OblivDB: A Protected Database with Near-Minimal Leakage Using SGX

Abstract

Applications meant to handle private data using Intel SGX often leak private information through access patterns to data stored outside the SGX enclave. Access patterns can be naively hidden by using an ORAM and directly porting various SQL operators to their oblivious counterparts, but this approach suffers from unacceptable performance loss. The challenge of oblivious SGX system design, then, lies in developing oblivious algorithms, with or without ORAM, that retain performance near to that of their non-oblivious counterparts.

We present OblivDB, a database built on Intel's SGX hardware that provides minimal leakage of cryptographically protected data. OblivDB achieves superior performance to prior work while leaking only the structure of queries and the protected data returned from them. In particular, OblivDB hides access patterns to data. OblivDB supports a broad range of SQL queries including groupings and joins and makes use of both ORAM-based oblivious indexes and linear search-based data structures. It also provides a range of oblivious algorithms to execute operators with different selectivity.

We evaluate OblivDB on several real-world data sets and queries, finding that OblivDB outperforms a baseline implementation by 9-500× and Opaque's oblivious mode [45] by 1.2-20×, coming within 2.1× of the performance of Spark SQL [3], a system with no data privacy guarantees.

1 Introduction

The advent of cloud computing has ushered in hopes of a future where data owners can outsource their databases while retaining data security. In recent years, a smorgasbord of solutions to the problems of cryptographically-protected databases and search over encrypted data has explored a large space of tradeoffs between performance and leakage of private queries and data [21], but performant systems which provide the highest levels of security – that of hiding even access patterns to protected data – remain out of reach. Meanwhile, increasing interest in trusted hardware solutions, lent impetus by the appearance of Intel's SGX [11] has allowed for dramatic speedups in tasks previously requiring heavy and slow cryptographic operations [19,29]. While SGX alone partially solves the problem of protected databases [20], preventing leakage of data access patterns requires more work. Several prior works [13,21,30] mention the possibility of generically using Oblivious RAM (ORAM) on top of SGX to hide these access patterns but point out that, unfortunately, simply running a generic database application over ORAM adds significant slowdowns.

We present OblivDB, an SGX-based system that specializes data structures and query execution for the database case and leaks only structural information about queries, results, and stored data – the leakage that can be hidden only by padding. OblivDB provides both indexed and unindexed tables and supports many SQL queries including the aggregates `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`, as well as `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `GROUP BY` and `JOIN` queries, a broader set of queries than those supported by most systems offering search over encrypted data that hide data access patterns [21]. OblivDB also supplies a number of algorithms that optimize operator performance based on the size of data to be returned from a query, thereby providing better performance without any additional leakage.

We implement a prototype of OblivDB and report on its performance, testing it with microbenchmarks on synthetic data of up to 500,000 rows and real queries on multiple real-world data sets of various sizes: domestic US flight data [16], consumer complaints to the consumer financial protection bureau [17], the NASDAQ stock exchange [18], and the tables and queries of the Big Data Benchmark [2]. We compare OblivDB to a baseline implementation where a database index is generically modified to provide data access obliviousness via ORAM and show that OblivDB outperforms the baseline by 4.5× and 1.2× on insertions and deletions, respectively, and by 9-499× on realistic queries to data. We also compare OblivDB to prior work and find that OblivDB performs comparably to the range search scheme of Demertzis et al [13] which does not hide access patterns and ranges from 1.2-20× faster than Opaque's Oblivious mode [45], an SGX-based data analytics platform that does hide access patterns, suggesting that OblivDB and Opaque are fundamentally well-suited to different use cases. Moreover, OblivDB comes within at least 2.1× of the performance of Spark SQL [3], which provides no security or privacy guarantees, on all implemented Big Data Benchmark queries. Finally, we show that the choices of oblivious data structures and algorithms available in OblivDB allow for meaningful optimizations in different data and query settings.

The rest of this paper is organized as follows: Section 2 gives an overview of OblivDB and the security model in which we operate. Section 3 gives background on relevant tools used in OblivDB, and Sections 4 and 5 detail OblivDB'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 OblivDB, the threat model for which it is designed, and the security properties it achieves.

