ObliDB: Oblivious Query Processing for Secure Databases

Saba Eskandarian
Stanford University
saba@cs.stanford.edu

Matei Zaharia
Stanford University/Databricks
matei@cs.stanford.edu

ABSTRACT

Hardware enclaves such as Intel SGX are a promising technology for improving the security of databases outsourced to the cloud. These enclaves provide an execution environment isolated from the hypervisor/OS, and encrypt data in RAM. However, for applications that use large amounts of memory, including most databases, enclaves do not protect against *access pattern* leaks, which let attackers gain a large amount of information about the data. Moreover, the naïve way to address this issue, using Oblivious RAM (ORAM) primitives from the security literature, adds substantial overhead.

A number of recent works explore trusted hardware enclaves as a path toward secure, access-pattern oblivious outsourcing of data storage and analysis. While these works efficiently solve specific subproblems (e.g. building secure indexes or running analytics queries that always scan entire tables), no prior work has supported oblivious query processing for general query workloads on a DBMS engine with multiple access methods. Moreover, applying these techniques individually does not guarantee that an end-to-end workload, such as a complex SQL query over multiple tables, will be oblivious. In this paper, we introduce ObliDB, an oblivious database engine design that is the first system to provide obliviousness for general database read workloads over multiple access methods.

ObliDB introduces a diverse array of new oblivious physical operators to accelerate oblivious SQL queries, giving speedups of up to an order of magnitude over naïve ORAM. It supports a broad range of queries, including aggregation, joins, insertions, deletions and point queries. We implement ObliDB and show that, on analytics workloads, ObliDB ranges from 1.1–19× faster than Opaque, a previous oblivious, enclave-based system designed *only* for analytics, and comes within 2.6× of Spark SQL, which provides no security guarantees. In addition, ObliDB supports point queries with 3–10ms latency, which is comparable to index-only trusted hardware systems, and runs over 7× faster than HIRB, a previous encryption-based oblivious index system that supports point queries.

PVLDB Reference Format:
S. Eskandarian and M. Zaharia. ObliDB: Oblivious Query Processing for Secure Databases. *PVLDB*, 13(2): 169-183, 2019.
DOI: https://doi.org/10.14778/3364324.3364331

Proceedings of the VLDB Endowment, Vol. 13, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3364324.3364331

1. INTRODUCTION

Many organizations outsource their databases to the public cloud to take advantage of its cost efficiency, high availability, and convenience. Due to the sensitivity of this data, both users and cloud providers would like strong privacy and security guarantees, ideally protecting against both external attackers and insiders that breach the cloud provider's security [13, 19, 75]. To address this problem, researchers have proposed approaches including property preserving encryption [35, 56, 57], trusted hardware [4, 10, 84], and algorithms to run specific computations securely [53, 78, 81], giving various tradeoffs between security, generality, and performance.

One of the most promising practical approaches to increase security is the hardware enclave [26, 27]. These enclaves provide an environment where a remotely verifiable piece of code can run without interference from the hypervisor and OS, accessing a small amount of private enclave memory and making upcalls to the operating system for I/O. Increasing availability of hardware enclaves has further spurred interest in strong cloud security guarantees [26, 27]. Enclaves are already available on many recent CPUs [6, 26] and will soon be offered on Microsoft and Google's public clouds [58, 64], making them a powerful technology to investigate for secure database hosting [60].

Unfortunately, although enclaves are powerful, they leave open one key threat: *access pattern* attacks. Applications that use an enclave to manage large amounts of data must still access data through the OS (e.g., to read new memory pages into the enclave or access the disk), so an attacker that controls the OS can see the pattern of addresses being accessed. This leaks a great deal of information, allowing attackers to learn details of both the data itself and users' queries on the data [39, 41, 54, 82]. The special case of encrypted databases has a long history of surprising leakage at the hands of access pattern attacks [3, 20, 36, 39, 42, 46, 59, 83].

In response to this threat, research has begun to press toward the goal of general-purpose *oblivious* (access pattern-hiding) SQL databases using hardware enclaves. The generic approach to establishing obliviousness uses Oblivious RAM (ORAM) [37, 73], which guarantees that any two sets of access patterns are indistinguishable from each other, so long as they are *of the same length*. Unfortunately, conventional query processing algorithms vary both the addresses and total number of memory accesses depending on data and queries, rendering generic use of ORAM alone insufficient. POSUP [40] and Oblix [49] explore oblivious indexes over encrypted data using specialized ORAM constructions as building blocks, but do not support general queries. Moreover, oblivious indexes alone do not fully solve the security problem: thus, an attacker can see *how many accesses to an index* occurred during a query operator.

