An Oblivious General-Purpose SQL Database for Cloud Computing

Paper # XXX

Abstract

We present ObliDB, a secure SQL database for the public cloud that supports both transactional and analytics workloads and protects against access pattern leakage. With many database workloads moving to the cloud, there is significant interest in securing them. Hardware enclaves offer a strong practical foundation towards this goal by providing encryption and secure execution, but they still suffer from access pattern leaks that can reveal a great deal of information. The naive way to address this issue—using generic Oblivious RAM (ORAM) primitives beneath a database—adds prohibitive overhead. Instead, ObliDB co-designs both its data structures (e.g., oblivious B+ trees) and query operators to accelerate SQL processing, giving up to 329× speedup over naive ORAM. On analytics workloads, ObliDB ranges from competitive to 19× faster than systems designed *only* for analytics, such as Opaque, and comes within 2.6× of Spark SQL. Moreover, ObliDB also supports point queries, insertions, and deletions on tables of 1 million rows in 3.6-9.4ms, making it usable for transactional workloads too.

1 Introduction

Relational databases are a lynchpin of modern computer applications, ranging from low-volume services inside a business to global social applications such as Facebook. With the advent of cloud computing, there is considerable interest in running databases securely in the cloud, protecting their sensitive content from both network attackers and insiders at the cloud provider (e.g., a hacker who breaches the cloud provider's security [47]). Researchers have proposed approaches including property-preserving encryption [23, 35, 36], secure hardware [3, 54], and algorithms to execute specific computations on encrypted data [33, 50], offering various tradeoffs between security, generality and performance.

One of the most promising practical approaches to increase security is hardware enclaves such as Intel SGX [15]. These enclaves provide an environment where a remotely verifiable piece of code can run without interference from the OS, accessing a small amount of enclave memory and making upcalls to the the OS when needed. However, applications using enclaves to manage a large amount of data must still access it through the OS (e.g., to read new memory pages or access the disk), which makes them susceptible to access pattern attacks. For database workloads in particular, access patterns can reveal a great deal of information even when the data is encrypted [14, 26, 31]. Some recent systems, such as Opaque [54] and Cipherbase [3], have proposed oblivious execution schemes that do not reveal access patterns, but these systems are limited to *analytics* workloads that scan entire tables to answer a query. Specifically, both systems use oblivious sort operators that sort all the data. These systems would not be efficient for more general database workloads that also include transaction processing (point queries and updates to a few records)—the most common use case for databases.

This paper presents ObliDB, an oblivious SQL database that supports both transactional and analytical processing using hardware enclaves. Unlike previous systems, ObliDB supports *two storage methods*—linear tables that must always be scanned, and *oblivious B+ trees* that allow efficient indexed access. Each table can be stored using one or both methods to achieve higher performance on the desired query workload.

The key idea in ObliDB's oblivious B+ trees is to use techniques from Oblivious RAM (ORAM) [46] to support fast lookups and updates to just part of the database, unlike previous systems that always scan the whole data. However, naively applying ORAM to a database engine (e.g., using a platform like ObliVM [29] that changes every memory access to use ORAM) results in prohibitively high overheads. Instead, ObliDB revisits tradeoffs inherent in the design of database indexes and optimizes for the oblivious setting, carefully co-designing both the data structures used to hold records and the processing algorithms for SQL operators to perform several orders of magnitude faster than naive ORAM.

The new data structures present in ObliDB allow new and diverse algorithms for SQL operators. ObliDB provides multiple versions of each operator based on the input and output representations, as well as on data properties such as output sizes. Finally, ObliDB can select the fastest implementation of each operator based on data statistics at runtime.

Together, these properties allow ObliDB to support a wide range of queries efficiently while not leaking any information beyond intermediate result sizes and the chosen query plan (the same security level as Opaque's oblivious mode).[1] ObliDB supports selections, aggregations and joins similar to other analytics systems [3, 54], as well as efficient low-cardinality operations, such as point lookups, insertions, deletions and updates.

[1] ObliDB also supports padding intermediate and final results, similar to Opaque's pad mode, if desired.

We implement a prototype of ObliDB and evaluate it on real and synthetic datasets of various sizes. We first compare ObliDB to a baseline implementation where a database index is generically modified to run over ORAM, and show that ObliDB outperforms it by up to 329×. For analytics workloads, we compare ObliDB to Opaque's oblivious mode [54] on the Big Data Benchark [2] and find that ObliDB is competitive with Opaque on most queries, but can also outperform Opaque by 19× on queries that can leverage indexes. ObliDB also comes within 2.6× of Spark SQL [5], which provides no security or privacy guarantees. For index workloads, ObliDB outperforms the recent Sophos encrypted search scheme [10] that does not hide access patterns by over 22×. Finally, we show that the choices of oblivious data structures and algorithms available in ObliDB enable meaningful optimizations during the query planning process.

The rest of this paper is organized as follows: Section 2 gives an overview of ObliDB and our security model. Section 3 gives background on relevant tools used in ObliDB, and Sections 4 and 5 detail ObliDB's design. Sections 6 and 7 describe our implementation and evaluation respectively, and Section 8 discusses related work before concluding in Section 9.

2 Overview

This section summarizes the functionality and architecture of ObliDB, its threat model, and the security properties it achieves.

2.1 Threat Model

We assume an attacker with full control of the operating system (OS), including power to examine and modify untrusted memory and any communication between the processor and memory. Moreover, the attacker can observe access patterns to trusted memory and maliciously schedule processes or interrupt the execution of an enclave. We note that an attacker can always launch an indefinite denial of service attack against an enclave, but such an attack does not compromise privacy and lies outside the scope of the security of the hardware enclave for our purposes.

We assume security of the trusted hardware platform in that the enclave hides the contents of protected memory pages from a malicious attacker with complete control of the operating system. While some attacks have been demonstrated against SGX [27, 51], a number of standard mitigations exist to handle these attacks [37, 42–44], and other hardware enclave designs avoid the pitfalls that leave SGX vulnerable [16, 28, 30].

Furthermore, we also assume a secure channel exists through which an outside user can send messages to the enclave. A client can establish such a connection, e.g. through TLS.

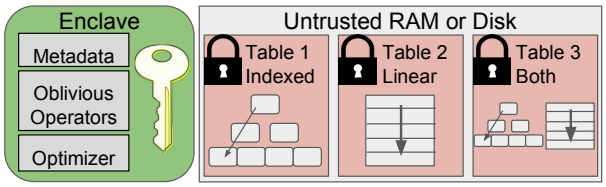

Figure 1: ObliDB provides an interface to a secure enclave with control over encrypted tables stored in untrusted memory. It stores tables either as an oblivious index or a linear scan data structure to ensure data-oblivious queries.

2.2 Security Goals

Queries in ObliDB leak only the sizes of tables involved, including intermediate tables, and the query plan used. This includes the sizes of tables in the database, the sizes of queries, and the sizes of responses to queries and provides the same level of security as Opaque's Oblivious Mode [54]. ObliDB additionally features a padding mode where sizes of all tables are padded to some chosen size. Further details regarding how to achieve these leakage properties appear in Sections 4 and 5, where the leakage of each operator is explained. Data at rest outside the enclave is encrypted and MACed and leaks only the size of the encrypted data. We do not make an effort to hide the number of tables in a database or which table(s) a particular query accesses.

We additionally make the integrity guarantee that ObliDB catches and reports any tampering with data by the malicious OS. We use a series of checks and safeguards to protect against arbitrary tampering within rows of a table, addition/removal of rows, shuffling of the contents of a table, or rollbacks to a previous system state. We discuss these protections in Section 4.