2.1 Threat Model

The threat model we assume for OblivDB is 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 maliciously schedule processes or interrupt the execution of an enclave. We note that it is always possible for a malicious OS to launch an indefinite denial of service attack against an enclave, but such an attack does not compromise security and is outside the scope of the security of SGX for our purposes.

We assume that the SGX platform and its protected memory pages are secure and do not directly handle side-channels known to affect SGX hardware such as page fault timing attacks [42] and branch shadowing [25]. General solutions exist to protect against such side channels and are compatible with OblivDB (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 (this is fairly straightforward with SGX), but we did not implement this for our tests, as it is not directly related to the functionality provided by OblivDB.

2.2 Security Goals

To carefully describe the level of security provided by OblivDB, we follow the convention [21] of describing security in terms of the level of leakage in our scheme. Queries in OblivDB reach the highest standard for leakage[1]: to leak only "structural" information about data, queries, and responses, i.e., to only reveal information that can be hidden by padding. This includes, for example, the sizes of tables in the database, the sizes of queries, and the sizes of responses to queries[2]. Hiding this leakage can only be accomplished with padding at a necessarily high performance cost. OblivDB can, however, be configured to provide a partial form of padding where sizes of all tables are padded to the next power of two, balancing a degree of padding with minimal performance costs (identical to the cost of having tables whose size is the next power of two). In general, OblivDB only leaks the structure of queries made on data, the structure of the data returned, and the query plan used to service a query. Structural information regarding intermediate tables created during query execution also leaks. Further details regarding how these leakage properties are achieved can be seen in Sections 4 and 5, where the leakage of each operator is explained. Data at rest outside the enclave is encrypted with a semantically secure encryption scheme and leaks only the size of the encrypted data.

We additionally guarantee authenticity of data and queries, meaning that any tampering with data executed by the malicious OS will be caught and reported. 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. These protections are discussed in Section 4.

Another concern with regards to security is that, however secure the properties of the database management system, the way it is used by 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 is a direct, if unexpected, consequence of structural leakage,

[1] Some query plans can reveal additional information as a security/performance tradeoff, but the full functionality of OblivDB can still be achieved without these shortcuts. Each such case is discussed in Section 5.

[2] 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.

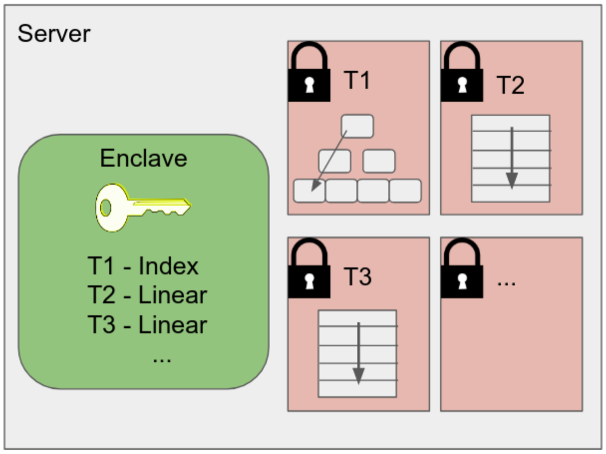

Figure 1: OblivDB provides an interface to a secure enclave with control over encrypted tables stored in untrusted memory. Tables can be stored either as an oblivious index or a linear scan data structure to ensure data-oblivious queries.

and it falls on application developers to consider performance goals against the ramifications of such leakage in their design process.

2.3 OblivDB Architecture Overview

OblivDB consists of a trusted code base inside an SGX enclave that provides an interface for users to create, modify, and query tables. OblivDB supports tables both with and without indexes, called Indexed and Linear tables, respectively. These tables are stored, encrypted, in unprotected memory and are obliviously accessed 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 OblivDB's architecture is summarized in Figure 1.

OblivDB 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, several different algorithms are included for the `SELECT` operator, 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 OblivDB, 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, 11]. SGX handles the process of entering and exiting an enclave and hiding the activity of the enclave when non-enclave code is being run, albeit imperfectly [25]. 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 can be compared with the expected value of the hash and rejected if there is any evidence of a corrupted or altered program.

