in T_0, T_1 consists only of the join attribute, omitting the remaining "payload". The pseudocode of the operator is given in Figure 8, and its steps are as follows:

Lines 1-2: First, concatenate T_0, T_1 vertically, creating table T. Perform an oblivious sort based on j, breaking ties based on the table ids. Note that tuples are sorted so that T_0 tuples appear before T_1, if tied (see Figure 7(a)).

Lines 3-11: Initialize arrays B, C. B contains chunks of rows with the same key; C contains chunks of rows with same key and table id. Compute multiplicity factors for the entries of each table separately. To do this, use two full oblivious aggregation trees, and store the result in M_0, M_1.

Lines 12-15: Use two oblivious compactions on (copies of) T concatenated with M_0, M_1, C, to split tables back into extended versions of original T_0, T_1. The first compaction treats the elements from table T_0 as the marked elements (see Section 2), whereas in second treats the elements from T_1 (Figure 7(b)).

Lines 16-19: Having split the tables, use two oblivious expansions to place their elements to their target positions according to their corresponding multiplicity factors in the other table (here, we slightly abuse notation by calling OExpand on the first input a table instead of an array). Note that, after this step, the expanded tables (T'_0, T'_1) have D_{out} tuples, i.e., equal to the size of the result of the join.

Lines 20-21: Duplicate distributed tuples (from the previous step) in the produced tables to the empty entries among them.

Lines 22-26: Next, we must align tuples of T'_1 with those of T'_0. At this point, we observe that the i tuple of T'_1 will appear every $M''_1[i]$ rows within the chunk of rows that contain the same attribute value. To do this, we also need to consider the relative positions of tuples within these chunks, which we compute with an oblivious aggregation tree $\text{AGGTREE}_{\langle\text{Prefix, +}\rangle}$.As a result, the final index of each tuple of T'_1 in the eventual join output is calculated by a downstream scan over T'_1, with the formula in line 22. We store the final indexes in array E, and we obliviously sort T'_1 using E as the key (Figure 7(c)).

Lines 27-28: Finally, we return the join result as the concatenation of the first column of each of the two tables.

Foreign Key Join. A foreign key join involves a primary table T_0 and a foreign table T_1 consisting of primary keys of table T_0. In this setting, we do not have to create any copies or align the tuples of T_1 since if a tuple exists in T_1 with a join key k, then it is guaranteed that the same join key will exist in T_0 exactly once. We can exploit this to simplify our oblivious foreign key join, as we explain next.

Similar to non-foreign key joins, we initially concatenate both tables into a table T and sort the new table based on the join key k. Note that tuples from T_0 appear before those from T_1. Then, we perform an aggregation tree $\text{AGGTREE}_{\langle\text{Prefix, dup}\rangle}$, using the table ids to denote the segments of tuples with the same key inside T. Note that in foreign key joins, duplication happens directly without breaking the tables and performing oblivious expansion. As a final step, we perform an oblivious compaction to push all the entries from T_0 to the end of T and truncate them. From the above discussion,

```
OBLJOIN($T_0, T_1$)
 1: $T \leftarrow T_0 // T_1$
 2: $T \leftarrow \text{OSort}(T, (T.j, T.t))$      ▷$j$: join key, $t$: table id
 3: Initialize arrays $B, C, M_0, M_1$ of size $n$ with all 0's
 4: Assign $M_0[i] = 1$ when $T[i] \in T_0$, parallel for $i \in [n]$
 5: Assign $M_1[i] = 1$ when $T[i] \in T_1$, parallel for $i \in [n]$
 6: parallel for $i \in [1 : n-1]$ do
 7:     $B[i] \leftarrow (T[i].j == T[i-1].j)$
 8:     $C[i] \leftarrow (T[i].j == T[i-1].j \wedge T[i].t == T[i-1].t)$
 9: end
10: $M_0 \leftarrow \text{AGGTREE}_{\langle\text{Full, +}\rangle}(M_0, B)$
11: $M_1 \leftarrow \text{AGGTREE}_{\langle\text{Full, +}\rangle}(M_1, B)$
12: Initialize two tables $T', T''$ of size $n$
13: $T' \leftarrow T || M_0 || M_1 || C$,    $T'' \leftarrow T || M_0 || M_1 || C$
14: $T'_0 \leftarrow \text{ORCompact}(T')$       ▷based on table id $t = 0$
15: $T'_1 \leftarrow \text{ORCompact}(T'')$      ▷based on table id $t = 1$
16: Denote by $M'_0, M'_1, C'$ the second, third, and fourth
    columns of $T'_0$
17: Denote by $M''_0, M''_1, C''$ the second, third, and fourth
    columns of $T'_1$
18: $T'_0 \leftarrow \text{OExpand}(T'_0, M'_1)$
19: $T'_1 \leftarrow \text{OExpand}(T'_1, M''_0)$
20: $T'_0 \leftarrow \text{AGGTREE}_{\langle\text{Prefix, dup}\rangle}(T'_0, C')$
21: $T'_1 \leftarrow \text{AGGTREE}_{\langle\text{Prefix, dup}\rangle}(T'_1, C'')$
22: Initialize table $E$ of size $|T'_1|$
23: $Q' \leftarrow \text{AGGTREE}_{\langle\text{Prefix, +}\rangle}(1^{|T'_1|}, C'')$
24: parallel for $i \in [|T'_1|]$ do
25:     $E[i] \leftarrow \lfloor Q'[i]/M''_0[i] \rfloor + (Q'[i] \mod M''_0[i]) \cdot M''_1[i]$
26: end
27: $T'_1 \leftarrow \text{OSort}(T'_1, E)$
28: Denote $R_0$ (resp. $R_1$) the first column $T'_0$ (resp. $T'_1$)
29: return $R_0 || R_1$
```

Figure 8: Our scalable algorithm for non-foreign key joins.

our oblivious foreign-key join algorithm is much simpler than the one for non-foreign key joins. It is also considerably more efficient as it entails concretely fewer operations.