On the other hand, Cipherbase [4] and Opaque [84] propose schemes that hide access patterns, but they are limited to workloads that *scan entire tables*. For example, Opaque relies on oblivious sorts over the entire dataset. These systems are not efficient for more general workloads that may also include point queries. Attempts to support general workloads, such as Obladi [28] and StealthDB [77], also lack key features – Obladi does not support indexes and requires operations to be processed in batches, and StealthDB does not provide integrity or hide access patterns to indexes. Thus, prior solutions do not provide algorithms for a general-purpose DBMS that combines queries of varying selectivity, the typical use case for outsourced databases (e.g. MySQL, Postgres, etc).

Our contributions. This paper introduces ObliDB, the first engine to provide efficient, oblivious read queries for relational workloads over multiple access methods. The key contribution is a set of oblivious query processing algorithms that work efficiently over both entire datasets and small subsets of data, closing the gap between prior work and general-purpose databases. Often the direct port of a standard operator into an oblivious version is not only slow but also inherently leaky. Our algorithms take advantage of knowledge about query selectivity to maintain obliviousness while outperforming naïve oblivious versions of standard techniques. For example, we offer four oblivious SELECT algorithms that vary their interaction with trusted/untrusted memory to achieve obliviousness while optimizing performance for different settings. Our algorithms only leak the structure of queries (hiding parameters) and the size of the output data, the same as Opaque's oblivious mode [84][1].

ObliDB's performance improves over prior systems by supporting multiple storage methods and including a query planner that obliviously chooses the best option among several algorithms to satisfy a given query. Unlike prior work, ObliDB provides two storage methods for its tables: a "flat" one, where the table is encrypted as a contiguous file and always scanned (as in Opaque and Cipherbase), and an *oblivious B+ tree* built over ORAM but modified to prevent leakage and performance penalties involved in a direct composition of B+ trees and ORAM. In particular, we hide the path taken in an index to retrieve records as well as the changes made to index data structures on insertions and deletions. Each table can be stored using one or both methods, similarly to how administrators can decide to create indexes in traditional databases. For instance, if a table is stored using both methods, ObliDB can use the index for point queries and the flat table for full-table aggregation queries.

Choosing between several algorithms to satisfy a query opens the possibility of leaking information about queries or data through algorithm choice. ObliDB's query planner mitigates this risk by basing optimization decisions on information already available to the attacker, such as table and query result sizes.

These features let ObliDB support a wide range of queries efficiently and securely. ObliDB supports selections, aggregations and joins, as well as efficient point and small range lookups, insertions, deletions, and updates. Since ObliDB's focus is on obliviously processing read queries, the engine does not provide full support for transactions, but techniques for concurrency and logging [28] can be added on top of ObliDB's algorithms and storage methods.

We implement ObliDB over Intel SGX [26] and evaluate it on diverse applications and find that it outperforms previous oblivious systems and achieves practical performance compared to systems with no security guarantees. For analytics, we compare ObliDB to Opaque [84] on the Big Data Benchmark [1] and find that it is competitive with Opaque on most queries, but can outperform Opaque by 19$\times$ on queries that can leverage indexes. ObliDB also comes within 2.6$\times$ of Spark SQL [7], which provides no security guarantees. For point queries, we compare to an open-source encrypted index and find that ObliDB outperforms the HIRB + vORAM of Roche et al. [63] by over 7$\times$. Moreover, point insertions, deletions and selects using ObliDB's indexes on a 1M row dataset take 3.6–9.4ms, which is acceptable for many applications and comparable to the other enclave-based indexes Oblix [49] and POSUP [40] that do not support the more general queries handled by ObliDB. Finally, we show that the choice of physical operators in ObliDB enables meaningful query optimization, yielding speedups of up to 11$\times$. ObliDB is open source at `https://github.com/SabaEskandarian/ObliDB`.

To summarize, our contributions are:

- Oblivious query processing algorithms optimized to run over both indexed and unstructured data, suitable for general purpose SQL databases.
- The design of ObliDB, an enclave-based oblivious database engine that efficiently runs general relational read workloads over multiple access methods.
- A lightweight query planner to choose between operator implementations offered by ObliDB.
- An implementation and evaluation of ObliDB with Intel SGX.

2. BACKGROUND & SECURITY GOALS

This section gives background on hardware enclaves, describes our threat model, and states our desired security properties. The fundamental goal of ObliDB is to protect both user data and query parameters from a malicious attacker with full power to manipulate components of the system lying outside a trusted hardware enclave. This includes protection against both direct observation/modification of data and indirect observation of access pattern leakage.