The feature of SGX with which we are most concerned is the protection of memory. SGX provides the developer with approximately 90MB of Enclave Page Cache (EPC), a memory region that is hidden from the OS and cleared whenever execution enters or exits an enclave. This memory can be used to 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 [22] 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 [24]. ORAM goes further and shuffles the locations of blocks in memory so repeated accesses 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 [39])**.** *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, **a**$_i$ denotes the identifier of the block being read or written, and **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 for two primary reasons. First, 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. Second, the definition of security for ORAM does not address side channels, so implementations using ORAMs must ensure that a program's branching behavior does not leak private information without relying on ORAM.

Although other, older schemes have recently received attention due their practical efficiency in certain practical parameter settings [43], the most efficient ORAM scheme known is the Path ORAM [39]. 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

OblivDB 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.

Tables in OblivDB are created with an initial maximum capacity that can be increased later by copying to a new, larger table. Although our prototype does not implement this feature, it is possible for one table to be represented by a Linear structure as well as multiple Indexed structures. Since tables are stored in unprotected memory, every block of each data structure is independently encrypted and MACed with a symmetric key generated inside the enclave. For both kinds of tables, each row of a table is stored in one block of the corresponding data structure, and the first byte of each block is reserved as a flag to indicate whether that block contains a row or is empty. The decision to store one row per block requires that the block size for each data structure be set close to the size of a row. This is not a necessity of our design but a choice of convenience, and the number of rows per block represents a parameter that can be adjusted in a search for optimal performance.

A Linear type table simply stores rows in a series of adjacent blocks with no additional mechanism to ensure obliviousness of memory accesses. This is 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.

In contrast to the simplicity of Linear 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 OblivDB can handle realistic data sets without any need for a recursive ORAM. That said, there is no reason OblivDB cannot be modified to make use of recursive ORAM at a modest performance penalty for Indexed tables. Moreover, the OblivDB 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 [43].

Although the security properties of ORAM guarantee that two access transcripts of the same length will be indistinguishable from each other, it is important in designing oblivious algorithms for operators over Indexed tables to make sure that the total number of accesses or the timing gaps between accesses do not leak any additional private data. For example, the property of the B+ tree that all data is stored 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 did not exhibit such a property would compromise the obliviousness of our operators on Indexed tables. These concerns are addressed for each operator in Section 5.

Data Integrity. Although encryption and oblivious data structures/algorithms ensure the privacy of data in OblivDB, additional protections are needed in order to 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. OblivDB protects against such attacks and reports any attempt by the OS to tamper with data.

Every block of data stored outside the enclave is MACed and encrypted, 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," and revision numbers are also stored inside the enclave. Any attempt to duplicate, shuffle, or remove rows within a data structure will be caught when one of OblivDB's operators discovers that the row number of data it has requested does not exist or does not correspond to that which it has received. Rollbacks of system state will be 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 OblivDB.

5 Oblivious Operators

In this section we describe the various oblivious operator algorithms used in OblivDB. OblivDB 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 structural information about the response to a query, OblivDB can use a different algorithm 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 O, and the table being selected from will be referred to as T. The number of rows in O is represented by o, the number of rows in T is N. o' and N' represent the number of blocks in the data structures holding O 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 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:

- *Continuous*: Should the rows selected form one continuous section of the data stored in the table, OblivDB employs a special strategy which requires only one additional pass over the table. First, table O is created with o rows. Then, for the ith row in

T, if that row should be in the output, it is written to row *i mod o* of O. If not, a dummy write takes place. Since the rows of O are one continuous segment of T, this procedure results in exactly the selected rows appearing in O.

In addition to the sizes of T and O, the fact that this algorithm is chosen 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. No other information about access patterns leaks, 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 O.

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

  This algorithm leaks only the sizes of T and O 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 O, which is revealed anyway.
- *Large*: If O contains almost every row of T, we create O as a copy of T and then make one pass over O 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 O 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 O because the data structure size is padded up to the size of T.
- *Hash*: In the case that none of the preceding special-case algorithms apply, OblivDB uses the following generalization of the continuous strategy. The main idea is that 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 solution 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 positin in O, where h is a hash function (we used the last several bits of the SHA256 hash function).

