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 a 9–500× speedup over naive ORAM. On analytics workloads, ObliDB ranges from competitive to 20× faster than systems designed *only* for analytics, such as Opaque. Moreover, ObliDB also supports point queries, insertions, deletions, and updates, 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 [**?**]). Researchers have proposed approaches including property-preserving encryption [**?**, **?**, **?**], secure hardware [**?**, **?**], and algorithms to execute specific computations on encrypted data [**?**, **?**], 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 [12]. 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 the the OS when needed. They therefore give a high level of protection from an attacker with no physical access to the machine, but full access to the OS. 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 [**?**]. Some recent systems, such as Opaque [**?**] and Cipherbase [**?**], 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. The key idea in ObliDB is to use techniques from Oblivious RAM (ORAM) [**?**] 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 [**?**] that changes every memory access to use ORAM) results in prohibitively high overheads. Instead, ObliDB carefully co-designs 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. Specifically, ObliDB supports two data representations—linear tables that must always be scanned, and oblivious B-trees that allow efficient indexed access. ObliDB also 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 [**?**, **?**], as well as efficient low-cardinality operations, such as point lookups, insertions, deletions and updates.

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 9-500×. For analytics workloads, we compare ObliDB to Opaque's oblivi-

[1] ObliDB can also be extended to pad intermediate or final results similar to Opaque's pad mode if desired.

ous mode [43] on the analytics queries used in the Opaque paper**?** and find that ObliDB is competitive with Opaque on most queries, but can also outperform Opaque by 20× on queries that can leverage indexes. ObliDB also comes within 2.1× of Spark SQL [3], which provides no security or privacy guarantees. For index workloads, ObliDB is competitive with the recent encrypted range search scheme of Demertzis et al [14] that does not hide access patterns. 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 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 a malicious operating system (OS) with power to examine and modify untrusted memory and any communication between the processor and memory. Moreover, the OS can observe access patterns to trusted memory and maliciously schedule processes or interrupt the execution of an enclave. We note that a malicious OS 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 SGX platform in that the enclave hides the contents of protected memory pages from a malicious OS, and we do not directly handle side-channels known to affect SGX hardware such as page fault timing attacks [40] and branch shadowing [24]. General solutions exist to protect against such side channels and are compatible with ObliDB (see Section 8 for an overview).

Furthermore, we also assume a secure channel exists through which an outside user can send messages to the enclave.

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 [43]. ObliDB additionally has 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. Our goals deal only with the security of data within individual tables.

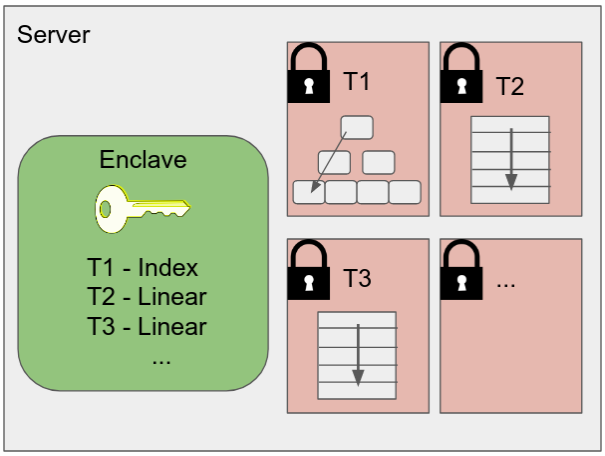


Figure 1: ObliDB provides an interface to a secure enclave with control over encrypted tables stored in untrusted memory. It stores tables either as an oblivious index or a linear scan data structure to ensure data-oblivious queries.

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.

We remark 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 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.

2.3 ObliDB Architecture Overview

ObliDB consists of a trusted code base inside an SGX enclave that provides an interface for users to create, modify, and query tables. ObliDB supports tables both with

and without indexes, called Indexed and Linear tables, respectively. 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 rely on accessing every block of the underlying data structure to ensure obliviousness. This overview of ObliDB's architecture is summarized in Figure 1.