2.1 Background

A *hardware enclave* provides developers with the abstraction of a secure portion of the processor that can verifiably run a trusted code base (TCB) and protect its limited memory from a malicious or compromised OS [2, 26]. Developers get a small memory region hidden from the OS and cleared when execution enters or exits an enclave. In this memory, the trusted code can keep secrets from an untrusted OS that otherwise controls the machine. The hardware handles the process of entering and exiting an enclave and hiding the activity of the enclave while non-enclave code runs. Enclave code may require access to OS resources such as networking and I/O, so developers specify an interface between the enclave and OS.

An enclave proves that it runs an untampered version of the desired code through an *attestation* mechanism. Attestation involves an enclave providing a signed hash of its initial state (including the running code), which a client compares with the expected value and rejects if there is any evidence of a corrupted program.

2.2 Threat Model

We leverage a trusted hardware enclave to protect against an attacker with full control of the operating system (OS). We assume that our attacker has the power to examine and modify untrusted memory, network communication, and communication between the processor and enclave. Moreover, it can observe access patterns to untrusted memory and maliciously interrupt the execution of an enclave. We note that an OS-level attacker can always launch an indefinite denial of service attack against an enclave, but such an attack does not compromise privacy. We also allow our attacker to use arbitrary auxiliary information about the nature of data stored. For example, if a database is storing patient data, this includes the incidence of various diseases in the general population.

[1]We also support padding intermediate and final results of complex queries to a fixed size, similar to Opaque's pad mode [84], if desired.

We assume the security of the trusted hardware platform in that the enclave hides contents of its protected memory pages and CPU registers from an attacker with control of the OS and the attacker cannot subvert the remote attestation process by which the enclave proves its authenticity. Power analysis and timing side channels are out of the scope. Furthermore, we assume a secure channel exists through which a user can send messages to the enclave: for example, a client can establish such a connection to the enclave through TLS.

We implement our techniques on Intel's SGX [26] due to its popularity and widespread availability. Although several side-channel attacks based on abusing page faults, branching history, or speculative execution have been demonstrated against SGX's protected memory [18, 24, 43, 76, 80, 82], mitigations exist to handle some of these attacks [61, 68–70], and other hardware enclave designs avoid the pitfalls that leave SGX vulnerable [27, 44, 47]. In particular, the RISC-V based Sanctum [27] provides a developer abstraction similar to SGX with minimal performance overhead.

Limited Oblivious Memory. We assume a limited amount of oblivious memory is available to the enclave and protected from access pattern leaks (as in Opaque [84], to which we compare). That is, when the enclave makes a memory access inside this region, the operating system cannot determine which part was accessed. We note that SGX does not provide this kind of obliviousness. However, other similar enclave designs such as Sanctum or RISC-V's Keystone do provide it with little additional overhead, and the principles of the ObliDB system can run just as well on any other enclave architecture. Moreover, many of our oblivious operators, including the query planner, all SELECT algorithms except the "Small" algorithm, and one of our JOIN algorithms, maintain obliviousness even with an enclave completely vulnerable to these attacks, i.e. with 0MB of oblivious memory. The quantity of oblivious memory can be set as small as a few megabytes. It primarily serves to store the root position map for our ORAM implementation, and is also used to improve performance for the aggregation, grouped aggregation, and join operators (Section 4) and hide accesses to code pages. The amount of oblivious memory can be reduced at the cost of decreased performance, but we evaluate using 20MB or less in all our experiments. We will discuss the oblivious memory costs of each of our data structures and algorithms as we present them. We show how changes in the oblivious memory budget affect a more complex query in Section 7.1.

2.3 Our Guarantees

Our algorithms leak only the sizes of input, intermediate, and result tables and the physical query plan chosen. This security level is the same as Opaque's oblivious mode [84] and Cipherbase [4]. One of our `SELECT` algorithms also leaks whether the rows returned by a query form a continuous segment of the table queried (e.g. as in a range query), but this algorithm can be turned off if the leakage is deemed too large and is not used in our performance comparison to prior work. For situations where leaking intermediate table sizes is unacceptable, ObliDB also has a padding mode where all intermediate results are padded to a chosen size and query optimization is not applied, leaking nothing about queries but the logical plan and the upper bound on result sizes (like Opaque's padding mode [84]). ObliDB can also be combined with more sophisticated padding techniques, like [11], that provide differential privacy instead of full obliviousness to reduce the padding.

In general, whether the size of intermediate tables is sensitive depends on the application. For example, in a join of two tables where only one row is selected from each table (say, a customer record and the customer's latest order), the sizes of those intermediate results do not reveal much information; however, a query that selects all of the customer's orders (and then perhaps aggregates them) would let an adversary know how many orders the customer made. ObliDB includes a fused select + project + aggregate operator that can avoid leaking intermediate result sizes even in some multi-operator queries by combining these operations into a single, oblivious operator.