Efficiency. Both flavors of our join algorithms are asymptotically dominated by the oblivious bitonic sort that takes $O(n \log^2 n)$. Additionally, oblivious compaction and bitonic sort are fully parallelizable, as per [96, 103]. Hence, considering p parallel processors, our joins are executable in $O(n \log^2 n/p)$. The concrete performance numbers are presented in Table 1. For a non-foreign key join with $n = 2^{24}$, KKS spends 78.76 seconds on sorting, 12.99 seconds on expansion, and 5.35 seconds on scanning. In contrast, Obliviator employs the aggregation tree for scans and replaces three sorts with oblivious compaction (2) and our expansion (1). As a result, it spends 16.01 seconds on sorting, 1.44 seconds on compaction, 0.99 seconds on expansion, and 1.45 seconds on scanning. For a foreign key join, Opaque requires 35.59 seconds for sorting and 0.14 seconds for scanning. Similarly, Obliviator uses the aggregation tree for scanning and replaces one of the sorts with compaction, resulting in 9.76 seconds for sorting, 0.09 seconds for scanning, and 1.26 seconds for compaction. Besides, both in [76] and in our approach, expansion executes in output size. As a result, in queries where the output size is significantly larger than the input, the parallelization of such oblivious building block becomes substantial.

Obliviousness and Security. The algorithm for non-foreign-key joins leaks no information to the adversary besides the output size; $\mathcal{L} = D_{out}$. As in Section 3.2, we can hide the output size by padding to the worst-case output size, or we can use adjustable padding [43]. On the other hand, for foreign-key joins, our approach has no leakage since the output size is bounded by the size of the foreign-key table, which is already known to the adversary. Intuitively, the obliviousness of our algorithms follows directly from that of the used oblivious primitives. All memory accesses and computations are data-independent. See our extended version for a formal proof.

4.2 Oblivious Filter

Description. Filter queries allow users to retrieve specific subsets of elements from a database (e.g., point or range queries). For instance, the result of the following query will return only the tuples of customers that are named 'John.'

```
SELECT * FROM Customers WHERE name = 'John';
```

Oblivious filtering ensures no adversary can distinguish which tuples of the database are filtered.

Prior Approach. The standard way for oblivious filters is from Opaque [131]. (1) Firstly, a scan to mark the tuples that satisfy the condition $O(n/p)$, (2) then, it performs an oblivious sort with complexity $O((n\log^2 n)/p)$ to bring these elements to the beginning of the array and return them to the client, resulting in a total runtime of $O((n\log^2 n)/p)$.

Our Solution. We observe that the oblivious sort in the prior approach can be replaced with an oblivious compaction, which is faster both asymptotically and in practice (see Section 2). This motivates **our first approach**, yielding a total complexity of $O((n\log n)/p)$, which improves by a logn-factor over sorting/Opaque's approach. **Our second approach** involves oblivious shuffling to break dependencies in the access patterns across the table's rows, followed by a parallel non-oblivious scan.

Efficiency and Breakdown. According to Section 2, both the oblivious compaction and shuffle lead to $O(n\log n)$ complexity. Hence, our approaches result in time complexity $O(n\log n)$, which can be further reduced to $O((n\log n)/p)$, with p processing units. Our first improvement is theoretical due to the reduced time complexity. Additionally, both approaches improve the practicality of oblivious filters in the following ways. Our first approach exploits an oblivious compaction (step 2), which requires significantly less execution time compared to an oblivious sort, as mentioned in Section 2. Our second approach uses an oblivious shuffle instead of an oblivious sort. While an oblivious shuffle provides only marginal improvements over an oblivious sort, it proves beneficial in scenarios where preprocessing part of the computation is possible (see Section 2). However, since we do not operate in an offline/online setting, our evaluation reflects the results of our first approach. Nonetheless, our second approach would also outperform Opaque. Table 1 contains the complexities and concrete time when performing an oblivious filter. In particular, for $n = 2^{24}$, Opaque spends 0.05 and 9.35 seconds on scan and sort. Obliviator spends 0.04 and 0.82 seconds on scan and compaction.

Obliviousness and Security. Once again, the adversary only learns the output size; hence $\mathcal{L} = D_{out}$. However, for filter queries, we can easily hide the output size by not truncating the table but replacing the tuples that do not satisfy the query with dummies. Regarding security, both approaches are oblivious. The first one consists of oblivious compaction and scanning over the table, plus truncation per the result size. For the second, since we obliviously shuffle the table, there is no information that correlates the fact that we included, say, the j-th element with its original position in the table. A formal proof can be found in our extended version.

4.3 Oblivious Aggregation

Description. In databases, aggregation queries are performed to generate statistics or a summary of multiple rows. E.g., the query `SELECT f FROM T GROUP-BY attr` will calculate an aggregation function f over the rows of a table after they are grouped based on their values on `attr`. In practice, f may correspond, e.g., to {Min, Max, Sum, Avg, Distinct}, etc.

Prior Approach. Opaque [131] handles aggregation queries by first performing an oblivious sort based on `attr`, "forming" the groups. Then, it scans the tuples to compute f for each group. This scan has to be sequential, since in some cases (e.g., Sum), the value of each tuple depends on previous ones. Lastly, each partition performs an oblivious sort to discard the dummy rows. This approach results in a $O(n\log^2 n)$ complexity, even if the input is pre-sorted. In the parallel setting, the complexity is updated to $O((n\log^2 n)/p+n)$, as sorting can be parallelized but the scan must remain sequential.

Our Solution. Similar to Opaque, our algorithm must sort the input, in order to form the chunks. Next, we observe that the result of the aggregation function f can be computed by deploying a FULL oblivious aggregation tree. Lastly, since in aggregation queries we only require one tuple of each chunk, we use oblivious compaction based on the tag bits to filter excess tuples from each chunk to the end of the table.