  The algorithm as stated above does not exactly represent how OblivDB works because a few changes are needed in order to ensure, first, that hash collisions are handled to ensure correctness, and, second, that obliviousness is maintained in handling any collisions. In order to maintain obliviousness, every real or dummy write to O must involve the same number of accesses to memory. This means that if any write resolves in a collision that must be resolved, every write must make as many memory accesses as in the case of a collision. Following the guidance of Azar et al [6], we hash each row number using two different hash functions (prepending 0 or 1 to the input of the SHA256 hash) and have a fixed-depth list of 5 slots for each position in O. This means that for each block in T, there will be 10 accesses to O, 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 are selected by the query. Since the hash is taken over the index of the row in the data structure and not over the actual contents of a row, there is no possibility that any information about the data itself can be leaked by observing the patterns of accesses as rows are written to O. As such, the only leaked information is still the sizes of T and O.
- *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 each row is examined, an ORAM operation takes place. If the examined row is to be included in the output, the operation is a write. If not, the operation is a dummy op (read an arbitrary block). After completing the scan of the input table, the ORAM is copied to a Linear table and returned.

Aggregates & Group By. Computing aggregates can be done far faster than selection and only requires one oblivious pass over a Linear table. An aggregate over a whole table or some selected subset of a table requires only one pass over the whole table where the aggregate is calculated cumulatively based on the data in each row. Since the memory access pattern of this operation will always be sequential reads of each block in the data structure, nothing is leaked from this operation beyond the size of T.

Groupings are handled similarly to aggregates without groupings, except that an array is kept inside the enclave that keeps track of the aggregate for each group. 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, a linear scan is made over all the known group values in order to check for a match. If no match is found, a new group is created.
- *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.

  OblivDB, as implemented, supports only a large fixed total number of groups (up to as many as can fit in an enclave), but it can be extended to support arbitrarily large numbers of groups by storing the hash table in an ORAM outside the enclave instead of in the enclave's trusted memory.

Join. The Join functionality for Linear tables is implemented as a variant of the standard hash join algorithm [15]. Although we only implement inner joins for Linear tables, other forms of join can be implemented without any additional technical or security-related obstacles. We will 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 the end of T_1 is reached. 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 the maximum possible size by dummy rows. 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 are largely similar to those for Linear tables, except all the operations are 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 OblivDB does not do this.

There are a few other differences between the behavior of OblivDB on Linear and Indexed tables that largely result from design and implementation decisions. Every query in OblivDB 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. Deletions in Indexed tables are designed to find one row matching the deletion criteria 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. The Large strategy for selection is not implemented 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. Finally, Joins are implemented differently for Indexed tables. Whereas we implement `INNER JOIN` functionality for Linear tables, Indexed tables only support `LEFT 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 [15]. 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

We implemented and evaluated OblivDB 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]. Our implementation includes the linear scan and oblivious B+ index from Section 4 as well as most of the oblivious operator algorithms described in Section 5, with the exception of those stated there to be unimplemented. Our final implementation consists of approximately 12,000 lines of code and builds upon the Remote Attestation sample code provided with the SDK 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 intend to make our implementation of OblivDB 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 implemented equivalents of malloc, free and the pointer dereference operator for our ORAM. Our memory management system simply consisted of an array of flags that would be set if the corresponding block was in use and unset if it was 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 OblivDB for the position map, ORAM stash, and other elements of system state recording the names, sizes, and types of existing tables.

There are many parameters in OblivDB that, set appropriately, could improve performance that we made no effort to optimize. 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, a number that we felt would allow for reasonably-sized rows without overflowing the memory available to the enclave.

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.