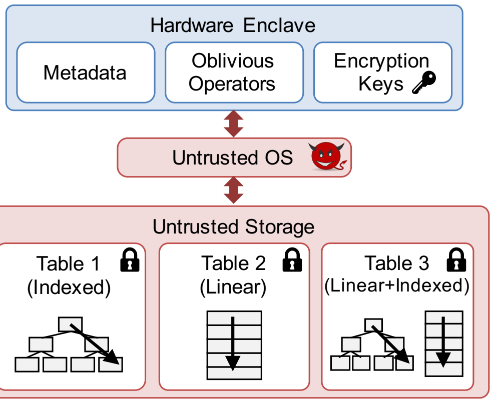


Figure 1: ObliDB runs in a hardware enclave and stores encrypted tables in untrusted memory accessed through the OS. It can store tables using either an oblivious B+ tree index, a flat array, or both.

Similar to leaking intermediate result sizes, leaking a query plan can reveal information about the structure of queries, e.g. whether an `INSERT` or `JOIN` query was executed, and whether an index was used. However, ObliDB hides query parameters such as *which* key in an index was requested. For example, by observing the physical plans used, an attacker could learn that a query performed a point lookup on an index, but not which key was requested, or whether the same key is requested again later. Likewise, ObliDB's query planner chooses between different implementations of selection and join operators based on the number of matching records, but the attacker does not learn which specific records were chosen (Section 5). In general, there is a fundamental tradeoff between information leakage and performance: if users want some queries to run faster than others, or to send back a smaller result set, an observer will learn that such a query was executed. However, in practice, hiding which data was accessed disables many access pattern attacks.

Finally, data at rest outside the enclave is encrypted and MACed, and leaks only its size. In both the padding and no-padding modes, we do not hide the number of tables in a database or which table(s) a query accesses. Beyond hiding data values and access patterns, we make the integrity guarantee that ObliDB catches any tampering with data by the malicious OS. We use a series of checks to protect against tampering within rows of a table, addition/removal of rows, shuffling of table contents, or rollbacks to a previous system state.

Appendix A presents a formalization of our security guarantees. We provide security arguments for the obliviousness of each storage method and operator as they appear in the text.

3. ObliDB ARCHITECTURE AND DATA STRUCTURES

Architecture overview. Figure 1 shows an overview of the ObliDB architecture. ObliDB consists of a trusted code base inside an enclave that provides an interface for users to create, modify, and query tables using our oblivious query processing algorithms, which we describe in Section 4. ObliDB stores tables, authenticated and encrypted, in unprotected memory and obliviously accesses them as needed by the various supported operators. The encryption key for data stored in unprotected memory always resides inside the enclave,

Table 1: Asymptotic performance of storage methods. Fast inserts on flat storage and large reads on indexed storage achieve better than expected asymptotics due to optimizations in Section 3.

Method	Flat	Index	Both
Space	N	$\sim 4N$	$\sim 5N$
Point Read	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insert	$O(1)$	$O(\log^2 N)$	$O(\log^2 N)$
Update	$O(N)$	$O(\log^2 N)$	$O(N)$
Delete	$O(N)$	$O(\log^2 N)$	$O(N)$

encrypting/decrypting blocks of data as they are written or read from unprotected memory. ObliDB can store data via two methods: flat and indexed. The indexed method consists of an ORAM with a B+ tree stored inside, whereas the flat method requires scanning the whole table on each query to ensure obliviousness.

ObliDB supports oblivious versions of the operators `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `GROUP BY` and `JOIN` as well as the aggregates `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`. It also includes a query planner that chooses between operator implementations for selection and join queries, which we describe in Section 5.

Since the core contribution of ObliDB lies in its oblivious query processing algorithms, it does not currently include support for transactions, but support for concurrency and logging can be added on top of the current operators. For example, a standard write-ahead log could be generically added to the system. Appends to such a log would not leak any additional information or affect obliviousness, as the only change would be to make a write to an encrypted log file before each insert/update/delete operation. Concurrent access to ORAM data structures could be facilitated by using an ORAM construction that supports parallel access [17, 21–23, 28, 52].

ObliDB can store data via two methods – flat and indexed – or combine both. We currently let system administrators decide which storage method(s) to use for each table based on the expected workload. Section 3.3 discusses costs and benefits of choosing either or both storage methods. ObliDB creates tables with an initial maximum capacity that can be increased later by copying to a new, larger table. We divide data into blocks of a configurable size[2]. Our current implementation assumes records are of fixed length and also stores a boolean flag with each record indicating whether it is in use.