Efficiency and Breakdown. Our algorithm is primarily bounded by the oblivious sorting (Section 2) in the first step required in the input, leading to an overall complexity of $O(n\log^2 n)$. Once the input is sorted, the subsequent steps of the aggregation tree (Section 3.1) and oblivious compaction (Section 2) execute in $O((n\log n)/p)$, where p denotes the number of parallel processing units. Compared to Opaque, our improved performance stems from replacing the linear scan with a parallel aggregation tree (see Section 3.1). For an aggregation query with $n = 2^{24}$, the 32-thread oblivious sort and linear scan take 0.76 seconds and 0.13 seconds, respectively. This performance gap narrows further when evaluating complex queries (see Section 5). Together with the discussion in Section 3.1, this highlights the advantages of aggregation trees as a more efficient alternative to linear scans. Moreover, we replace oblivious sort with oblivious compaction, which offers significant speedups (see Sections 2 and 4.2).

We further compare the performance of oblivious aggregation in Table 1. The breakdown times are as follows. For $n = 2^{24}$ elements, Opaque's initial sort, the linear scan, and final sort to discard dummies, take 13.17, 0.13 and 13.15 seconds, respectively. In contrast, our method achieves much lower times: oblivious sort, aggregation tree, and oblivious compaction take 9.30, 0.13, and 0.53 seconds, respectively.

Obliviousness and Security. Our solution is clearly oblivious as it consists of running an oblivious aggregation tree followed by an oblivious compaction, only revealing the input size and the result size (i.e., the number of groups); hence, $\mathcal{L} = D_{out}$. As in the previous section, we can choose not to truncate the array and replace the excess rows in a group with dummies, thus hiding the result size from the adversary. The proof can be seen in our extended version.

5 Experiments

Experimental Setup. We ran our experiments on Azure DCsv3-series with Intel's SGXv2. We used Standard_DC32s_v3 virtual machine with 3rd Generation Intel® Xeon Scalable Processor 8370C, 32 physical cores, and 256 GiB of memory, where 192 GiB is EPC, running on Ubuntu 20.04. We implemented our oblivious operators in C, using the parallel version of the oblivious bitonic sort and oblivious shuffling introduced in [96], and ORCompact [103] for oblivious compaction. For oblivious writes, swaps, and comparisons, we use the oblivious primitives from Section 2. In all experiments, implementations are compiled with -O3 optimizations unless stated otherwise. Additionally, the default block size is $B = 32$ bytes, which stores the join attribute, the tuple ID, and some metadata. Experiments were repeated five times, and the average result is reported.[1] For the rest of this section, we will refer to our implementations as OBL for our oblivious operators and FOBL for our optimization on foreign key joins. the number following an implementation denotes the number of threads used in that experiment. For instance, FOBL32 refers to the execution of our foreign key join algorithm on 32 threads.

Baselines. We compare our implementation of non-foreign key joins with that of Krastnikov et al. [76], which we will refer to as KKS. As stated in [76], KKS is not oblivious when compiled with -O3 optimizations. We used Intel's Pin Tool [67] (also used in [76]) to verify that their implementation is not oblivious in -O3, but only in -O2. We isolated that this occurs due to their implementation of oblivious writes, swaps, and comparisons (see Appendix A). We re-implemented KKS's writes, swaps, and comparisons using our approach described in Section 2. We refer to this version of [76] code as KKS*. To ensure no violations of obliviousness and to maintain fair comparisons in all subsequent experiments, KKS is compiled with -O2 optimizations, while KKS* is compiled with -O3. As demonstrated below, our modifications made KKS* both more secure and efficient.

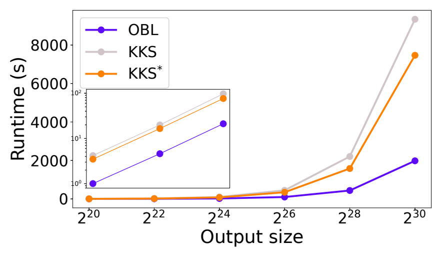
Opaque [131] (denoted as OPQ) is designed for a distributed shared-nothing setting and is not an immediate competitor to our work. However, Opaque's concepts can be adapted to our shared-memory context. Therefore, we implemented a variant of Opaque [131] for shared-memory scenarios, focusing on filter, aggregation, and foreign key queries[2].

Other approaches. Another approach for oblivious joins involves using oblivious index nested loop joins from Chang et al. [30]. Although that work targets a different setting, assuming the existence of a trusted proxy, it can be adapted to our trusted-hardware setting using an ORAM/Data Structure approach "friendly" to hardware enclaves. We tested the approach of [30] using the state-of-the-art such oblivious data structure Omix++ from [25] that proposes an improved-efficiency eviction algorithm. However, Omix++ cannot be parallelized since eviction is an atomic operation in ORAMs. Unfortunately, its performance is much worse than ours (for reference, on $n = 2^{10}$ elements on a foreign key join it requires 47.66 seconds, while our foreign key algorithm takes 0.30 milliseconds–see Table 1). Due to this, we do not include any results achieved with Omix++ to the experiments below.

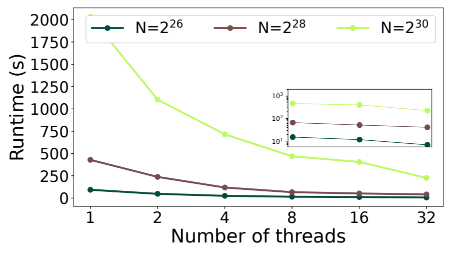
Datasets. Obliviousness is a very strong security property that prevents any information from being leaked to adversaries for the input data (except the defined leakage). For selection, aggregation (without truncation), and foreign-key joins, which are zero-leakage cases, the algorithm's behavior, including memory access patterns and performance costs, remains identical for any input of the same size. This means that any real or synthetic dataset of the same size will be indistinguishable and exhibit the same performance. For non-foreign-key joins, the same applies to any dataset with the same input and output sizes (since the join result size is leaked if padding is not used). Hence, the data contents are not important (neither for the memory access patterns nor for the performance), but the

[1]The code is available at [1].