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. Additionally, we include several different algorithms for the `SELECT` and `GROUP BY` operators, each of which performs better for a different output table size. Our `SELECT` implementation begins by scanning the table being queried to determine which algorithm to use and then executing the appropriate choice for the expected output size.

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. For more information on work using these primitives, particularly applications, attacks, and defenses for SGX, see Section 8.

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, 12]. SGX handles the process of entering and exiting an enclave and hiding the activity of the enclave while non-enclave code runs, albeit imperfectly [24]. 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 [21] 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 [23]. 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 [38])**.** *Let* $\overrightarrow{y} := ((\boldsymbol{op}_M, \boldsymbol{a}_M, \boldsymbol{data}_M), \ldots, (\boldsymbol{op}_1, \boldsymbol{a}_1, \boldsymbol{data}_1))$ *denote a data request sequence of length M, where each* $\boldsymbol{op}_i$ *denotes a* $\boldsymbol{read}(\boldsymbol{a}_i)$ *or a* $\boldsymbol{write}(\boldsymbol{a}_i, \boldsymbol{data})$ *operation. Specifically,* $\boldsymbol{a}_i$ *denotes the identifier of the block being read or written, and* $\boldsymbol{data}_i$ *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 ORAM. 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 pattern data.

Although other, older schemes have recently received attention due their practical efficiency in certain practical parameter settings [41], the most efficient ORAM scheme known is the Path ORAM [38]. 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 Oblivious Data Structures

ObliDB stores data at rest in two types of tables: Linear and Indexed. This section discusses each type of table 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 as well as multiple Indexed structures 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 kinds of tables, 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.

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.

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 a nonrecursive Path ORAM that holds a B+ tree where the actual data of the table resides. 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 [41].

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. We address considerations in ensuring obliviousness for each operator over Indexed tables in Section 5.

Data Integrity. Although encryption and oblivious data structures/algorithms ensure the privacy of data in ObliDB, additional protections make certain that a malicious OS 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 by the OS 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 a malicious OS 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: the first unused block encountered during the linear scan of the table will have the contents of the inserted row written to it instead of a dummy write.
- Deletion: any row matching the deletion criteria will be marked as unused and overwritten with fake data. Deletions and updates support the same kinds of conditions as selection, 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 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.

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 maintain 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 [6], 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 [16]. 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 structural, 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 structural leakage, can be hidden by padding, but ObliDB does not do this.

There are a few other differences between the behavior of ObliDB on Linear and Indexed tables that largely result from design and implementation decisions. Every query in ObliDB results in the generation of a Linear scan table with the response, so responses to queries on Indexed tables still appear in Linear tables. 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. The Large strategy for selection is not applicable for Indexed tables because the strategy of copying the whole table is not as applicable where a query is aimed at a small fraction of the table and the data are not stored in consecutive blocks but across multiple nested tree-based data structures.

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 [16]. 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.

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 [5], the latter of which was heavily edited in order to support duplicate labels and the dynamic memory abstraction we built on top of ORAM. We will make our implementation of ObliDB open source and publicly available online.

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 management system 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.

We use the last several bits of the SHA256 hash function whenever we require a hash function and key it by fixing the first byte of the hash input before concatenating the hash input. We use 128 bit AES with Galois Counter Mode for authenticated encryption. We only implement an inner join functionality for linear tables and a left join functionality for indexes, but other forms of join can be implemented without any additional technical or security-related obstacles. Finally, deletions in Indexed tables are designed to find one row matching the deletion criteria

Table Name	Rows	Notes
CFPB	107,000	Customer complaints to the US Consumer Financial Protection Bureau [17].
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 2: Tables with real data used in our evaluation and comparisons.

to remove, whereas deletions for Linear tables delete all rows matching the deletion criteria since performance for deleting one row or deleting all matching rows does not differ in the Linear table regime.