2.3 ObliDB Architecture Overview

Figure 1 shows an overview of the ObliDB architecture. ObliDB consists of a trusted code base inside an enclave that provides an interface for users to create, modify, and query tables. ObliDB supports two *storage methods* for each table: Linear and Indexed. It stores tables, encrypted, in unprotected memory and obliviously accesses them as needed by the various supported operators. Indexed tables consist of an ORAM with a B+ tree stored inside, whereas Linear tables scan the whole table on each query to ensure obliviousness.

ObliDB supports oblivious versions of the SQL operators `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `GROUP BY` and `JOIN` as well as the aggregates `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`. Each operator is implemented for both Linear and Indexed tables. Finally, ObliDB includes a query optimizer that can choose the right operator plan for each query. For example, for selection queries, the optimizer first determines the size of the selection and then executes the best-performing `SELECT` algorithm for the data to be returned. Choices include algorithms that take advantage

of the possibility of caching small results inside the enclave or quickly handling very large return sets by making a copy of the original table and returning it whole with the few missing rows obliviously erased.

2.4 Limitations

ObliDB provides strong privacy guarantees with practical performance for a broad range of applications but was not designed as a distributed database system. Although this may render it unsuitable for certain high-performance settings, the vast majority of web applications only require a single-server database solution with the ability to efficiently handle diverse queries. This constitutes the target population ObliDB hopes to entice as users in order to bring heightened privacy to everyday web users.

We note in this vein that however secure the properties of a database management system, an application interacting with it can leak additional information. For example, if a web application using ObliDB makes a second query to a database based on the results of a first query, observing the size of the response to the second query may leak additional information about the first query or its response. This direct, if unexpected, consequence of size leakage requires that application developers consider performance goals against the ramifications of such leakage in their design process.

3 Background

In this section we give a basic overview of Intel SGX and ORAM, the primary tools used in ObliDB, providing only sufficient detail for the subsequent sections.

3.1 Intel SGX

SGX provides developers with the abstraction of a secure *enclave* which can verifiably run a trusted code base (TCB) and protects its limited memory from a malicious or compromised OS [1, 15]. SGX handles the process of entering and exiting an enclave and hiding the activity of the enclave while non-enclave code runs, albeit imperfectly [27]. Enclave code invariably requires access to OS resources, so SGX provides an interface between the enclave and the OS based on *OCALLs* and *ECALLs*. OCALLs are calls made from inside the enclave to the OS, usually for procedures requiring resources managed by the OS, such as access to files on disk. ECALLs allow code outside the TCB to call the enclave to execute trusted code.

SGX proves that the code running in an enclave is an untampered version of the desired code through a mechanism named *attestation*. Attestation involves an enclave providing a hash of its initial state which a client compares with the expected value of the hash and rejects if there is any evidence of a corrupted or altered program.

The most significant feature of SGX for our purposes concerns the protection of memory. SGX provides the developer with approximately 90MB of Enclave Page Cache (EPC), a memory region hidden from the OS and cleared whenever execution enters or exits an enclave. In this memory, the enclave can execute trusted code and keep secrets from a malicious OS who otherwise controls the machine executing the code.

3.2 ORAM

Oblivious RAM, or ORAM, is a cryptographic primitive first proposed by Goldreich and Ostrovsky [24] that hides access patterns to data in untrusted memory. In the traditional ORAM setting, a small trusted processor uses a larger memory over a bus on which an adversary may examine communications. Merely encrypting the data that travels over the bus still reveals the access patterns to the data being requested and can be used to glean private information about the data or the queries on it [26]. ORAM goes further and shuffles the locations of blocks in memory so repeated accesses to the same block and other patterns are hidden from the observing adversary. ORAMs guarantee that any two sets of access patterns of the same length are indistinguishable from each other. More formally, the security of ORAM is defined as follows:

Definition 1 (ORAM Security [46])**.** *Let* $\overrightarrow{y} :=$ ***((op**$_M$**, a**$_M$**, data**$_M$**),...,(op**$_1$**, a**$_1$**, data**$_1$**))** *denote a data request sequence of length M, where each* **op**$_i$ *denotes a* ***read(a**$_i$**)*** *or a* ***write(a**$_i$**, data)*** *operation. Specifically,* $\mathbf{a}_i$ *denotes the identifier of the block being read or written, and* $\mathbf{data}_i$ *denotes the data being written. Index* 1 *corresponds to the most recent load/store and index M corresponds to the oldest load/store operation.*

Let $A(\overrightarrow{y})$ *denote the (possibly randomized) sequence of accesses to the untrusted storage given the sequence of data requests* $\overrightarrow{y}$*. An ORAM construction is said to be secure if:*

1. *For any two data request sequences* $\overrightarrow{y}$ *and* $\overrightarrow{z}$ *of the the same length, their access patterns* $A(\overrightarrow{y})$ *and* $A(\overrightarrow{z})$ *are computationally indistinguishable by anyone but the client ORAM controller.*
2. *The ORAM construction is correct in the sense that it returns on input* $\overrightarrow{y}$ *data that is consistent with* $\overrightarrow{y}$ *with probability* $\geq 1 - negl(|\overrightarrow{y}|)$*, i.e., the ORAM may fail with probability* $negl(|\overrightarrow{y}|)$*.*

The scope of the security guarantees provided by ORAM create important consequences for oblivious data structures and algorithms built on top of it. ORAM only makes guarantees of indistinguishability for access patterns of the same length. This means that oblivious algorithms using ORAM must always make the same number of memory accesses or risk leaking access patterns.

Although other, older schemes have recently received attention due their practical efficiency in certain practical

parameter settings [52], the most efficient ORAM scheme known is the Path ORAM [46]. Path ORAM belongs to a family of schemes known as tree-based ORAMs, which operate by storing the blocks of the oblivious memory in a tree structure. Each block is associated with a leaf in the tree in a position map that guarantees the block will be found somewhere on the path to that leaf. An access to the ORAM involves reading a path down the tree from the root to the leaf corresponding to the desired block. After retrieving the desired block, a second pass is made on the same path where each block is re-encrypted with new randomness and the retrieved block is assigned a new leaf, remaining stored in a small "stash" if the path does not allow space for it to be written back on the path to its new assigned leaf. Although it is not always necessary in practice, the position map holding the assigned leaves for each block of the ORAM can be recursively stored in its own ORAM to reduce the trusted processor memory required by this scheme to a constant.

4 ObliDB's Storage Methods

ObliDB stores data at rest in two types of tables: Linear and Indexed. This section discusses each storage method and the security considerations involved in building algorithms for operators over them.

ObliDB creates tables with an initial maximum capacity that can be increased later by copying to a new, larger table. Data can be represented by a Linear structure, an Indexed structure, or both in parallel, so as to take advantage of the properties of both table types. Since tables are stored in unprotected memory, ObliDB independently encrypts and MACs every block of each data structure with a symmetric key generated inside the enclave. For both storage methods, it stores each row in one block of the corresponding data structure and reserves the first byte of each block as a flag to indicate whether that block contains a row or is empty.

4.1 Linear Tables

A Linear type table simply stores rows in a series of adjacent blocks with no additional mechanism to ensure obliviousness of memory accesses. This constitutes a "trivial" ORAM where every read or write to the table must involve accesses to every block of the structure in order to maintain obliviousness of access patterns. As such, operators acting on such a tables, as will be seen in Section 5, involve a series of linear scans over the entire data structure. This data structure performs best with small tables, tables where operations will typically require returning large swaths of the table, or aggregates that involve reading most or all of the table regardless of the need for obliviousness. The challenge in designing algorithms for Linear tables lies in using the right data structures within the limited space of the enclave to reduce the number of scans and data processing operations involved in each operator.