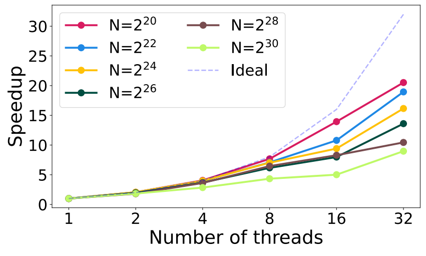
[2]Opaque relies on a mix of column-sort (used in distributed environments), tunable-oblivious sort, and sequential scan-based operators. In our shared-memory context, column-sort introduces unnecessary overhead, so we do not use it. To match our security level and prevent side-channel attacks, tunable obliviousness must be set to full (i.e., double) obliviousness. Instead of using Opaque's column sort, we employ [96]'s optimized version. To the best of our knowledge, we implemented Opaque's algorithms in the most efficient way for our setting. Interestingly, our approach could also improve Opaque in their distributed shared-nothing environment.



(a) Single thread comparison



(b) Multi-thread runtime



(c) Multi-thread speedup

Figure 9: Experiments on Synthesized Dataset. (a) demonstrates the performance of all implementations on non-foreign key joins. (b) demonstrates OBLIVIATOR's scalability. (c) illustrates the speedup relative to our single-threaded runtime across varying data sizes, with the dashed line representing ideal speedup. In (b) and (c), N refers to the output size (i.e., $N = 2n$).

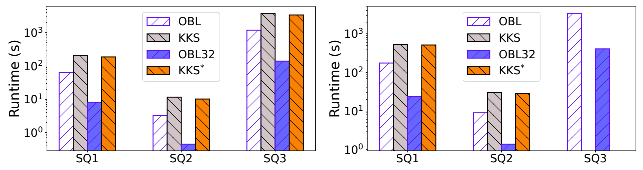


Figure 10: Evaluation of our algorithm against KKS for three non-foreign key join queries on the datasets from the TPC-H benchmark used in [30]. The left plot demonstrates scaling factor $s = 1.0$, whereas the plot on the right $s = 1.6$.

block sizes and number of elements are. For synthesized data, we first generated a dataset using a tool from [76] consisting of two tables each of size n, with a predetermined output size of $2n$, and used this for non-foreign key join queries. We also used the datasets of the TPC-H benchmark [112] and ran the same (non-foreign key and foreign ke)y equi-join queries as Chang et al. [30] (see Appendix C for details). We used larger scaling factors than [30], demonstrating the practicality of our approaches. We also used the Big Data Benchmark [7] that has been tested in prior works [50, 131].

For real-world data, we experimented on five open-source datasets: Amazon [81, 126], Jokes [56, 81], Slashdot [79, 81], IMDb [79, 81], and Twitter Social Graph (TSG) [24, 30, 82]. The first four are used in FEJ-Bench [81]. The last one was used in Chang et al. [30] with 3 join queries, SE-1, SE-2, and SE-3, which we adopt here. We note that in that work they could only use a reduced-size version of the dataset, whereas here we use the full dataset (sizes range from 11.2 MB to 19 GB), leading to a more expansive evaluation. Finally, going beyond simple joins, to evaluate our other operators, we ran more complex queries from the BDB datasets the TPC-H benchmark (namely, TPC-H queries Q3, Q5, and Q6, which include foreign key joins, filters and multiple aggregations).

5.1 Join Query Performance

Figure 9 shows the evaluation of Obliviator, KKS, and KKS* on synthesized tables generated with the tool of [76]. Tables consist of n elements, with the join results containing $2n$ elements. Input sizes range from 64MB to 64GB. In detail, Figure 9 (a) demonstrates the performance of the three implementations in a sequential setting with varying input sizes. Notably, the modifications we made to KKS not only render it oblivious but also enhance its efficiency. Additionally, the figure indicates that as the input size increases, our approach outperforms both KKS and KKS*. Specifically, our approach is $4.61-5.14\times$ better than KKS, and $3.68-3.95\times$ than KKS*. For instance, with an input and output size of 64 GB in total, our approach completes the join in 33 minutes; meanwhile, KKS takes 155 minutes, and KKS* takes 124 minutes. As mentioned in the Introduction, this improvement is expected since we reduce the number of oblivious sorts, parallelize every step of our algorithm, and use better oblivious primitives than KKS. Furthermore, Figure 9 (b) shows the scalability of our algorithm. Our runtime significantly drops ($8.96-20.51\times$) when transitioning from the sequential to the parallel setting with 32 threads, and it also performs $39.48\times$ and $41.23\times$ faster compared to KKS* and KKS, respectively. Specifically, for the same input/output (i.e., for 64 GB) size mentioned above, it completes the join in 226 seconds. Finally, in Figure 9 (c), we illustrate the speed-up of our approach compared to our sequential approach across variable input/output sizes (tested up to 64 GB for $N = 2^{30}$).

We also tested on data generated from the TPC-H benchmark and ran the non-foreign key join queries described in Appendix C (see Figure 10). Scaling factors $s = 1$ and $s = 1.6$ were chosen to ensure the output of the third query fits within the available EPC memory. The input sizes range from 610 KB to 14.65 MB, while the output sizes vary from 244.58 MB to 137.3 GB. We were unable to run KKS with scaling factors above $s = 1.0$ for the third query due to the prolonged runtime required. Figure 10 highlights the performance of our approach compared to KKS and KKS* on a single thread. Our method outperforms both approaches across all queries and scaling factors, achieving improvements of $3.21\times-3.54\times$ over KKS and $2.86\times-3.11\times$ over KKS*. These improvements are even more pronounced in the parallel setting, where our approach achieves peak improvements of $27.25\times$ and $24.31\times$ compared to KKS and KKS*, respectively.

Foreign Key Joins. While our focus has been on non-foreign key joins, we also tested a foreign key join on the same TPC-H generated dataset (also shown in Appendix C) with scaling

	FOBL	FOBL32	OPQ	OPQ32
IMDb	26.8477	2.2722	71.1326	3.7023

Table 2: Execution time (in sec) of the foreign key join FOBL, FOBL32, OPQ and OPQ32 for the real-world dataset, IMDb.