7 Evaluation

We extensively evaluate OblivDB to demonstrate that choices of data structures and algorithms offered by OblivDB allow for meaningful optimizations in handling queries without increasing leakage. We also execute real queries on multiple data sets:

- NASDAQ [18] consists of the last sale price and other related information for all approximately 3,200 stocks on the NASDAQ stock exchange.
- CFPB [17] contains individual complaints on financial products and services sent to the US consumer financial protection bureau and contains about 106,000 rows.
- FLIGHTS [16], a subset of the flight dataset used to evaluate Splinter [40], contains data on arrival/departure points, flight numbers, ticket prices, and delays for 250,000 US domestic flights.
- RANKINGS [2], a table from the Big Data Benchmark, contains 360,000 rows of URLs, PageRanks, and average visit durations.
- USERVISITS [2], another Big Data Benchmark table of 350,000 rows, stores server logs for many web pages.
- SYNTH contains synthesized data used for Microbenchmarks and varied in size from 100 to 500,000 rows during our experiments. We report results for 100,000 rows.

A summary of queries executed on real-world data is shown in Figure 3.

We compare OblivDB to a baseline oblivious database implementation that naively converts data structures and algorithms to their oblivious counterparts using ORAM, to Opaque's oblivious mode [45] run on a single node, providing security guarantees comparable to OblivDB, and Spark SQL running on one core [3], which provides

Data Set	Query	OblivDB	Baseline	Speedup
	Aggregates and Joins			
CFPB	`SELECT COUNT(*) FROM CFPB WHERE (Product="Credit card" OR Product="Mortgage") AND Timely_Response="No" GROUP BY Bank`	0.72s	>1000s	>1390×
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×
	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 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×
	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×

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

no security guarantees. OblivDB dramatically outperforms the baseline and also performs better than Opaque while coming, at worst, within just over a factor of 2 of the performance of Spark SQL.

7.1 Microbenchmarks: Effectiveness of Low-Leakage Optimization

By providing both Indexed and Linear tables and optimizing `SELECT` and `GROUP BY` queries based only on information from structureal leakage, OblivDB to makes meaningful performance improvements for diverse queries. To this end, Figure 4 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 [32] 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.

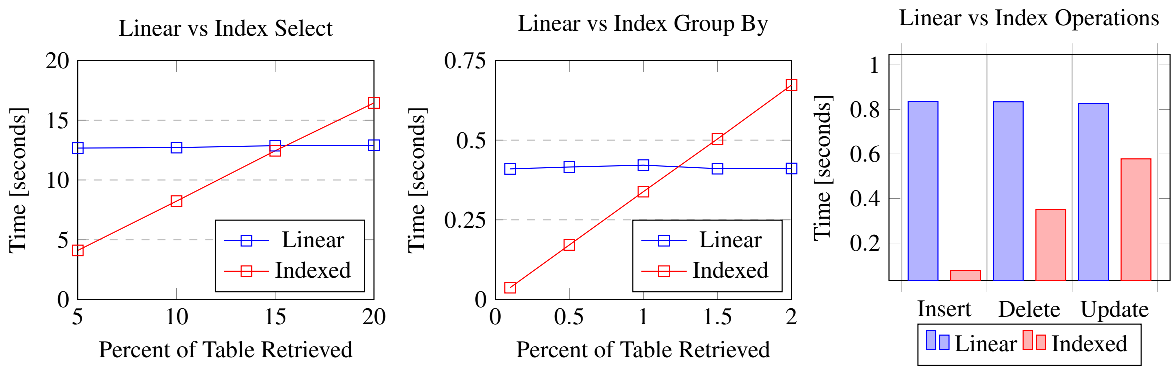

Figure 4: Comparison of Linear and Index versions of operators over 100,000 rows of fabricated data. Linear scans do better when most of the data needs to be accessed, but Indexed structures perform far better for small queries. Operations involving modification of the database are far faster for indexed structures.

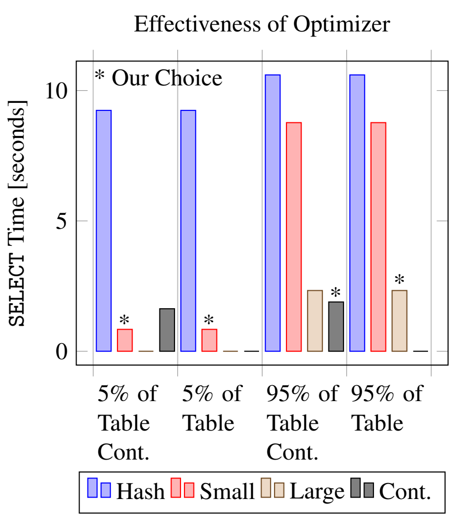

Figure 5: Our optimizer picks the best algorithm for handling `SELECT` queries based on a preliminary scan that determines whether the data to be returned is small, large, or consists of a continuous set of rows in the table.