Although encryption and oblivious data structures/algorithms ensure the privacy of data in ObliDB, additional protections stop an attacker from tampering with data. ObliDB MACs and encrypts every block stored outside the enclave, preventing the OS from modifying or adding new rows to tables. Each block of MACed data includes a record of which row(s) the block contains and a current "revision number" for that block, a copy of which ObliDB also stores inside the enclave. Each time a block is modified, we increment the block's revision number. 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. Rollbacks of system state are caught when the revision numbers of blocks do not match the last revision numbers for those blocks recorded in the enclave. Rollbacks on encrypted enclave data sealed to disk can be prevented either by storing revision numbers with the client or using an enclave rollback protection system like ROTE [48].

[2]In our current implementation, data in leaves and flat storage are fixed to one record per block.

3.1 Flat Storage Method

The flat storage method simply stores rows in a series of adjacent blocks with no built-in mechanism to ensure obliviousness of memory accesses, so every read or write to the table must involve accesses to every block to hide access patterns. As such, operators acting on these tables, as will be seen in Section 4, involve a series of scans over the entire table. This performs best with small tables, tables where operations will typically require returning large swaths of the table, or analytics that involve reading most or all of the table regardless of the need for obliviousness. The challenge in designing algorithms for this storage method lies in using the limited space of the enclave effectively to reduce the number of scans and data processing operations involved in each operator.

Insertions, updates, and deletions on flat tables involve one pass over the table, during which unaffected blocks receive a dummy write (overwriting a row with the data it already held, re-encrypted and therefore re-randomized). For insertions, the first unused block encountered receives a real write. For updates or deletion, any row matching the specified criteria will be updated or marked unused and overwritten with dummy data, respectively. All of these 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 scan over a table where each encrypted block is read and then written with a fresh encryption.

In tables with few deletions, an administrator can choose an alternative, constant-time insertion algorithm that saves the index of the last row where an insert occurred and always inserts directly into the next block in memory, skipping the scan. This insertion leaks no additional information beyond the sizes of tables because the access pattern of the insert does not depend on the content of the data except on the number of insertions made, which our adversary can already learn by observing the sizes of tables over time. Since every entry in a table is encrypted, an adversary will not be able to tell if later operations modify or even remove the inserted data, despite knowing ahead of time where each new record will be placed.

3.2 Indexed Storage Method

Standard insertion and deletion operations for B+ trees, even when combined with ORAM, leak information about the tree's internal structure, compromising obliviousness by splitting or merging nodes when they reach fixed threshold numbers of children. We ensure obliviousness by padding all insertions and deletions with additional dummy ORAM accesses until the number of accesses matches the worst-case number for the respective operation. The property of B+ trees that all data resides in the leaves of the tree means that any lookup already accesses the same number of nodes, so no modification is required for this case. Once each operation involves a fixed number of accesses to memory, we can leverage ORAM's security to guarantee obliviousness. We use the ORAM interface as a black box, so the details of the underlying ORAM can be omitted except to state that our choice of the Path ORAM scheme [73] incurs an $O(\log N)$ overhead for each access to memory. See Appendix B for details on the construction and formal guarantees of this scheme. We use a separate ORAM for each table because we already leak which table queries access, and using multiple smaller ORAMs is more computationally efficient than using a single monolithic one.

Two optimizations dramatically improve the performance of our oblivious B+ trees. First, our implementation operates on a "lazy write back" principle, only writing to the ORAM when necessary and otherwise keeping nodes in the enclave until they are no longer needed. Second, we remove all parent pointers from our implementation. Normal B+ tree implementations often have pointers in each

Table 2: Oblivious physical operators. N and M are table sizes (in number of rows), C the max chain length of the hash table, S is the total available oblivious memory, and R the number of rows in the output of a query. Selection over indexes incurs a multiplicative factor of $O(\log^2 N)$ in time complexity but runs over the smaller range of rows returned by the index, not a whole table. Each indexed table requires $8N$ Bytes of oblivious memory to store and access obliviously.

Algorithm	Time Complexity	Obliv. Mem.	Summary
Small Select	$O(N^2/S)$	S Bytes	Fast when data almost fits in enclave: scan table once per enclave-full of data
Large Select	$O(N)$	0 Bytes	Fast when almost entire table selected: copy table and clear unselected rows
Cont. Select	$O(N)$	0 Bytes	Fast when continuous segment of table selected: write to output table for each row of input (wrap around at the end), making dummy writes unless row is to be selected
Hash Select	$O(N \cdot C)$	0 Bytes	Use if other strategies don't apply: hash selected rows to location in output table
Naïve Select	$O(N \log N)$	$O(R)$ Bytes	Used only as baseline: ORAM operation for each row of table
Aggregate	$O(N)$	0 Bytes	Scan table, compute aggregate in one pass
Gp. Aggregate	$O(N)$	$O(R)$ Bytes	Store groups in hash table in oblivious memory and, for each row, check if there is a matching group in the table or add a new group to the table.
Hash Join	$O(\frac{N}{S} \cdot M)$	S Bytes	Block by block, make hash table from one table and see if rows of second table hash to same places – variant of standard hash join algorithm
Opaque Join	$O((N+M)\log^2(\frac{N+M}{S}))$	S Bytes	Sort tables by join column (use quicksort in obliv. mem. to accelerate), then linear scan to merge blocks of rows.
0-OM Join	$O((N+M)\log^2(N+M))$	0 Bytes	Bitonic sort tables by join column, then linear scan to merge matching rows