		OBL	OBL32	KKS	KKS*
TSG	SE-1	143.85	13.25	1010.43	765.20
	SE-2	281.30	23.60	1630.24	1243.44
	SE-3	2928.68	391.48	N/A	N/A
Amazon		5.78	0.54	28.09	21.69
Jokes		1.33	0.09	7.92	5.22
Slashdot		50.18	4.42	194.92	163.52

Table 3: Execution time (in sec) of non-foreign key joins OBL, OBL32, KKS, and KKS* for four real-world datasets.

factors $s = 1, 5, 10, 50$, input sizes from 229 MB to 22.35 GB, and output sizes from 366.29 MB to 17.88 GB. The top left figure in Figure 11 shows the execution times of our foreign-key oblivious approach compared to the modified Opaque described above. In a sequential setting, we are approximately $2.98\times$ faster; this speedup significantly improves in the parallel setting, where it peaks at $60\times$ against Opaque's sequential execution. Our improvement is expected since, for foreign key joins, our algorithm replaces one oblivious sort with an oblivious compaction and performs aggregations in parallel, whereas Opaque requires extra oblivious sort and linear scans.

Experiments on Real-world Datasets Regarding real data, for non-foreign key joins we compare our OBL and its 32-thread version OBL32 with KKS and KKS* [76], on the Amazon [81, 126], Jokes [56, 81], Slashdot [79, 81], and Twitter Social Graph [24, 30, 82] datasets (Table 3). For foreign-key joins, we compare our foreign key algorithm (FOBL) with Opaque and their 32-thread versions FOBL32 and OPQ32 on the IMDb dataset [79, 81] (Table 2). The total input/output size of the joins ranges between 2^{21}-2^{31} (110 MB - 67 GB).

Overall, our methods are significantly faster than prior ones, e.g., OBL32 becomes up to 77x and 57x faster than KKS and KKS*, respectively. Note that we are unable to run KKS and KKS* on SE-3 due to the very long times required. Additionally, our algorithm outperforms Opaque with speedups of $2.64\times$ and $1.7\times$ in the sequential and parallel setting.

5.2 Other Operators and Complex Queries

We tested our other operators with two queries from the BDB dataset: Q1 which is a scan query (involving our filter operator), and Q2 which is an aggregation. Furthermore, we experimented with running more complex queries consisting of multiple operators. Specifically, BDB Q3 involves filtering, followed by a foreign-key join and an aggregation. Finally, we evaluated three complex queries from TPC-H that involve series of filtering operations, followed by joins and/or aggregation. As discussed previously, for such complex queries we break them into subqueries and evaluate each operator separately. For all the above we compare Obliviator with Opaque which is the only other system that can handle such queries.

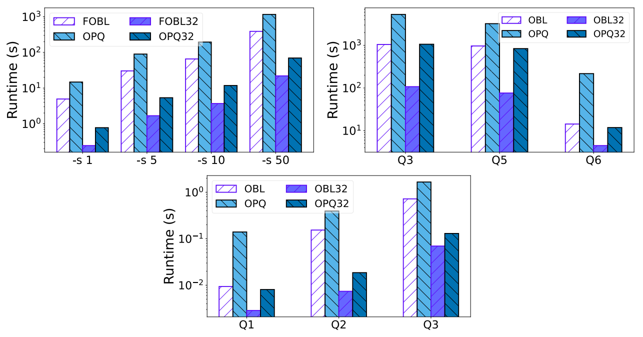


Figure 11: Evaluation of the foreign key join of Obliviator and Opaque on the TPC-H generated dataset (top left), the complex queries Q3, Q5, and Q6 from TPC-H benchmark (top right), and on Big Data Benchmark: (Q1) Scan Query, (Q2) Aggregation Query, (Q3) Complex Query (bottom).

TPC-H. We ran queries Q3, Q5, and Q6 from the TPC-H benchmark [3]. The top right figure in Figure 11 compares the performance of Opaque and our system, Obliviator, in both single-threaded and multi-threaded settings with a scaling factor $s = 50$. The input sizes range from 36.82 GB to 46.21 GB, and the output sizes vary from 840 bytes to 26.96 MB. In the sequential setting, Obliviator outperforms Opaque by $3.33 - 15.22\times$. In the parallel setting with 32 threads, Obliviator outperforms Opaque by $2.65 - 11.01\times$.

Big Data Benchmark. The bottom middle figure in Figure 11 demonstrates the performance of our oblivious operators on the benchmark's queries. In the scan query (Q1), we achieve $15.9\times$ improvement compared to Opaque. In the parallel setting, this improvement escalates to $52.3\times$ and $2.8\times$ against sequential and parallel Opaque, respectively. The improvement on a single thread stems from the practicality of compaction. However, OSORT exhibits superior scalability, narrowing the gap to $2.8\times$. For the aggregation query (Q2), we perform better in the sequential setting due to replacing one OSORT with more practical oblivious compaction, reaching a speedup of $2.6\times$. Additionally, with 32 threads, we are up to $53\times$ and $2.5\times$ faster than Opaque in sequential and parallel modes.

Finally, for Q3 we are $2.58\times$ faster than Opaque in the sequential case and up to $37.57\times$ and $11.87\times$ faster compared to 1 and 32 threads Opaque. Once again, our improvement is expected since, for all queries, we reduce the number of oblivious sorts and perform parallel executions in all steps.

6 Conclusion

We introduced OBLIVIATOR, a highly scalable oblivious relational database approach for shared memory environments. OBLIVIATOR outperforms [76, 131] both in the sequential and the parallel setting. Our main contributions are efficient algorithms for oblivious database equi-joins for non-foreign and foreign key joins, i.e., leaking no information besides the size of the input and output relations.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their constructive feedback. This work was partially supported by the Sui Foundation via an Academic Research Award.