4.2 Indexed Tables

Indexed type tables make use of both an ORAM and a B+ tree in order to provide better performance without losing obliviousness for large data sets. The data structure consists of an ORAM that holds a B+ tree where the actual data of the table resides, with each node of the B+ tree corresponding to one block of the ORAM.

Memory Management. Since the structure of a B+ tree changes dynamically as rows are added and removed from a database, the B+ tree implementation must use some form of dynamic memory management and pointers between nodes in the tree. In order to accommodate this, we implement equivalents of malloc, free and the pointer dereference operator for our ORAM. Our memory allocator consists of an array of flags that we set if the corresponding block is in use and unset if it is not. This increases the protected memory needed over the ORAM's position map by 20% but does not represent a dramatic increase in memory requirements over the total space needed by ObliDB for the position map, ORAM stash, and other elements of system state recording the names, sizes, and types of existing tables.

New Tradeoffs. The key insight in making this data structure efficient is that an ORAM underneath the B+ tree alters the usual performance tradeoffs inherent in B+ trees in unexpected ways. A typical B+ tree may have pointers in each node to its parent as a low-cost solution to avoid a search down the tree to find a node's parent, and setting the maximum degree of nodes in the tree depends on a balance between achieving low depth overall and not having too many pointers to examine and update within a single node. ORAM upsets the balance on which these intuitions rely by increasing the cost of following or updating pointers between nodes – resulting in an ORAM lookup – to the point where many operations within a single node become almost free by comparison to following a pointer. As such, the degree of a node can become as large as can fit inside an ORAM block well before the cost of operations within a node reaches parity with the cost of an ORAM operation. This, however, clashes with the parent-saving optimization because although keeping track of each node's parent saves a handful of pointer lookups every time a parent must be accessed, the cost of updating the parent for each node's children far outweighs the savings accrued when nodes are split or merged. Note that since we wish to preserve obliviousness in all queries, every B+ tree operation takes on its worst-case running time every time it runs, and the costs of usually uncommon splits and merges of nodes must be paid on every insertion or deletion to an Indexed table. To reduce the number of writes needed to the ORAM for each operation,

Method	Linear	Index	Combined
Space	N	~$4N$	~$5N$
Point Read	$O(N)$	$O(log^2N)$	$O(log^2N)$
Large Read	$O(N)$	$O(NlogN)$	$O(N)$
Insertion	$O(1)$	$O(polylogN)$	$O(polylogN)$
Deletion	$O(N)$	$O(polylogN)$	O(N)

Figure 2: Asymptotic performance of each storage method.

ObliDB operates on a lazy write-back principle where it does not write changes to the ORAM until the last possible moment. In sum, rethinking common sense B+ tree optimizations for the ORAM setting results in drastically improved performance.

4.3 Practical and Security Considerations

Choosing Storage Methods. ObliDB currently has system administrators manually decide which storage format to use for each table, a decision that can easily be made depending on the kinds of queries that are expected to be run on the data. Indexed tables perform best on small reads that access one or a few rows of a table, whereas queries which expect to return large segments of a table should use Linear tables. Using both types of table, while incurring the cost of both for insertions and deletions, proves effective when queries of diverse selectivities must run on the same data Figure 2 compares the asymptotic operations of standard operations on each table type.

Obliviousness. Although the security properties of ORAM guarantee that two access transcripts of the same length will be indistinguishable from each other, designers of oblivious algorithms for operators over Indexed tables must ensure that the total number of accesses or the timing gaps between accesses do not leak any private data. For example, the property of the B+ tree that all data resides in the leaves of the tree, always at the same depth, means that any search in the tree will make the same number of accesses to intermediate nodes before finding the desired data. Using a different data structure that does not exhibit such a property would compromise the obliviousness of our operators on Indexed tables. On the other hand, standard insertion and deletion operations for B+ trees could leak information about the internal structure of the tree because they involve splitting and merging nodes when they reach fixed threshold numbers of children. Even in the case of a Linear table, satisfying a `SELECT` query via a straightforward scan that copies each row matching the given criteria into an output table does not provide obliviousness despite touching every row in the table. We address considerations in ensuring obliviousness for each operator in Section 5.

Data Integrity. Although encryption and oblivious data structures/algorithms ensure the privacy of data in ObliDB, additional protections make certain that an attacker does not tamper with data. Such tampering could take the form of tampering within rows of a table, addition/removal of rows, shuffling of the contents of a table, or rollbacks to a previous system state. ObliDB protects against such attacks and reports any attempt to tamper with data.

ObliDB MACs and encrypts every block of data stored outside the enclave, preventing the OS from modifying rows or adding new rows to tables. This leaves the possibility of duplicating/removing rows, shuffling rows, or rolling back the system state. Included in each block of MACed data is a record of which row the block contains and its current "revision number," a copy of which ObliDB also stores inside the enclave. Any attempt to duplicate, shuffle, or remove rows within a data structure will be caught when an operator discovers that the row number of data it has requested does not exist or does not correspond to that which it has received. Spoofing a fake revision number requires either breaking the security of the MACs used or breaking the security of the enclave to modify the stored copy, neither of which lies within the power of an attacker in our model. Rollbacks of system state are caught when the revision numbers of rows in a table do not match the last revision numbers for those rows recorded in the enclave. These lightweight protections suffice to discover and block any malicious tampering of data in ObliDB.

5 Oblivious Operators

In this section we describe the various oblivious operator algorithms used in ObliDB. ObliDB provides support for a large subset of SQL, including insertions, updates, deletions, joins, aggregates (count, sum, max, min, average), groupings, and selection with conditions composed of arbitrary logical combinations of equality or range queries. Moreover, depending on known information about the size of a response to a query, ObliDB can choose which algorithm to use in order to maximize performance in each situation. We will begin by discussing algorithms for Linear tables and then discuss the modifications or entirely different solutions used for Indexed tables. Each operation will be accompanied by a security argument.

The following notation will be used in subsequent paragraphs: the table being returned will be referred to as R, and the table being selected from will be referred to as T. The number of rows in R is represented by r, the number of rows in T is N. r' and N' represent the number of blocks in the data structures holding R and T, respectively.

5.1 Linear Tables

Insert, Update, Delete. Insertions, updates, and deletions for Linear tables involve one pass over the table, during which any unaffected block receives a dummy write and affected blocks are written to as follows:

Insertion: ObliDB offers two options for insertions into Linear tables. First, it can conduct a linear scan

of a table, making a dummy write on each row except for the first unused block it encounters, in which it will make a real write. In tables with few deletions, a fast insertion algorithm keeps track of the last row where an insert occurred and always inserts directly into the next row.

Deletion: any row matching the deletion criteria will be marked as unused and overwritten with dummy data. Deletions and updates support arbitrary conditions, similar to selections, so any logical combination of conditions on equality or inequality of entries in a row is acceptable.

Update: any row matching the update criteria will have its contents updated instead of a dummy write.

All of the above operations, with the exception of fast insertion, leak nothing about the parameters to the query being executed or the data being operated on except the sizes of the data structures involved because they consist of one linear scan over a table where each encrypted block is read and then written with a fresh encryption. Fast insertion also leaks no additional information beyond the sizes of tables because the access pattern of the insert does not depend at all on the content of the data except on the number of insertions made, which our adversary can already learn by observing the sizes of tables over time.