Many parameters in ObliDB contribute to optimizing performance for different settings. Most importantly, we set the bucket size of the ORAM to 4 and used a binary tree for the tree structure of the PATH ORAM. We also set the maximum branching factor of the B+ tree to 20. We set the number of rows that can fit in the enclave for the "Small" selection strategy at 5,000 and the maximum number of groups in aggregates and rows in `JOIN` queries to 350,000 each. The large, fixed maximum size for aggregates and joins can be removed at some performance cost.

7 Evaluation

In this section we evaluate ObliDB on tables of up to 1.4 million rows and measure its performance against a baseline oblivious database implementation as well as existing private database systems built with and without secure enclaves. A summary of real-world data sets used in our experiments appears in Figure 2. 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 an Intel Nuc box with an 1.9 GHz Intel Core i5-6260U Dual-Core processor and 32GB of RAM running Ubuntu 16.04.2 and the SGX Linux SDK version 1.8 [1].

We conclude that ObliDB dramatically outperforms a baseline implementation and can leverage its indexes to achieve order of magnitude performance improvements over previous private database systems regardless of whether or not they use secure enclaves in their design.

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. Figure 3 compares ObliDB to a baseline oblivious database implementation where ORAM accesses naively replace memory accesses. ObliDB achieves up to 29× speedup for `SELECT` queries and over 284× speedup for aggregates. `SELECT` queries over Linear tables enjoy much larger speedup 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. Likewise, the relatively smaller – although by no means inconsequential – gains on insertion and deletion in indexes arise from the fact that most of the work of designing an oblivious index goes into achieving the obliviousness property in the first place.

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 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 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 4 compares ObliDB with Opaque's oblivious mode [43] and Spark SQL [3] 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

Data Set	Query	ObliDB	Baseline	Speedup
	Linear Selection			
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2013-05-14`	1.40s	40.78s	29.1×
RANKINGS	`SELECT pageURL, pageRank FROM RANKINGS WHERE pageRank > 1000`	3.03s	57.51s	19.0×
	Index Selection			
CFPB	`SELECT * FROM CFPB WHERE Date_Received=2013-05-14`	0.63s	0.80s	1.3×
RANKINGS	`SELECT pageURL, pageRank FROM RANKINGS WHERE pageRank > 1000`	0.11s	0.14s	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")`	0.20s	0.99s	5.0×
CFPB	`DELETE FROM CFPB WHERE Bank="Bank of America" LIMIT 1`	0.43s	0.60s	1.4×
	Aggregates and Joins			
CFPB	`SELECT COUNT(*) FROM CFPB WHERE (Product="Credit card" OR Product="Mortgage") AND Timely_Response="No" GROUP BY Bank`	0.69s	128.52s	187.1×
USERVISITS	`SELECT SUBSTR(sourceIP, 1, 8), SUM(adRevenue) FROM USERVISITS GROUP BY SUBSTR(sourceIP, 1, 8)`	3.52s	>1000s	>284×
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`	15.20s	>1000s	>65.8×

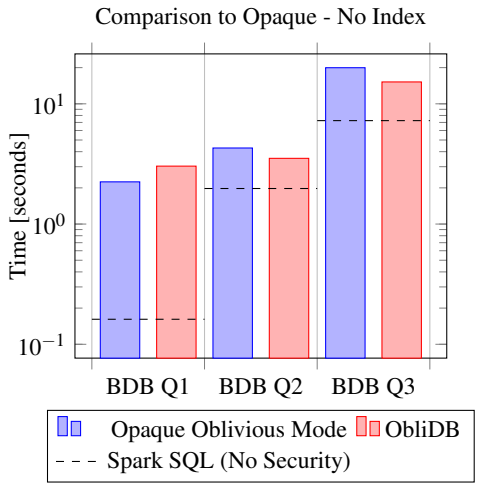
Figure 3: Comparison of ObliDB and a baseline where a naive oblivious database implementation directly ports non-oblivious algorithms to their oblivious counterparts via ORAM. ObliDB outperforms the baseline on all queries.