Ethics Considerations & Open Science

Our work is in full accordance with the USENIX '25 ethics guidelines. We propose new algorithms and implement systems with a positive impact on preserving data privacy. Our experiments involved neither testing on live systems without prior consent, nor human participants. All our tests were executed either on synthetic datasets whose creation we describe or on already publicly available real-world datasets. They include TPC-H, a synthesized benchmark with created contents, a twitter social graph that is available to the public and contains the anonymized topology of the Twitter social network (also used in [24, 30, 82]), a public IMDb dataset that contains the public information of title names and actors (used in [79, 81]), a public Amazon dataset that records frequently co-purchased products (used in [81, 126]), a joke dataset that contains anonymous ratings of jokes by different users (used in [56,81]), and slashdot dataset that contains technology news website with friend/foe links between users (used in [79, 81]). We would like to point out that none of these benchmarks/-datasets can cause any type of harm and are strictly used to evaluate our algorithms. Additionally, we open-source all artifacts required for recreating our algorithms and experiments in our anonymous repository [1]. They include all our code in this paper, scripts to generate the synthesized dataset, scripts to process public benchmark and datasets, configuration information, and scripts to reproduce our evaluation.

References

[1] Obliviator. https://zenodo.org/records/14723872.

[2] Open enclave. https://github.com/openenclave/openenclave.

[3] Tpc-h benchmark. http://www.tpc.org/tpch.

[4] Mohamed Ahmed Abdelraheem, Tobias Andersson, and Christian Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. *IACR Cryptol. ePrint Arch.*, 2017.

[5] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *NDSS*, 2019.

[6] Amazon. Amazon redshift - cloud data warehouse - aws. https://aws.amazon.com/, 2024.

[7] AMPLab, University of California, Berkley. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2014.

[8] Erik Anderson, Melissa Chase, F Betül Durak, Esha Ghosh, Kim Laine, and Chenkai Weng. Aggregate measurement via oblivious shuffling. *IACR Cryptol. ePrint Arch.*, 2021.

[9] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. Azure sql database always encrypted. In *SIGMOD*, 2020.

[10] Arvind Arasu, Ken Eguro, Raghav Kaushik, and Ravi Ramamurthy. Querying encrypted data. In *ICDE*, 2013.

[11] ARM Limited. Arm trustzone technology. Technical report, ARM Holdings, 2004.

[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *OSDI'16*, 2016.

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

[14] Microsoft Azure. Azure sql - family of sql cloud databases. https://azure.microsoft.com/en-us/products/azure-sql, 2024.

[15] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. Secret-shared joins with multiplicity from aggregation trees. In *CCS*, 2022.

[16] Kenneth E Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[17] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N Kho, and Jennie Rogers. Smcql: Secure query processing for private data networks. *PVLDB*, 2017.

[18] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: efficient sql query processing in differentially private data federations. *PVLDB*, 2018.

[19] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, 2017.

[20] Guy E Blelloch. Prefix sums and their applications. 1990.

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

[22] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[23] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, M Rosu, and Michael Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.

[24] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*.

[25] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. Graphos: Towards oblivious graph processing. *Proc. VLDB Endow.*, 16(13):4324–4338, 2023.

[26] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. Dynamic searchable encryption with optimal search in the presence of deletions. *USENIX Security*, 2022.

[27] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[28] Javad Ghareh Chamani, Yun Wang, Dimitrios Papadopoulos, Mingyang Zhang, and Rasool Jalili. Multi-user dynamic searchable symmetric encryption with corrupted participants. *IEEE Trans. Dependable Secur. Comput.*, 2023.

[29] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS*, 2005.

[30] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In *SIGMOD*, 2022.

[31] Melissa Chase and Seny Kamara. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*, 2010.

[32] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *AsiaCCS*, 2017.

[33] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. Practical order-revealing encryption with limited leakage. In *FSE*. Springer, 2016.

[34] Google Cloud. Cloud sql for mysql, postgresql, and sql server. https://cloud.google.com/sql, 2024.

[35] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. 1989.

[36] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In Thorsten Holz and Stefan Savage, editors, *USENIX Security*, 2016.

[37] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, 2018.

[38] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. Technical report, 2016.

[39] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[41] Jonathan L Dautrich Jr and Chinya V Ravishankar. Compromising Privacy in Precise Query Protocols. In *EDBT*, 2013.

[42] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. *NDSS*, 2020.

[43] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX*, 2020.

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

[45] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. *TODS*, 2018.

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

[47] Sam Dittmer and Rafail Ostrovsky. Oblivious tight compaction in O(n) time with smaller constant. In *SCN*. Springer, 2020.

[48] F Betül Durak, Chenkai Weng, Erik Anderson, Kim Laine, and Melissa Chase. Precio: Private aggregate measurement via oblivious shuffling. *Cryptology ePrint Archive*, 2021.

[49] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. Benchmarking the second generation of intel sgx hardware. In *DaMoN*, 2022.

[50] Saba Eskandarian and Matei Zaharia. Oblidb: oblivious query processing for secure databases. *PVLDB*, 2019.

[51] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware oblivious ram controller. In *FCCM*, 2015.

[52] Kyle Fredrickson, Ioannis Demertzis, James Hughes, and Darrell Long. Sparta: Practical anonymity with long-term resistance to traffic analysis. In *SP*, 2024.

[53] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[54] Craig Gentry. Computing Arbitrary Functions of Encrypted Data. *Commun. of the ACM*, 2010.

[55] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In Pierangela Samarati, Mohammad S. Obaidat, and Enrique Cabello, editors, *ICETE*, 2017.

[56] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Springer*, 2001.

[57] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *ACM CCSW*, 2011.

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

[59] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, pages 315–331. ACM, 2018.

[60] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE Symposium on Security and Privacy*, pages 1067–1083. IEEE, 2019.

[61] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1353–1364. ACM, 2016.

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

[63] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. Efficient oblivious sorting and shuffling for hardware enclaves. *Cryptology ePrint Archive*, 2023.

[64] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS*, 2019.

[65] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[66] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 2007.