Select. Our Select algorithm begins by scanning once over the desired table and keeping a count of the number of rows that are to be selected. This step leaks only the size of the table T. Then, based on the size of the output set and whether the selected rows form one continuous block in the table or not, it executes one of several strategies:

Naive: included as a baseline for comparison, the naive oblivious algorithm mirrors a straightforward translation of a non-oblivious `SELECT` to an oblivious one via an ORAM. After examining each row, it executes an ORAM operation. If the examined row is to be included in the output, it makes a write. If not, it makes a dummy read (reading an arbitrary block). After completing the scan of the input table, it copies the ORAM to a Linear table which it returns.

Our techniques to improve upon this baseline consist of finding the right balance between using data structures inside the enclave to remove the need for an ORAM and making multiple fast passes over data. These ideas constitute the guiding principle in designing our remaining `SELECT` algorithms and choosing between them.

Continuous: Should the rows selected form one continuous section of the data stored in the table, ObliDB employs a special strategy which requires only one additional pass over the table. First, it creates table R with r rows. Then, for the ith row in table T, if that row should be in the output, it writes the row to position i mod r of R. If not, it makes a dummy write. Since the rows of R are one continuous segment of T, this procedure results in exactly the selected rows appearing in R.

In addition to the sizes of tables T and R, the fact that ObliDB chooses this algorithm over one of the other options leaks the fact that the result set is drawn from a continuous set of rows in the table. Users concerned about this additional leakage could disable this option and use one of the other options with no reduction in supported functionality. The execution of the algorithm itself is oblivious, however, because the memory access pattern is fixed: at each step, the algorithm reads the next row of T and then writes to the next row of R.

Small: In the case where r is small, that is, where all the rows of table R only require a few times the space available in the enclave, a selection strategy that makes multiple fast passes over the data proves effective. We take multiple passes over table T, each time storing any selected rows into a buffer in the enclave and keeping track of the index of the last checked row. Each time the buffer fills, its contents are written to R after that pass over T. Although this strategy could result in a number of passes linear in the size of R, it proves effective for small r, as demonstrated in Section 7.

This algorithm leaks only the sizes of tables T and R because every pass over the data consists only of reads to each row of the table and the number of passes reveals only how many times the output set will fill the enclave, a number that can be calculated from the size of R, which we reveal anyway.

Large: If table R contains almost every row of table T, we create R as a copy of T and then make one pass over R where each unselected row is marked unused and each selected row receives a dummy write.

The copy operation reveals no additional information about T or R because it could be carried out by a malicious OS with no input from the enclave or a user. The process of clearing unselected rows involves a read followed by a write to each block of the table, so it also reveals no information beyond the size of T. This algorithm, in fact, does not even reveal the size of the output set R because we pad the data structure to the size of T.

Hash: In the case that none of the preceding special-case algorithms apply, ObliDB uses the following generalization of the continuous strategy. We wish to apply the technique used for continuous data on data that may be arbitrarily spread throughout T, not just in one continuous block. Our approach is to resort to a hashing-based solution. For the ith row in T, if the row is to be included in the output, we write the content of the row to the $h(i)$th position in R, for some hash function h.

The algorithm as stated above does not exactly represent how ObliDB works because we need a few changes in order to ensure, first, that we properly handle hash collisions to ensure correctness, and, second, that we maintain obliviousness in handling collisions. In order to main-

tain obliviousness, every real or dummy write to R must involve the same number of accesses to memory. This means that if any write resolves in a collision, every write must make as many memory accesses as in the case of a collision. Following the guidance of Azar et al [8], we use double hashing and have a fixed-depth list of 5 slots for each position in R. This means that for each block in T, there will be 10 accesses to R, 5 for each of the two hash functions.

The modifications above ensure that data access patterns are fixed regardless of the data in the table and which rows the query selects. Since the hash is taken over the index of the row in the data structure and not over the actual contents of a row, information about the data itself cannot be leaked by access patterns when rows are written to R. As such, only the sizes of T and R leak. The selection strategy also leaks, but this information can be deduced just from knowledge of the sizes of T and R and therefore leaks no additional information.

Aggregates & Group By. A baseline solution to aggregation queries performs very poorly, but we can compute aggregates far faster than selection if we do it correctly and only require one oblivious pass over a Linear table to do so. An aggregate over a whole table or some selected subset of a table requires only one pass over the whole table where we calculate the aggregate cumulatively based on the data in each row. A naive approach uses an oram to keep track of the aggregate and needs to access it for every row, causing an unnecessary slowdown. We achieve better performance by keeping the aggregate statistic inside the enclave and avoiding the ORAM overhead. Since the memory access pattern of this operation always involves sequential reads of each block in the data structure, nothing leaks from this operation beyond the size of table T.

We handle groupings similarly to aggregates without groupings, except we keep an array inside the enclave that keeps track of the aggregate for each group where a naive solution would check an entire array via oram for each row of table T. The method for determining which group each row belongs to is handled differently for low and high-cardinality aggregation:

Low-Cardinality: In the low-cardinality setting, we make a linear scan over all known group values in order to check for a match. If we find no match, we create a new group.

High-Cardinality: Linearly scanning over all known groups becomes prohibitively expensive as the number of groups becomes larger, so high-cardinality groupings employ a hash table where each group's value is hashed and inserted into a hash table held in the enclave. Each row scanned is hashed and checked against the table. If there is a match, then the row under examination corresponds to a known group referenced in the table, and if not, then the current row is added as a new group.

Join. We implement the join functionality for Linear tables as a variant of the standard hash join algorithm [19]. We refer to the two tables being joined as T_1 and T_2. The Join proceeds by making a hash table out of as many rows of T_1 as will fit in the enclave and then hashing the variable to be joined from each row of T_2 to check for matches. This process repeats until reaching the end of T_1 . After each check, a row is written to the next block of an output linear table. If there is a match, the joined row is written. If not, a dummy row is written to the table at that position. This algorithm reveals the sizes of the tables T_1 and T_2, but not the size of the output table, which is padded to a parameter representing the maximum possible size by dummy rows (which can be set to less than $|T_1| * |T_2|$ if desired). Since each comparison between the two tables results in one write to the output structure regardless of the results of the comparison, the memory access pattern of this algorithm is oblivious.

5.2 Indexed Tables

Operations for Indexed tables largely behave similarly to those for Linear tables, except all the operations take place over the ORAM and B+ tree data structure described in Section 4. The important difference between the two lies in the fact that the index can be used to restrict a search to a particular relevant area of a table without having to scan every row to maintain obliviousness. The use of an index, however, comes with some security ramifications. In the case that the block of rows accessed by a query are a continuous set beginning and ending with a specified value of the index column, no additional information leaks because knowledge of the size of the portion of the table scanned is equivalent to knowledge of the size of the query output. On the other hand, if the rows returned by a query are not continuous, the leakage also includes the size of the segment of the database scanned in the index. For example, supposing that there is one student named Fred in a table of students and student IDs, the query `SELECT * FROM students WHERE NAME = ''Fred'' AND ID > 50 and ID < 60` leaks not only that the size of the result set is 1 but also that 9 rows were scanned in the execution of the query. We consider this leakage to be included in the sizes of intermediate tables, as a query plan that selects a noncontinuous segment from an Indexed table is equivalent to one which selects a continuous segment from an Indexed table and then selects a noncontinuous segment from the returned table. This leakage, like all such leakage, can be hidden by padding.

Insert, Update, Delete. Insertions and Deletions for Indexed tables pad the number of operations made on the underlying ORAM so no information can be leaked about the internal structure of the B+ tree being modified. As discussed in Section 4, this means every B+ tree operation

has worst-case running time and that design decisions involved in constructing the trees and operations on them differ from the traditional setting without ORAM. Updates on segments of an Indexed table behave similarly to Linear tables.