Figure 5 demonstrates the effectiveness of OblivDB's choice of `SELECT` algorithms, comparing our various algorithms on queries that retrieve 5% and 95% percent of the SYNTH. Although the "Hash" algorithm performs the best asymptotically, the figure demonstrates that knowledge gleaned only from OblivDB'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, as demonstrated by the query on the USERVISITS table in Figure 3 – the baseline algorithm uses only the low-cardinality `GROUP BY` algorithm whereas OblivDB can use the more appropriate high-cardinality variant in this situation.

7.2 Comparison to Baseline/Prior Work

OblivDB's Indexed tables perform better than a baseline implementation (e.g. what could be achieved with a generic tool for converting legacy applications). The queries on the SYNTH table in Figure 3 show comparisons on `INSERT` and `DELETE` operations between OblivDB and a baseline solution where a database index is generically modified to provide obliviousness of data access patterns via an ORAM. Whereas OblivDB's Indexed tables include a number of implementation optimizations to minimize the number of ORAM reads and writes needed in any query, our generic baseline makes an ORAM access every time a non-oblivious database would. The `INSERT` query runs much faster than `DELETE` because `INSERT` only needs to find one place in the tree to insert a row, whereas `DELETE` needs to find a position in the leaves of the B+ tree and then scan forward until it finds a row to delete. OblivDB outperforms the baseline by 4.5× on insertions and 1.2× on deletions. As for queries on static data, Figure 3 also shows that OblivDB achieves an 8.6-498.6× speedup over the baseline on various real-world queries [2, 16–18].

Comparison to Prior Work. Figure 6 compares OblivDB with our baseline implementation, Opaque's oblivious mode [45], and Spark SQL [3] on the first three queries of the Big Data Benchmark [2] on tables of 360,000 and 350,000 rows. The fourth query of the benchmark is omitted as neither OblivDB nor Opaque support the external scripts needed for it. The baseline implementation makes use of the naive `SELECT` and low-cardinality `GROUP BY` algorithms described in Section 5. Opaque can be configured in either an "encryption" mode, which leaks access patterns but offers performance close to Spark SQL (which is why it is omitted

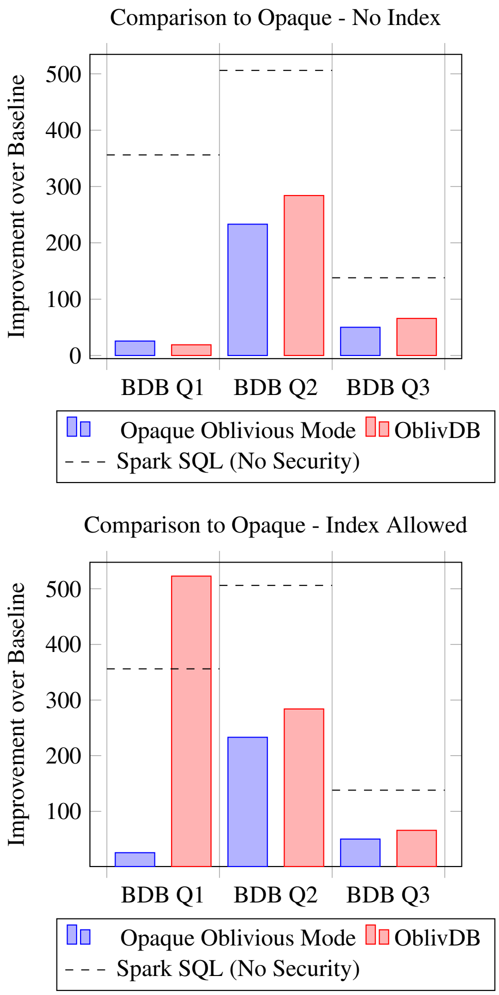

Figure 6: OblivDB outperforms Baseline and Opaque oblivious [45] by 66-523× and 1.2-20× respectively and never runs more than 2.1× slower than Spark SQL [3] on the Q1-Q3 of the Big Data Benchmark [2]. Even without use of an index, OblivDB performs comparably to Opaque Oblivious and outperforms a baseline by one or more orders of magnitude.

here) 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 comparable 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.