[67] Intel. Pin - a dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[68] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*, 2004.

[69] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[70] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[71] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *CACM*, 2020.

[72] Evgenios M Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. Leakage inversion: Towards quantifying privacy in searchable encryption. In *CCS*, 2022.

[73] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *SP*. IEEE, 2019.

[74] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *IEEE Symposium on Security and Privacy*, pages 1223–1240. IEEE, 2020.

[75] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In *SP*, 2021.

[76] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *VLDB*, 2020.

[77] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *SP*, 2018.

[78] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*, 2020.

[79] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *SIGCHI*, 2010.

[80] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *CCS*, 2016.

[81] Shuyuan Li, Yuxiang Zeng, Yuxiang Wang, Yiman Zhong, Zimu Zhou, and Yongxin Tong. An experimental study on federated equi-joins. *TKDE*, 2024.

[82] Xiang Li, Nuozhou Sun, Yunqian Luo, and Mingyu Gao. Soda: A set of fast oblivious algorithms in distributed secure data analytics. 2023.

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

[84] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 2014.

[85] Qiyao Luo, Yilei Wang, Ke Yi, Sheng Wang, and Feifei Li. Secure sampling for approximate multi-party query processing. *PACMMOD*, 2023.

[86] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. Technical report, 2013.

[87] Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. Cracking-like join for trusted execution environments. *PVLDB*, 2023.

[88] Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, Jonas Traub, and Volker Markl. What is the price for joining securely? benchmarking equi-joins in trusted execution environments. *PVLDB*, 2021.

[89] Evangelia Anna Markatou and Roberto Tamassia. Full database reconstruction with access and search pattern leakage. In Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis, editors, *ISC*. Springer, 2019.

[90] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *HASP*, 2016.

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

[92] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security Symposium*, pages 383–398. USENIX Association, 2015.

[93] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *IEEE SP*, 2018.

[94] Priyanka Mondal, Javad Ghareh Chamani, Ioannis Demertzis, and Dimitrios Papadopoulos. {I/O-Efficient} dynamic searchable encryption meets forward & backward privacy. In *USENIX Security*, 2024.

[95] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *2015 IEEE symposium on security and privacy*, pages 377–394. IEEE, 2015.

[96] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. Distributed & scalable oblivious sorting and shuffling. In *SP*, 2024.

[97] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, pages 1570–1581. ACM, 2015.

[98] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[99] David Pouliot and Charles V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS 2016*, pages 1341–1352. ACM, 2016.

[100] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. SP, 2018.

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

[102] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive*, 2017.

[103] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[104] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

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

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

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

[108] Signal. https://github.com/signalapp/, 2014.

[109] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *J. ACM*, 2018.

[110] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE SP*, 2013.

[111] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security 16*, 2016.

[112] Transaction Processing Performance Council. TPC Benchmark H (TPC-H). http://www.tpc.org/tpch/, 1992.

[113] Chia-Che Tsai, Donald E Porter, and Mona Vij. {Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX}. In *USENIX ATC*, 2017.

[114] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. 2013.

[115] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *USENIX Security*, 2018.

[116] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full sql query support. *PoPETs*, 2019.

[117] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. 2010.

[118] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *EuroSys*, 2019.

[119] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. Incshrink: architecting efficient outsourced databases using incremental mpc and differential privacy. In *SIGMOD*, 2022.

[120] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. Operon: An encrypted database for ownership-preserving data management. *PVLDB*, 2022.

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

[122] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, 2015.

[123] Yilei Wang and Ke Yi. Query evaluation by circuits. In *PODS*, 2022.

[124] Yun Wang and Dimitrios Papadopoulos. Multi-user collusion-resistant searchable encryption with optimal search time. In *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security*, pages 252–264, 2021.

[125] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SP*, 2015.

[126] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *MDS*, 2012.

[127] Chengliang Zhang, Junzhe Xia, Baichen Yang, Huancheng Puyang, Wei Wang, Ruichuan Chen, Istemi Ekin Akkus, Paarijaat Aditya, and Feng Yan. Citadel: Protecting data privacy and model confidentiality for collaborative learning. In *SoCC*, 2021.

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

[129] Yanping Zhang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. Longshot: Indexing growing databases using mpc and differential privacy. *PVLDB*.

[130] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX*, 2016.

[131] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

A Compiler Optimizations

As discussed in [76], third level compiler optimizations can break obliviousness. -O2 and -O3 are compiler optimization levels. -O2 enables a range of low-level transformations to improve efficiency without significantly increasing compilation time, while -O3 applies more aggressive optimizations, such as advanced loop scheduling, high-order loop transformations, and inlining small procedures. In designing an oblivious system, such optimizations must be carefully evaluated, as they can compromise obliviousness. In this paragraph we expand our discussion on how we experimented with Intel's Pin Tool [68] and what can lead to such result.

Intel's Pin Tool is a dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools. We used Intel's Pin Tool to analyze the memory access

```
1 void foo(elem a[], elem b[], size_t n_) {
2     for (int i = 0; i < n_; i++) {
3         a[i] = a[i].x > b[i].x ? a[i] : b[i];
4     }
5 }
```

Figure 12: An example of a ternary operation that will break obliviousness.

```
1 mov eax, DWORD PTR [rsi]
2 cmp DWORD PTR [rdi], eax
3 jg .L4
4 mov rax, QWORD PTR [rsi]
5 mov QWORD PTR [rdi], rax
```

Figure 13: Assembly code produced over line 3 of Figure 12. This assembly shows how obliviousness is broken

locations and the number of instructions during the execution of KKS. The paragraph below demonstrates why their code breaks obliviousness on certain optimization levels and what experiments were executed to prove that.

Krastnikov et al. [76] employed basic C operators (e.g., the ternary operator), which compromise obliviousness under third-level compiler optimizations. For instance, as shown in Figure 12, when performing self-assignments within a ternary operator, the compiler (with `-O3` optimization flag) ignores the assembly code corresponding to the condition where $a > b$ is true. Specifically, instructions leading to `int a = a` are omitted by a jump operation, i.e., Figure 13 Line 3 shows that if a > b, then Lines 4 and 5 are skipped. It creates a data dependency in the code traces observable by the adversary (see Threat Model in Section 2).