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 1.2× speedup on query 2 to 20× 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. 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 1.8× slower on query 2 and 2.1× slower on query 3, putting ObliDB safely in the realm of tools whose performance is practical 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 [8] in Figure 5. 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 much more powerful machine than ours: an Intel Core i7 4790K 4.00GHz CPU with 8 logical cores,



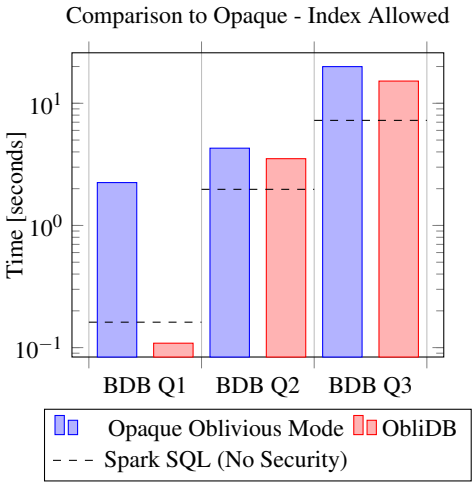


Figure 4: ObliDB outperforms Opaque Oblivious [43] by 1.2-20× and never runs more than 2.1× slower than Spark SQL [3] on the Q1-Q3 of the Big Data Benchmark [2]. Even without use of an index, ObliDB performs comparably to Opaque Oblivious.

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 4.7-19× 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 would remain constant.

7.4 Padding Mode Overhead

In Figure 6, we report on the overhead of using padding mode in ObliDB to hide not only access patterns to data but also all information regarding the sizes of tables,

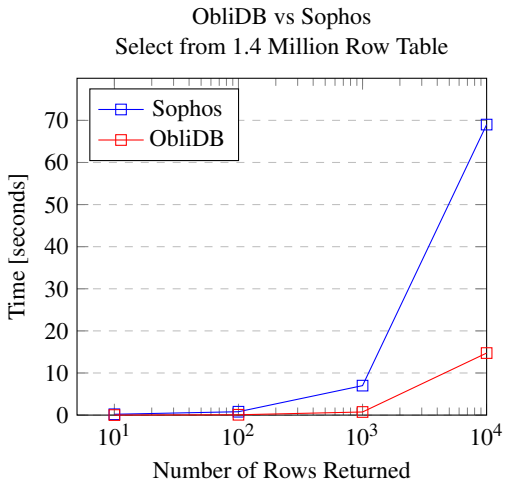


Figure 5: Comparison of ObliDB to Sophos SSE scheme [8] on 1.4 million rows of data. ObliDB outperforms Sophos by 19× when selecting 10 rows and by 4.7× when selecting 10,000 rows. Unlike ObliDB, Sophos does not use SGX but leaks access patterns to data.

Query Type	No Padding	Padding	Slowdown
Aggregate	0.72s	7.31s	10.2×
Select (Linear)	1.40s	8.12s	5.8×
Select (Index)	0.63s	2.35s	3.7×
Insert (Index)	0.20s	0.25s	1.3×
Delete (Index)	0.43s	0.49s	1.1×

Figure 6: Slowdown of ObliDB in padding mode for queries in the CFPB table of 107,000 rows padded to 500,000 rows.

whether they be at rest in the database, intermediate tables constructed in handling a query, or the results returned from a query. The table shows query results for the 107,000 row CFPB table padded up to 500,000 rows.

Opaque [43] 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 5×.