node to quickly find its parent (e.g. [8]). However, each time a tree splits or merges a node, all the children of nodes involved need to have their parent pointers updated, a very slow process in the regime where every node requires an ORAM write to update.

If the cost of maintaining both indexed and flat representations of data is too high, e.g. storage is limited or tables are very frequently updated, the indexed storage data structure can also be scanned linearly as a table using the flat method would be, ignoring the index structure. Our algorithms can treat both internal tree nodes and extra ORAM blocks as dummy blocks with no security consequences. This scan has additional overhead over directly using the flat storage method because of the extra space required by ORAM and the index structure, but in practice this overhead is less than 2.5×.

3.3 Complexity Analysis

Table 1 compares the asymptotic operations of standard read, insertion, and deletion operations as well as space overhead for each table type. The indexed method performs best on small reads that access one or a few rows of a table, whereas queries which expect to return large segments of a table should use the flat method, which performs faster than a linear scan over the contents of an index despite equal asymptotic runtimes. Using both storage methods, while incurring the cost of both for insertions and deletions, proves effective when queries of diverse selectivities run on the same data. We empirically measure these tradeoffs in Section 7.2. In terms of storage overhead, our oblivious indexes inherit the 4× storage overhead required by the Path ORAM [73] we use and each encrypted block is slightly larger than a plaintext block (as is always the case with authenticated encryption). All other sources of storage overhead, e.g. that required for data integrity measures, only add a few bytes to each block of data, amounting to less than 1% additional overhead. Path ORAM contains a data structure that we need to store in oblivious memory at a cost of 8 Bytes of memory per row of an indexed table. We can reduce the oblivious memory required by using a *recursive* ORAM as described in Appendix B or remove it completely via a *doubly-oblivious* ORAM as described by Oblix [49] or ZeroTrace [66].

4. OBLIVIOUS QUERY PROCESSING

The key to executing queries in ObliDB is a set of new oblivious query processing algorithms that can efficiently run queries over either flat or indexed storage. This section describes our oblivious query processing algorithms for a large subset of SQL, including selection with conditions composed of arbitrary logical combinations of equality or range queries, joins, aggregates (count, sum, max, min, average), and grouped aggregation. In cases where the storage method used for a table admits multiple algorithms to satisfy a given query, ObliDB's query planner chooses the algorithm that maximizes performance. At a high level, the planner makes a quick preliminary scan of the table being queried and uses known information about input and output table sizes to make an optimization decision without leaking more information about the query or data. Details on the query planner appear in Section 5.

We will begin by discussing the algorithms in the context of flat storage and then discuss the modifications needed for compatibility with indexes, if any. Each operation is accompanied by a security argument. Since stored rows do not persist inside the enclave between queries, there is no opportunity for a caching side channel based on which rows can be retrieved faster in a subsequent query. Thus the whole engine runs obliviously so long as each of the operators is individually oblivious.

Whenever we refer to rows of a table being read or written without explicitly stating where they are stored, it is implied that the data resides in unprotected memory, is decrypted before being read inside the enclave, and is re-encrypted before being written back outside. Table 2 summarizes our algorithms and their complexity. We evaluate the performance of our operators in Section 7.

We refer to the subject of a query as table T and the results as table R. We leak only the sizes of T and R. In the following, the enclave learns the size of R from the query planner before executing the operator, allowing output data structures of the appropriate size to be allocated before scanning the data needed to fill them.