Select, Aggregates, & Group By. Indexed tables make use of the same selection, aggregation, and grouping algorithms as linear tables, with the major difference being the use of the index to find the correct part of a table to scan. The Large strategy for selection does not apply to Indexed tables because the strategy of copying the whole table is not as applicable where a query aims at a small fraction of the table and the data are not stored in consecutive blocks but across multiple nested tree-based data structures.

Join. Since the rows of an Indexed table are always sorted by the index column in the leaves of the B+ tree, it is possible to efficiently sort-merge join two tables with the same index [19]. Tables T_1 and T_2 are scanned at the same time, and any matching rows are placed in an output ORAM, just as for Linear tables. More specifically, at each step, the next row of each of T_1 and T_2 is read. If the rows match, the pointer on the right table advances and there is a write to the ORAM, and if they do not match, a dummy write takes place and the pointer on the table with the lesser value advances. This process proceeds until pointers reach the end of both tables. Obliviousness holds because each step of the algorithm consists of exactly one read to each of T_1 and T_2 and one write to an ORAM, and the total number of steps is only a function of the sizes of the three data structures involved.

5.3 Complexity

Prior systems that implement oblivious operators include Opaque [54], Cipherbase [3], and the Oblivious Query Processing algorithms of Arasu and Kaushik [4]. All three works focus on oblivious algorithms for analytic queries and only propose algorithms that involve scans over entire tables. The approach to this kind of operator typically involves a combination of oblivious sorts and filters. In contrast, our work uses new ideas to achieve similar functionalities for both Linear and Indexed tables, providing support for a broader set of general database use cases. Whereas sort and filter based approaches always have complexity $O(N)$ or $O(NlogN)$ in the size N of a table, ObliDB's solutions range in complexity from $O(1)$ to $O(N^2)$, but our optimizer picks the algorithms it expects will perform best in practice regardless of asymptotics.

6 Implementation

Our implementation of ObliDB includes the linear scan and oblivious B+ index from Section 4 as well as the oblivious operator algorithms described in Section 5. It consists of over 14,000 lines of code of which approximately 10,000 are new and builds upon the Remote Attestation sample code provided with the SGX SDK [1] and the B+ tree implementation of [7], the latter of which was heavily edited in order to support our ORAM memory allocator. We used the libraries made available by the SGX SDK for encryption, MACs, and hashing. We will make our implementation of ObliDB open source and publicly available online.

Table Name	Rows	Notes
CFPB	107,000	Customer complaints to the US Consumer Financial Protection Bureau [20].
USERVISITS	350,000	Server logs for many sites. Part of the Big Data Benchmark data set [2].
RANKINGS	360,000	URLs, PageRanks, and average visit durations for many sites. Part of the Big Data Benchmark data set [2].

Figure 3: Real data used in our evaluation and comparisons.

We tuned ObliDB's parameters for the protected memory space made available by the SGX enclave. We chose a nonrecursive PATH ORAM [46] as our choice of ORAM scheme. The nonrecursive ORAM can fit up to about 15 million rows before needing a second layer of recursion in order to fit the position map in an SGX enclave, so ObliDB can handle realistic data sets without any need for a recursive ORAM. That said, there is no reason why ObliDB cannot be modified to make use of recursive ORAM at a modest performance penalty for Indexed tables. Moreover, the ObliDB implementation allows for easy swapping of ORAM schemes through a common interface, so our choice of ORAM can easily be replaced, say, to optimize the ORAM scheme used to fit the data as in [52].

7 Evaluation

In this section we evaluate ObliDB on tables of up to 1.4 million rows and measure its performance against a baseline implementation that naively modifies a database to use ORAM as well as existing private database systems. A summary of real-world data sets used in our experiments appears in Figure 3. We also measure the overhead of ObliDB's padding mode and demonstrate the effectiveness of ObliDB's query optimizer as well as the efficacy of Linear and Indexed tables in different situations through a series of microbenchmarks. We evaluated ObliDB on a desktop computer with an Intel Core i7-6700 CPU @3.4GHz and 8GB of RAM running Ubuntu 16.04.2 and the SGX Linux SDK version 1.9 [1].

We find that ObliDB dramatically outperforms a baseline implementation and can leverage its indexes to achieve order of magnitude performance improvements over previous private database systems. In particular, ObliDB matches Opaque [54] for scan-based queries but

Data Set	Query	ObliDB	Baseline	Speedup
	Linear Selection			
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2013-05-14`	1.192s	34.79s	29.2×
RANKINGS	`SELECT pageURL, pageRank FROM RANKINGS WHERE pageRank > 1000`	2.434s	46.33s	19.0×
	Index Selection			
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2013-05-14`	0.472s	0.678s	1.4×
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2017-08-17` (point query)	0.0027s	0.0033s	1.5×
RANKINGS	`SELECT pageURL, pageRank FROM RANKINGS WHERE pageRank > 1000`	0.082s	0.107s	1.3×
	Index Insertion/Deletion			
CFPB	`INSERT INTO CFPB (Complaint_id, Product, Issue, Date_received, Company, Timely_response, Consumer_disputed) VALUES (4242, "Credit Card", "Rewards", 2017-09-01, "Bank of America", "Yes", "No")`	0.011s	0.708s	64.4×
CFPB	`DELETE FROM CFPB WHERE Bank="Bank of America" WHERE Date_Received=2013-05-14 LIMIT 1`	0.224s	0.451s	2.0×
CFPB	`DELETE FROM CFPB WHERE Bank="Bank of America" AND Date_Received=2017-08-17` (point query)	0.015s	0.220s	15.0×
	Aggregates and Joins			
CFPB	`SELECT COUNT(*) FROM CFPB WHERE (Product="Credit card" OR Product="Mortgage") AND Timely_Response="No" GROUP BY Bank`	0.595s	110.3s	185.4×
USERVISITS	`SELECT SUBSTR(sourceIP, 1, 8), SUM(adRevenue) FROM USERVISITS GROUP BY SUBSTR(sourceIP, 1, 8)`	3.042s	>1000s	>328.7×
RANKINGS USERVISITS	`SELECT sourceIP, totalRevenue, avgPageRank FROM (SELECT sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue FROM Rankings AS R, UserVisits AS UV WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN Date('1980-01-01') AND Date('1980-04-01') GROUP BY UV.sourceIP) ORDER BY totalRevenue DESC LIMIT 1`	12.774s	>1000s	>78.3×

Figure 4: Comparison of ObliDB and a baseline where a naive oblivious database implementation directly ports non-oblivious algorithms to their oblivious counterparts via ORAM. ObliDB outperforms the baseline on all queries. Our aggregate and join queries were run over Linear tables because they involve reading most or all rows of the relevant tables.

can outperform it by 18.8x when it uses an index. ObliDB also performs 22.6-24.6× faster than Sophos [10], a recent index-based searchable encryption scheme.

7.1 Comparison to Baseline

ObliDB outperforms a baseline implementation (i.e. what could be achieved with a generic tool for converting legacy applications) by as much as two orders of magnitude. Since a direct translation of the memory accesses of a non-oblivious data structure to an ORAM does not guarantee obliviousness, we modified its data structures and algorithms as little as possible to achieve an oblivious version, always erring on the side of stronger performance. Our baseline uses the same data structure as ObliDB for Linear tables but uses a naive B+ tree implementation that does not take advantage of the ORAM-related optimizations discussed in Section 4. The baseline also uses the naive varieties of operators as described in Section 5.