7.5 Microbenchmarks

Comparison of Linear and Indexed tables. 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 bet-

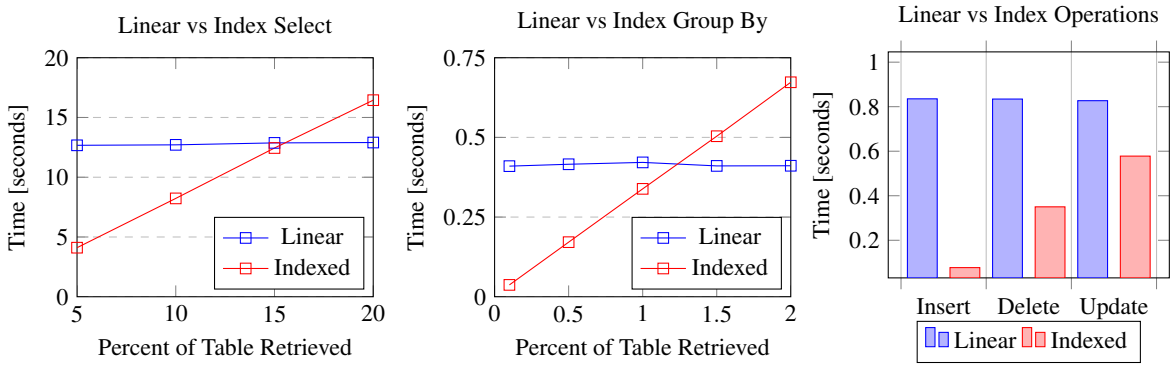


Figure 7: Comparison of Linear and Index versions of operators over 100,000 rows of fabricated data. Linear scans do better when most of the data needs to be accessed, but Indexed structures perform far better for small queries. Operations involving modification of the database are far faster for indexed structures.

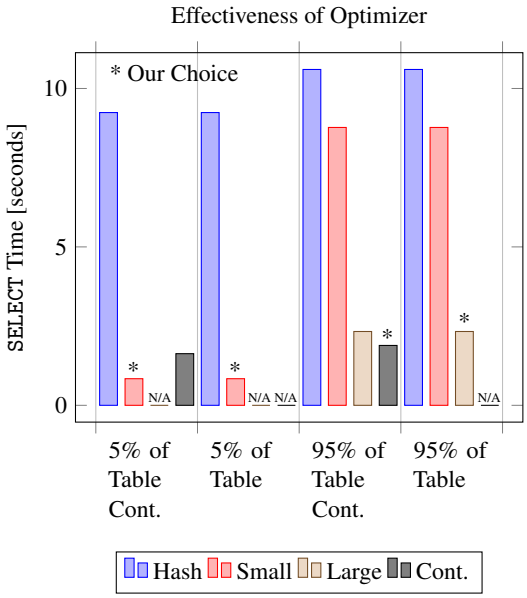


Figure 8: Our optimizer picks the best algorithm for handling `SELECT` queries based on a preliminary scan that determines whether the data to be returned is small, large, or consists of a continuous set of rows in the table.

ter 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 [31] that indicate linear scans always perform better than ORAM for oblivious memory access. In general, Indexed `INSERT`, `DELETE`, and `UPDATE` queries significantly outperform Linear tables. Indexed insertions complete faster than deletions and updates because insertions only need to find a place in the B+ tree, but deletions and updates need to scan forward from that point to find rows that need to be deleted/updated.

Effectiveness of Optimizer. Figure 8 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 3) performs 58× better than the low-cardinality algorithm would on the same query.

8 Related Work

ObliDB is related to a number of prior works involving cryptographically-protected databases and applications of trusted hardware.

Cryptographically-protected database search. A testament to the importance of the problem of search over encrypted data in databases lies in the extensive prior work on the subject, summarized and systematized by Fuller et al [20]. Perhaps the most widely known work in this area is CryptDB [30], which implements a tradeoff between security and performance by encrypting each field in a table according to the type of operation expected to be used on the data in that field. Arx [29], 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 [14]. These schemes rely on searchable symmetric encryption (SSE) as a primitive, a recent example of which is Sophos [8], which boasts forward

security – queries authorized in the past do not leak any new information when additional documents are added to an existing database. A very recent follow-up work to Sophos introduces a notion of backward-security and provides improved constructions [9].