4.1 Oblivious Selection

Selection queries involve choosing elements from a table that match a given predicate (e.g. `date>'2018-09-01'`). One natural way to implement a `SELECT` operator would be to sequentially read each record in the targeted table and write out the row if it should be selected. Despite touching each row in the table once, this implementation does not provide obliviousness. An adversary observing the pattern of accesses to the input and output tables would know whether a row is written to the output after each read: both tables are accessed each time a row is selected, but only the input table is accessed when the a row is not selected. For example, consider a table `Checkins` that logs when employees enter

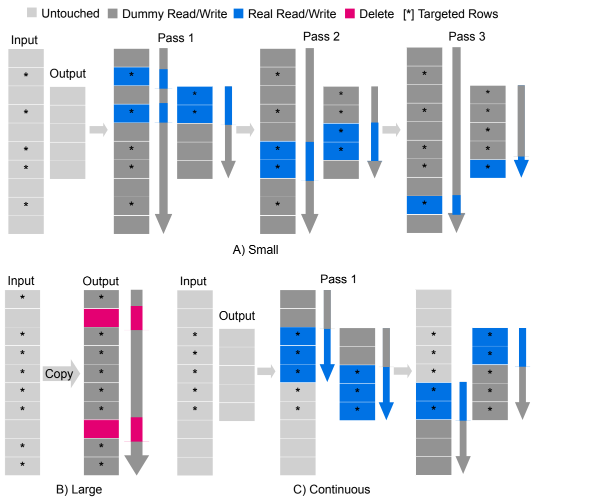


Figure 2: Small, Large, and Continuous SELECT algorithms. The enclave in this example is only large enough to store two rows of data, so the Small (A) algorithm, which scans the table once per enclave-full of data, takes three passes to complete. The Large (B) and Continuous (C) algorithms always make only one pass. The Large algorithm copies the input table and clears unselected rows, and the Continuous algorithm writes to the output table for each row of the input table, wrapping around at the end, making dummy writes for rows that are not selected.

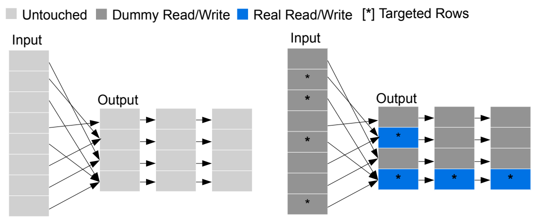


Figure 3: Hash SELECT algorithm. Left: The access pattern for any input and output table sizes is fixed because the hash of the block number is taken, not of the data itself. Right: a sample execution. Each input cell is read followed by either a dummy or real write following the arrows to the right. Our implementation uses double hashing in addition to the chaining shown.

or exit an office building. An attacker observing access patterns on the query `SELECT * FROM Checkins WHERE uid=3172 AND date>'2018-01-01'` could infer from the chosen rows when the user had entered the building or (without seeing the query) what dates the query targets.

To defend against this and other subtle attacks, including those based on prior knowledge of the data distribution, ObliDB provides the following oblivious `SELECT` algorithms (summarized with their complexities in Table 2, Figure 2, and Figure 3). In each algorithm, ObliDB has access to the output table size $|R|$ based on information provided by the query planner during its initial scan of the data.

Naïve. included as a baseline, the naïve oblivious algorithm is a direct translation of a non-oblivious `SELECT` to an oblivious one via ORAM. After examining each row, it executes an ORAM operation. If the row is included in the output, it makes a write. If not, it makes a dummy read to an arbitrary block. There must be an ORAM operation after reading each row or else an adversary would know that any row which did not coincide with an ORAM operation was not included in the output. After completing the scan of the input table, it copies the contents of the ORAM to the flat storage format and returns it. This algorithm requires $4|R|$ Bytes of oblivious memory to store the ORAM it uses to build the output table.

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

Small. In the case 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's oblivious memory 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 is effective for small tables. Since it requires oblivious memory to store rows in the enclave, this algorithm uses whatever quantity of oblivious memory is made available to it. However, reducing the amount of oblivious memory does not affect correctness, only performance. This algorithm is depicted in Figure 2A.

This algorithm leaks only the sizes of tables T and R because every pass over the data consists of one read to each row 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.

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. Obliviousness holds because the copy operation does not depend on the data copied and we clear unselected rows with a read followed by a write to each block of the table, revealing only the size of T. This algorithm, shown in Figure 2B, uses no oblivious memory.

Continuous. Should the rows selected form one continuous section of the data stored in the table, ObliDB requires only one pass over the table, as shown in Figure 2C. Such a situation can arise when range queries are made over sorted data such as names, dates, ID numbers, etc, or retrieved in the same order they were inserted. To handle such queries, ObliDB first creates table R. Then, for the ith row in table T, if that row should be in the output, it writes the row to position $i \bmod |R|$ of R. If not, it makes a dummy write. Since the rows that need to be included in R make up one continuous segment of T, this procedure results in exactly the selected rows appearing in R. This algorithm uses no oblivious memory.

In addition to the sizes of tables T and R, the fact that ObliDB chooses this algorithm over one of the other options leaks that the result is drawn from a continuous set of rows in the table. In these cases, however, knowing that users are selecting a range is often not so sensitive as what that range is, which we do hide. 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 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.