Figure 4 compares ObliDB to the baseline oblivious database. ObliDB achieves up to 29× speedup for `SELECT` queries and over 328× speedup for aggregates. For comparison to an insecure system, a point selection query in ObliDB performs only 7.9× slower than the same query run on mysql over a table stored in memory. `SELECT` queries over Linear tables enjoy much larger speedup over the baseline than the same queries over Indexed tables because an oblivious B+ tree lookup takes most of time in the indexed `SELECT` queries, and we used the same algorithm for this lookup in both the baseline and actual implementations. We made this decision because a naive application of ORAM to a B+ tree search algorithm does not yield an oblivious B+ tree, as described in Section 4.

The most staggering speedups over the baseline appear in aggregation queries, where ObliDB gains two orders of magnitude in performance. This arises from the need

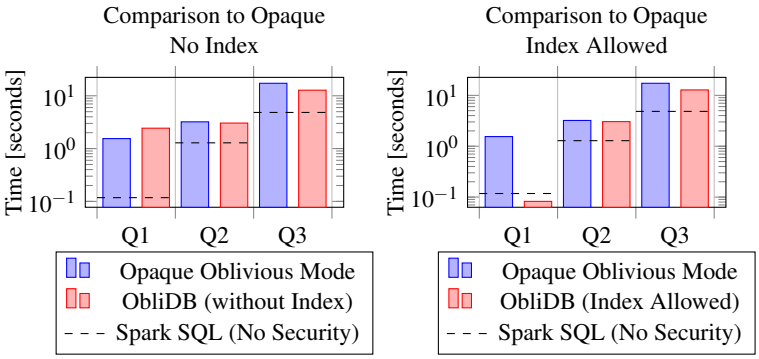

Figure 5: ObliDB outperforms Opaque Oblivious [54] by 1.1-18.8× and never runs more than 2.6× slower than Spark SQL [5] on Queries Q1-Q3 of the Big Data Benchmark [2]. Even without use of an index, ObliDB performs comparably to Opaque Oblivious.

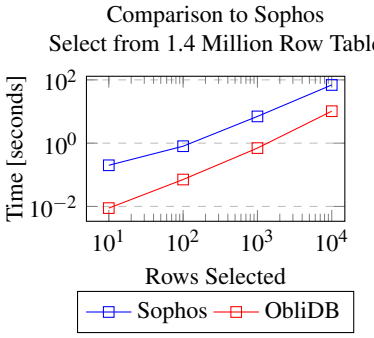

Figure 6: Comparison to Sophos SSE scheme [10]. ObliDB always outperforms Sophos by at least 22.6×. Unlike ObliDB, Sophos leaks access patterns to data.

to hide data structures that keep statistics for each group without revealing when a row does not match with any known groups and needs to begin its own new group. The possibility of this occurrence necessitates, in the naive algorithm, an access to each group's data for each row. With the system-wide maximum number of groups set to the hundreds of thousands, such a query has little chance of completing within a reasonable time frame and may take well over the 1,000 seconds at which we cut off our experiments. The aggregation over the CFPB table completes in a shorter period of time because we used our prior knowledge of the number of banks to set the maximum number of groups to a lower threshold (200, in this case).

7.2 Comparison to Opaque

Figure 5 compares ObliDB with Opaque's oblivious mode [54] and Spark SQL [5] on the first three queries of the Big Data Benchmark [2] on tables of 360,000 and 350,000 rows. We omit the benchmark's fourth query as neither ObliDB nor Opaque support the external scripts needed for it. Opaque, like ObliDB, centers its design on a secure SGX enclave and can be configured in either an "encryption" mode, which leaks access patterns but offers performance close to Spark SQL and an "oblivious" mode that hides access patterns to data by making sure to fit sensitive memory accesses inside the trusted enclave memory and achieving a security level similar to ours, albeit by very different means. Spark SQL provides no security guarantees and provides a measure of the performance achievable without any security concerns. Both Opaque and Spark SQL are run in a single node configuration on our device.

ObliDB, when allowed to use indexes, outperforms Opaque on all three BDB queries, ranging from 1.1× speedup on query 2 to 18.8× speedup on query 1. The strong performance of ObliDB compared to Opaque and Spark SQL on query 1 is due to ObliDB's oblivious index, which allows it to only examine a small portion of the table whereas Opaque and Spark SQL, which aim to primarily handle analytic workloads, scan the entire table to satisfy the query. Indexes do not provide a speedup on queries 2 and 3 because those queries require scanning larger segments of the input tables. Even without using an index to speed up query 1, ObliDB performs comparably to Opaque. Moreover, for queries 2 and 3, although ObliDB is slower than Spark SQL, it is only 2.4× slower on query 2 and 2.6× slower on query 3, putting ObliDB safely in the realm of tools with practical performance for real applications.

7.3 Comparison to Sophos

We compare the performance of ObliDB's oblivious index to the searchable symmetric encryption (SSE) scheme Sophos [10] in Figure 6. Sophos does not provide obliviousness guarantees, meaning it leaks access patterns to rows in a table, and it does not use hardware enclaves to achieve security. It does, however, offer a "forward-secrecy" property not found in many SSE schemes. As such, it provides a good point of comparison for the performance of our SGX-based oblivious Indexed tables with a non-SGX based, non-oblivious index that still provides reasonable privacy guarantees. Sophos only provides the ability to search for exact keyword matches via an inverted index, so we compare on simulated data. Specifically, we compare against the performance numbers reported in the original Sophos paper for a table of 1.4 million rows and measured using a more powerful machine than ours: an Intel Core i7 4790K 4.00GHz CPU with 8 logical cores, 16GB of RAM, a 250 GB Samsung 850 EVO SSD, running on OS X.10. Despite the difference in hardware and the fact that the Sophos implementation is multithreaded, ObliDB outperforms Sophos by 22.6-24.6× depending on the size of the number of rows returned by a query. We observe that the performance tipping point between Indexed and Linear tables in ObliDB arrives between the 10^4 and 10^5 rows returned marks in this experiment, and ObliDB's performance on larger queries beyond that point

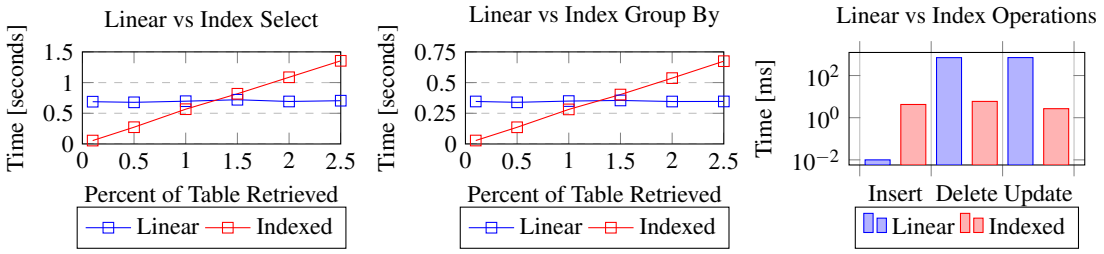

Figure 7: Comparison of Linear and Index versions of operators over 100,000 rows of fabricated data. Linear scans do better when more of the data needs to be accessed, but Indexed structures perform far better for small queries.

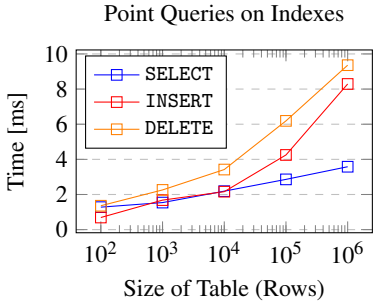

Figure 8: Point queries for Indexed tables of various sizes. Query time grows polylogarithmically in table size.