We tested [76]'s and our implementation using Intel's Pin Tool to compare the number of instructions executed and memory accesses performed on two different inputs: one already sorted and one randomly generated. In both cases, our code exhibited identical instruction counts and memory accesses, ensuring consistent behavior. Conversely, the implementation in [76] produced varying instruction counts and memory accesses for the same inputs, revealing a lack of obliviousness.

Nonetheless, we made sure that our implementation did not use any ternary or other operators that can harm obliviousness. A simple way to avoid such a case is by converting the ternary (or an if statement) with binary operations that are data-independent. For example, to convert the code of Figure 12 such that it does not produce a jump in the assembly, we can use binary operations (see Figure 14).

B Other Relevant Works

Here, we discuss other relevant work, not covered in our introduction.

```
void foo (elem a[], elem b[], size_t n_) {
 for (int i = 0; i < n_; i++) {
  bool cond = (a[i].x > b[i].x)
  unsigned char *a_ser = serialize(a[i]);
  unsigned char *b_ser = serialize(b[i]);
  unsigned char mask =
                ~((unsigned char) cond - 1);

  *a_ser ^= *b_ser;
  *b_ser ^= *a & mask;
  *a_ser ^= *b;

  a[i] = deserialize(a_ser);
  b[i] = deserialize(b_ser);
 }
}
```

Figure 14: Transforming a swap performed with a ternary operator to an oblivious swap using binary data-independent computations.

Searchable Encryption. There has been extensive work on Searchable Encryption [29, 31, 44–46, 124]. However, such research leaks more information and does not support updates. Hence, Searchable Encryption is extended to dynamic by introducing additional update leakage [23, 26–28, 42, 94].

Encrypted query processing in the MPC setting. There are other approaches for encrypted query processing (not based on TEEs) that work in the Multi-Party Computation (MPC) setting. The main difference to our approach is that they operate under different security assumptions (e.g., honest majority of participants, non-colluding adversaries, etc.) and they aim at optimizing different performance metrics (e.g., minimizing communication). Badrinarayanan et al. [15] proposed an aggregation tree based on prefix-sum for the honest-majority setting; our aggregation can be seen as an adaptation of theirs to the TEE setting with parallel processing. Wang et al. [123] and Luo et al. [85] propose query processing and aggregation of associative operators, also in the honest majority setting. A series of works, e.g., [18, 119, 129], introduce query optimizations via a query coordinator and database indexing combining MPC with differential privacy. Volgushev et al. [118] introduce an oblivious algorithm for joins in a hybrid protocol, where a selectively trusted party is chosen (after consensus) that gathers data from all parties and computes the result in plaintext. This algorithm can also operate under oblivious security guarantees but is rather expensive, i.e., $O(n^2)$ and $O(n\log^2 n)$ for non-foreign key joins and aggregation queries, respectively. Furthermore, Bater et al. [17] use an honest broker that collects securely computed outputs from the MPC, decrypts it, and sends it to the client. The proposed non-foreign key join algorithm leads to a $O(n^2\log^3 n)$ overhead.

Oblivious computation. Many applications and systems design oblivious computation as their core functions to ensure data security and privacy, e.g., DBMS [50, 100, 131], graph processing [25, 95], or data shuffling [8, 48] Preventing memory access leakage is critical and has been the primary focus of recent efforts. Obfuscuro [5] is the first system providing program obfuscation using trusted hardware. It utilized ORAM and Intel SGX [86], to design an oblivious array structure to store data and secure code execution for arbitrary programs. Building on this, Zhang et al. [128] propose Klotski that emulates a secure memory subsystem by utilizing ORAMs. Besides, secure multi-party computation is also used to achieve obliviousness, e.g., ZeroTrace [102]. At the same time, other oblivious approaches rely on specialized hardware (e.g., FPGA [51]), trusted-proxy [37, 101, 110], etc.

Hardware enclaves. Hardware enclaves are widely used in database and cloud computing applications [100, 127]. Many software and platforms are designed for hardware enclaves, e.g., SCONE [12], and Graphene [113]. However, hardware enclaves are still vulnerable to attacks, e.g., cache-timing [21], rollback attacks [111]. Enclave side channels allow attackers to exploit data-dependent memory accesses to extract enclave secrets. Several works are based on Intel SGX to achieve high-level obliviousness. Sasy et al. [102] presented their library of oblivious memory primitives, ZeroTrace. ZeroTrace proposes doubly-oblivious PathORAM [109] and CircuitORAM [122]. But it is outperformed by Oblix [93]. Another work that utilizes PathORAMs is Snoopy [39], that improves the scalability of PathORAMs. [39] has been utilized in Anonymous Communication applications (e.g., [52]).

C Queries on Synthetic Data

In this section we demonstrate the queries that were used in our evaluation. As mentioned in Section 5, we used the queries introduced in [30] on the TPC-H dataset. We used four queries in total, the non-foreign key join queries are the following:

SQ1.
```
SELECT s_suppkey, c_custkey, s_nationkey
FROM Supplier, Customer
WHERE s_nationkey = c_nationkey;
```
SQ2.
```
SELECT s1.s_suppkey, s2.s_suppkey,
    s1.s_nationkey
FROM Supplier s1, Supplier s2
WHERE s1.s_nationkey = s2.s_nationkey;
```
SQ3.
```
SELECT c1.c_custkey, c2.c_custkey,
    c1.c_nationkey
FROM Customer c1, Customer c2
WHERE c1.c_nationkey = c2.c_nationkey;
```

Additionally, we used an extra query on the TPC-H benchmark to evaluate the foreign key join on our algorithm with scaling factors $s = 1$ and $s = 50$, which can be found below:

```
SELECT l_orderkey, o_orderkey
FROM LineItems, Orders
WHERE l_orderkey = o_orderkey;
```