OblivDB 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 OblivDB compared to Opaque and Spark SQL on query 1 is due to OblivDB's oblivious index, which allows it to only examine a small portion of the table whereas Opaque and Spark SQL scan the entire table to satisfy the query. For queries 2 and 3, although OblivDB is slower than Spark SQL, it is only 1.8× slower on query 2 and 2.1× slower on query 3, putting OblivDB safely in the realm of tools whose performance is practical for real applications.

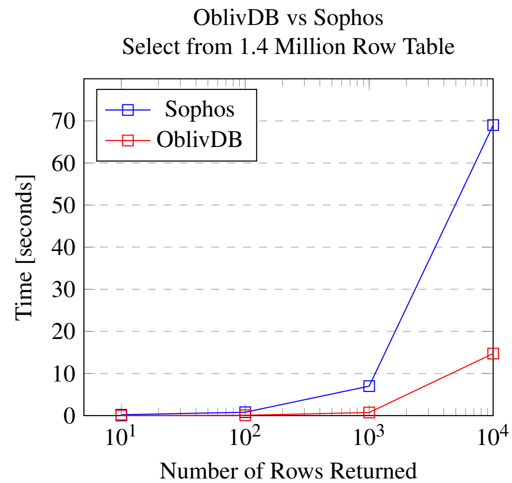

Figure 7: Comparison of OblivDB to Sophos SSE scheme [8] on 1.4 million rows of data. OblivDB outperforms Sophos by 19× when selecting 10 rows and by 4.7× when selecting 10,000 rows. Unlike OblivDB, Sophos does not use SGX but leaks access patterns to data.

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 8: Slowdown of OblivDB in padding mode for queries in the CFPB table of 107,000 rows padded to 500,000 rows.

8 Related Work

OblivDB 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 [21]. Perhaps the most widely known work in this area is CryptDB [31], 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 [30], 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 [13]. 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.

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 [24] and continuing with improved results such as those of Naveed et al [28] and Cash et al [10] 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 [44] 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 OblivDB 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 [12,26,27]. Intel SGX, on which we will focus, has been used to implement practical functional encryption [19] and obfuscation [29], 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, 38] provide SGX-protected Linux OS and container abstractions. In the distributed setting, Ryoan [23] 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 [14] and VC3 [34] provide MapReduce and cloud data analytics functionalities, respectively, and Opaque [45] provides secure support for Spark SQL. SecureKeeper [9] 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 [20], a database index in SGX, and ZeroTrace [33] 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 [42] use page faults and other "controlled channel" side channels to extract images and text documents from protected memory, and Lee et al [25] 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 [41] compromise security by leveraging synchronization bugs. Defenses against such attacks include the work of Shinde et al [37] and Raccoon [32] which close side channels by making the memory trace of a program oblivious or obfuscated. SGX-Shield [35] enables address space layout randomization for SGX, and, finally, T-SGX [36] 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. OblivDB 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 OblivDB, a cryptographically-protected database system based on Intel SGX which leaks only structural information about data and queries. We have shown that OblivDB handles practical data sets with performance surpassing prior work with similar security properties and comparable to other existing solutions with more permissive leakage functions. It is our hope that solutions like OblivDB 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. ∑οφος: 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] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] ENIGMA. Arrival data for non-stop domestic flights by major air carriers for 2012. `https://app.enigma.io/table/us.gov.dot.rita.trans-stats.on-time-performance.2012`.

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

[18] ENIGMA. Nasdaq stock exchange. `https://app.enigma.io/table/com.nasdaq.exchange.company-list.nasdaq`.

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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).

[25] 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).

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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*.

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

[37] 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.

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

[39] 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.

[40] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), pp. 299–313.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.