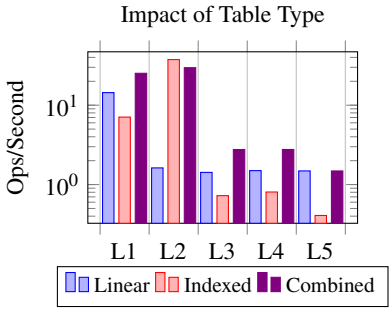

Figure 9: Linear, Indexed, and combined representations of data performing on various workloads over a 100,000 row table. Point reads access 1 record, small reads access 50 records, and large reads access 5% of the table.

Workload	L1	L2	L3	L4	L5
% Point Reads	5	0	50	45	0
% Small Reads	0	90	0	0	0
% Large Reads	5	0	50	45	90
% Insertions	90	9	0	5	5
% Deletions	0	1	0	5	5

would remain constant.

7.4 Padding Mode Overhead

ObliDB's padding mode hides not only access patterns to data but also all information regarding the sizes of tables, whether they be at rest in the database, intermediate tables constructed in handling a query, or the results returned from a query. We evaluated the performance cost of this mode by running queries on the CFPB table of 107,000 rows padded up to 200,000 rows. We found that our aggregate query over a Linear table had a 4.4× slowdown and a select had a 2.4× slowdown. The larger slowdown for aggregates results from the padding algorithm padding to the maximum supported number of groups for aggregates – in this case, 350,000. We did not evaluate the padding mode for Indexed tables because the entire benefit of Indexed tables results from the knowledge of the selectivity of a query, the exact information padding removes by hiding table sizes.

Opaque [54] describes an oblivious pad mode, but this mode remains unimplemented. To our knowledge, no other comparable system exists with an implemented oblivious padding mode, so we are unable to compare our padding performance to prior work. The results do, however, represent reasonable slowdowns for inflating the size of a table by approximately 2×.

7.5 Table Representation Choices

By providing both Indexed and Linear tables and optimizing queries based on information gained through a first pass over the data, ObliDB to makes meaningful performance improvements for diverse queries. To this end, Figure 7 compares the performance of Linear and Indexed tables on `SELECT` (hash algorithm), `GROUP BY` (low-cardinality), `INSERT`, `DELETE`, and `UPDATE` queries. Linear scans perform better as the amount of data retrieved from a table increases since the cost of the scan is amortized over more rows, but smaller queries perform significantly better using an index. This finding contrasts claims in prior work [37] that indicate linear scans always perform better than ORAM for oblivious memory access. In general, Indexed `DELETE`, and `UPDATE` queries significantly outperform Linear tables, but the fast Linear `INSERT` query – the one measured in the figure – outperforms the Indexed `INSERT`. The performance of linear tables (outside of fast insertions) degrades linearly with the size of the table, but point operations on Indexed tables take time only polylogarithmic in the table size. Figure 8 shows the running time of insertions, point selections, and point deletions as the size of a table grows by factors of 10. This very gradual increase in running times enables the performance improvements ObliDB enjoys over oblivious analytics systems on queries that admit efficacious use of an index.

In some situations, a combined table representation that maintains both an Indexed and Linear table for the same data proves effective. Although ObliDB pays insertion and deletion costs for both types of tables to maintain a combined table, it can use the better representation for each query, an important benefit given that many real-world workloads rely heavily on different kinds of reads.

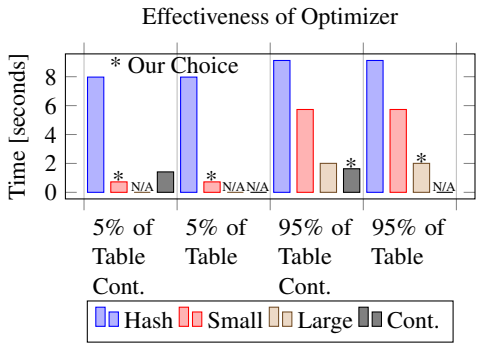

Figure 10: Our optimizer picks the best algorithm for handling `SELECT` queries based on a preliminary scan that determines whether the data to be returned is small, large, or consists of a continuous set of rows in the table.

Figure 9 shows ObliDB running various workloads with Linear, Indexed, or combined tables. Although there exist cases in which a Linear or Indexed table alone undisputedly perform best, a combined representation often performs comparably to or much better than either table type alone.

7.6 Effectiveness of Optimizer

Figure 10 demonstrates the effectiveness of ObliDB's choice of `SELECT` algorithms, comparing our various algorithms on queries that retrieve 5% and 95% percent of a fabricated 100,000 row table. Although the "Hash" algorithm performs the best asymptotically, the figure demonstrates that knowledge gleaned only from ObliDB's intended leakage about the results of a query (whether it is small/large or a continuous set of rows) suffices to pick an algorithm that will perform much better in practice. Equally impressive gains appear in the choice of `GROUP BY` algorithm for real queries: the high-cardinality aggregation algorithm used for the query on the USERVISITS table (shown in Figure 4) performs 168× better than the low-cardinality algorithm would on the same query.

8 Related Work

Cryptographically-protected database search. Fuller et al [23] summarize and systematize prior work on cryptographically-protected database search. The widely-known CryptDB [35] implements a tradeoff between security and performance by encrypting each field according to the security needs of that column. Arx [34], a more recent system, keeps all data encrypted at the highest level of security and makes clever use of data structures to allow for efficient operations over data. Another common class of solutions are those which use an inverted index to allow searches on stored encrypted data, as exemplified by Demertzis et al [17]. These schemes rely on searchable symmetric encryption (SSE) as a primitive, recent examples of which include Sophos, Janus, and Diana [10, 11].

Cryptographically protected databases have been subject to attacks [14, 26, 31, 53] showing that inference from known context or leakage can compromise schemes in ways not covered by their original security models.

Trusted hardware. A number of generic tools provide legacy applications the heightened security available from SGX [6, 9, 25, 45]. In addition to general tools, many works implement variations of existing tools and services rendered secure via SGX. M2R [18] and VC3 [40] provide MapReduce and cloud data analytics functionalities, respectively, and Opaque [54] provides secure support for Spark SQL. Like Opaque, Cipherbase [3] provides a secure and optionally oblivious variant of Microsoft's SQL Server. SecureKeeper [13] uses SGX to build a confidential version of Apache's ZooKeeper (`zookeeper.apache.org`). HardIDX [22] builds a database index in SGX, and ZeroTrace [39] provides oblivious memory primitives based on ORAM.

Side channel attacks on SGX make use of page faults [12, 51], branch history [27], thread scheduling [49], and other "controlled channel" side channels, where an attacker uses its control of the system to learn more about the enclave's behavior than it should know. Recently, Schwarz et al [41] show how malicious code running inside an enclave can attack other enclaves. Shinde et al [44] and Raccoon [37] close side channels by making the memory trace of a program oblivious or obfuscated. SGX-Shield [42] enables address space layout randomization for SGX, and, finally, T-SGX [43] protects against side-channel attacks by using Transactional Synchronization Extensions (TSX) to close side channels.

Intel SGX has been used to implement practical functional encryption [21] and obfuscation [32], neither of which could be constructed efficiently previously. Outside SGX, there exist other hardware solutions that render programs' memory traces oblivious [16, 28, 30].