The diversity of security goals and varied use cases for which cryptographically protected databases have been designed have led to a plague of attacks which show that real-world applications of schemes proven secure in theoretical models can in fact leak far more data than would be expected from an initial examination of a system's security properties. Initiated by Islam et al [23] and continuing with improved results such as those of Naveed et al [27] and Cash et al [11] to name only a few, such attacks show that inference from known context of the data used, additional correlated public data, or even just the leakage inherent in a scheme itself, can be used to attack various schemes in ways not anticipated in their original security models. Zhang et al [42] show that even schemes with very little leakage are susceptible to attack. In hiding even the access patterns to data in our solution, we hope to minimize the extent to which ObliDB is vulnerable to such techniques.

Trusted hardware. Trusted hardware can be used to achieve security properties that are difficult, impractical, or potentially impossible with traditional cryptographic assumptions. For example, a number of hardware or hardware/software based solutions exist with the explicit goal of rendering programs' memory traces oblivious [13, 25, 26]. Intel SGX, on which we will focus, has been used to implement practical functional encryption [18] and obfuscation [28], both functionalities which can currently only be constructed using heavy cryptographic machinery.

In recent years, a number of generic tools have been designed to provide legacy applications the heightened security available from SGX. Haven [7] shields execution of legacy programs from a malicious OS. Panoply and SCONE [4, 37] provide SGX-protected Linux OS and container abstractions. In the distributed setting, Ryoan [22] is a sandbox for computation on secret data.

In addition to general tools, applications to securely conduct data analytics or handle data in the cloud represent a compelling practical use case for SGX hardware. In this vein, many works implement variations of existing tools and services rendered secure via SGX. M2R [15] and VC3 [33] provide MapReduce and cloud data analytics functionalities, respectively, and Opaque [43] provides secure support for Spark SQL. SecureKeeper [10] uses SGX to build a confidential version of Apache's ZooKeeper (`zookeeper.apache.org`). More fundamental primitives for databases and oblivious computation in general are provided by HardIDX [19], a database index in SGX, and ZeroTrace [32] which provides oblivious memory primitives based on ORAM as well as an analysis of parameter optimizations for using an ORAM controller in SGX for data storage. None of the solutions above, with the exception of ZeroTrace, use ORAM to hide memory access patterns. Instead, they either make use of memory-oblivious algorithms suited to the tasks they undertake or remain vulnerable to side-channel attacks targeting memory access patterns.

Trusted hardware assumption like those underlying SGX fundamentally differ from traditional mathematical assumptions in that, whereas the validity of a mathematical assumption cannot be challenged by the vicissitudes of succeeding implementations, attacks, and side-channels, the legitimacy of a hardware assumption relies directly on the ability of a piece of manufactured hardware to repel practical attacks. As such, the SGX literature includes a number of works that aim to reveal practical side channels in the implementation of SGX and develop techniques to obviate the risks presented by each known family of attacks. Xu et al [40] use page faults and other "controlled channel" side channels to extract images and text documents from protected memory, and Lee et al [24] use the fact that SGX does not clear branch history when leaving an enclave to infer details of branches taken in protected code. In the multithreaded regime, Weichbrodt et al [39] compromise security by leveraging synchronization bugs. Defenses against such attacks include the work of Shinde et al [36] and Raccoon [31] which close side channels by making the memory trace of a program oblivious or obfuscated. SGX-Shield [34] enables address space layout randomization for SGX, and, finally, T-SGX [35] protects against side-channel attacks by using another set of hardware features, Transactional Synchronization Extensions (TSX) to close side channels that could otherwise be exploited by a malicious OS. ObliDB can be generically combined with any of these solutions to provide higher levels of confidence in the security of the enclave.

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 additionally offers a padding mode to hide even table sizes). We have shown that ObliDB handles practical data sets with performance surpassing prior work with similar security properties as well as other existing solutions with more permissive leakage functions. 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] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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).

[24] 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).

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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*.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.