Data Structures over ORAM. Wang et al. [48] develop several general data structures that can be used efficiently on top of ORAM. Desiring a map data structure with a history independence property, Roche et al. [38] design a "HIRB tree" that they build on top of vORAM, an ORAM variant that offers variable-sized blocks.

9 Conclusion

We have presented ObliDB, a cryptographically-protected database system based on Intel SGX that leaks only table sizes and its query plan (and even hides table sizes in pad mode). We have shown that ObliDB handles practical data sets with performance surpassing prior work that had similar or weaker security properties. It is our hope that solutions like ObliDB based on SGX and other techniques that leverage hardware-based advantages can enable rapid advances in the performance and security of solutions to difficult problems related to private databases and search over encrypted data.

References

[1] Intel software guard extensions sdk for linux os, developer reference. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf.

[2] AMPLAB, UNIVERSITY OF CALIFORNIA, B. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[3] ARASU, A., BLANAS, S., EGURO, K., KAUSHIK, R., KOSSMANN, D., RAMAMURTHY, R., AND VENKATESAN, R. Orthogonal security with cipherbase. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings* (2013).

[4] ARASU, A., AND KAUSHIK, R. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.* (2014), pp. 26–37.

[5] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., AND ZAHARIA, M. Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), pp. 1383–1394.

[6] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M., GOLTZSCHE, D., EYERS, D. M., KAPITZA, R., PIETZUCH, P. R., AND FETZER, C. SCONE: secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* (2016), pp. 689–703.

[7] AVIRAM, A. F. Interactive b+ tree (c). http://www.amittai.com/prose/bplustree.html.

[8] AZAR, Y., BRODER, A. Z., KARLIN, A. R., AND UPFAL, E. Balanced allocations. *SIAM J. Comput. 29*, 1 (1999), 180–200.

[9] BAUMANN, A., PEINADO, M., AND HUNT, G. C. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst. 33*, 3 (2015), 8:1–8:26.

[10] BOST, R. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (2016), pp. 1143–1154.

[11] BOST, R., MINAUD, B., AND OHRIMENKO, O. Forward and backward private searchable encryption from constrained cryptographic primitives. *IACR Cryptology ePrint Archive 2017* (2017), 31.

[12] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.* (2017).

[13] BRENNER, S., WULF, C., GOLTZSCHE, D., WEICHBRODT, N., LORENZ, M., FETZER, C., PIETZUCH, P. R., AND KAPITZA, R. Securekeeper: Confidential zookeeper using intel SGX. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016* (2016), p. 14.

[14] CASH, D., GRUBBS, P., PERRY, J., AND RISTENPART, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 668–679.

[15] COSTAN, V., AND DEVADAS, S. Intel SGX explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[16] COSTAN, V., LEBEDEV, I. A., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 857–874.

[17] DEMERTZIS, I., PAPADOPOULOS, S., PAPAPETROU, O., DELIGIANNAKIS, A., AND GAROFALAKIS, M. N. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 185–198.

[18] DINH, T. T. A., SAXENA, P., CHANG, E., OOI, B. C., AND ZHANG, C. M2R: enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), pp. 447–462.

[19] ELMASRI, R., AND NAVATHE, S. B. *Fundamentals of Database Systems (6th Edition)*. Pearson, 2010.

[20] ENIGMA. Consumer complaints. https://app.enigma.io/table/us.gov.cfpb.consumer-complaints.

[21] FISCH, B. A., VINAYAGAMURTHY, D., BONEH, D., AND GORBUNOV, S. Iron: Functional encryption using intel sgx. *IACR Cryptology ePrint Archive 2016*.

[22] FUHRY, B., BAHMANI, R., BRASSER, F., HAHN, F., KERSCHBAUM, F., AND SADEGHI, A. Hardidx: Practical and secure index with SGX. In *Data and Applications Security and Privacy XXXI - 31st Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017, Proceedings* (2017), pp. 386–408.

[23] FULLER, B., VARIA, M., YERUKHIMOVICH, A., SHEN, E., HAMLIN, A., GADEPALLY, V., SHAY, R., MITCHELL, J. D., AND CUNNINGHAM, R. K. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (2017), pp. 172–191.

[24] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious rams. *J. ACM 43*, 3 (1996), 431–473.

[25] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016,*

Savannah, GA, USA, November 2-4, 2016. (2016), pp. 533–549.

[26] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* (2012).

[27] LEE, S., SHIH, M., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *CoRR abs/1611.06952* (2016).

[28] LIU, C., HARRIS, A., MAAS, M., HICKS, M. W., TIWARI, M., AND SHI, E. Ghostrider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015* (2015), pp. 87–101.

[29] LIU, C., WANG, X. S., NAYAK, K., HUANG, Y., AND SHI, E. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 359–376.

[30] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: practical oblivious computation in a secure processor. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 311–324.

[31] NAVEED, M., KAMARA, S., AND WRIGHT, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 644–655.

[32] NAYAK, K., FLETCHER, C. W., REN, L., CHANDRAN, N., LOKAM, S., SHI, E., AND GOYAL, V. Hop: Hardware makes obfuscation practical. In *NDSS*.

[33] NIKOLAENKO, V., WEINSBERG, U., IOANNIDIS, S., JOYE, M., BONEH, D., AND TAFT, N. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 334–348.

[34] PODDAR, R., BOELTER, T., AND POPA, R. A. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive 2016* (2016), 591.

[35] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: processing queries on an encrypted database. *Commun. ACM 55*, 9 (2012), 103–111.

[36] POPA, R. A., STARK, E., VALDEZ, S., HELFER, J., ZELDOVICH, N., AND BALAKRISHNAN, H. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), pp. 157–172.

[37] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), pp. 431–446.

[38] ROCHE, D. S., AVIV, A. J., AND CHOI, S. G. A practical oblivious map data structure with secure deletion and history independence. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016* (2016), pp. 178–197.

[39] SASY, S., GORBUNOV, S., AND FLETCHER, C. W. Zerotrace : Oblivious memory primitives from intel SGX. *IACR Cryptology ePrint Archive 2017* (2017), 549.

[40] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 38–54.

[41] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware guard extension: Using SGX to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings* (2017), pp. 3–24.

[42] SEO, J., LEE, B., KIM, S., SHIH, M.-W., SHIN, I., HAN, D., AND KIM, T. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*.

[43] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*.

[44] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016* (2016), pp. 317–328.

[45] SHINDE, S., TIEN, D. L., TOPLE, S., AND SAXEENA, P. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*.

[46] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C. W., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 299–310.

[47] THIELMAN, S. Yahoo hack: 1bn accounts compromised by biggest data breach in history, 2016. `https://www.theguardian.com/technology/2016/dec/14/yahoo-hack-security-of-one-billion-accounts-breached`.

[48] WANG, X. S., NAYAK, K., LIU, C., CHAN, T. H., SHI, E., STEFANOV, E., AND HUANG, Y. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014* (2014), pp. 215–226.

[49] WEICHBRODT, N., KURMUS, A., PIETZUCH, P. R., AND KAPITZA, R. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I* (2016), pp. 440–457.

[50] WU, D. J., ZIMMERMAN, J., PLANUL, J., AND MITCHELL, J. C. Privacy-preserving shortest path computation. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* (2016).

[51] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 640–656.

[52] ZAHUR, S., WANG, X. S., RAYKOVA, M., GASCÓN, A., DOERNER, J., EVANS, D., AND KATZ, J. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016* (2016), pp. 218–234.

[53] ZHANG, Y., KATZ, J., AND PAPAMANTHOU, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 707–720.

[54] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), pp. 283–298.