Hash. If none of the preceding special-case algorithms apply, ObliDB uses a hashing solution illustrated in Figure 3. 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. Otherwise, we make a dummy write to the $h(i)$th position in R. Since the hash is on the index of the row in the data structure and not over the actual contents of a row, information about the data cannot be leaked by access patterns when rows are written to R, and the algorithm uses no oblivious memory.

The algorithm above needs a couple changes to ensure that we properly handle collisions while maintaining obliviousness. We can use standard techniques to resolve collisions, but in order to maintain obliviousness, every row of T must make the same accesses to memory regardless of whether it is included in R. We handle this by having every write make as many memory accesses as in the case of the worst expected chain of collisions, regardless of whether the row under consideration in T actually appears in R. Following the guidance of Azar et al. [9] to get small probability of failure, 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. As mentioned above, since we hash the index of the row in the data structure and not 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, we leak only the sizes of T and R.

Selection over Indexes. Selection over the indexed storage method works identically to flat storage except that the linear scan begins inside an ORAM at a point specified by an index lookup. 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 indexed by student IDs, the query `SELECT * FROM students WHERE NAME = ''Fred'' AND ID > 50 AND ID < 60` leaks that 9 rows were scanned in the execution of the query. We consider this leakage to be included in the sizes of intermediate tables, as this query is equivalent to a query plan which selects a continuous segment from an index and then selects a noncontinuous segment from the returned table. Padding can hide this leakage. The Large algorithm is not used in the indexed storage method because indexes are meant for queries that request a small fraction of a table, not almost all of it. In terms of complexity, algorithms running over the index have the same complexity as their flat counterparts in Table 2, but each algorithm incurs a $\log^2 |T|$ multiplicative overhead due to use of the index structure for reads. On the other hand, the actual query runs on table T', the range of rows returned from the query to the index, instead of the full table T.

Example. Consider the following queries on a flat table:

```
SELECT * FROM Checkins WHERE date='2018-08-14'
SELECT * FROM Checkins WHERE date>'1900-01-01'
```

When ObliDB receives such queries, it first runs the query planner, which determines which algorithm will perform best for the given query. Since the first query requests rows from a specific date, there will only be a handful of entries, so it chooses the Small algorithm. On the other hand, the second query likely predates the construction of the building and will therefore select every row of the table. The planner will choose the Large algorithm for this query.

For query 1, ObliDB will scan the table, storing any records from the chosen date inside the enclave until the end of the scan. Then it will write all the matching rows to an output table at once. If the enclave fills before reaching the end of the table, ObliDB will finish the scan without storing any more records and then conduct a second scan that begins storing records in the enclave where the first left off. For query 2, ObliDB copies the table to create an identical output table and then makes a scan of the copy to delete any rows from before the year 1900. If the above queries were part of a larger query or if the user decided to make a subsequent query on the output, ObliDB would then use the output of these queries as the input to the next query and run the appropriate operator.

4.2 Oblivious Aggregation & Grouping

An aggregate over a subset or entirety of a table requires only one pass over the table where we calculate the aggregate cumulatively based on the data in each row in $O(|T|)$ time. We keep the aggregate statistic inside the enclave. Since the memory access pattern of this operation always involves sequential reads of each block in the table followed by an update to the aggregate statistic, nothing leaks beyond the size of table T and no oblivious memory is needed.

We handle grouped aggregation similarly, except an array in the enclave keeps track of aggregates for each group. Since we need to hide which group's aggregate each row modifies, we require 4 Bytes of oblivious memory to store the aggregate for each group. We use a hash bucketing approach where each group's value is hashed and inserted into a hash table in the enclave. Each row scanned is hashed and checked against the hash 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 to the hash table as a new group. This method results in running time $O(|T|)$. If the number of groups becomes so large that the hash table cannot fit in oblivious memory (a situation that did not arise in any of our experiments, as each additional group requires very little space), we could switch to using the sort-and-filter approach introduced by Opaque [84] which runs in time $O(|T| \log^2 |T|)$.

Combining Aggregation and Selection. In order to improve performance and avoid leaking intermediate table sizes for common queries that combine selection, aggregation, and grouped aggregation, ObliDB provides a combined select/group/aggregate implementation. The SELECT algorithms described above require multiple passes over a table in order to provide obliviousness, but if the next query only takes an aggregate, the obliviously produced intermediate table can immediately be discarded, wasting all the effort of creating it. We remove this inefficiency by computing aggregates directly over the input table while filtering it for the selection criteria. Since selected rows don't need to be written anywhere, we skip the extra effort required by general-purpose oblivious selection.

4.3 Oblivious Joins

Arasu and Kaushik [5] and Opaque [84] introduced oblivious join algorithms that are also applicable to ObliDB. We support Opaque's join and two additional algorithms: an oblivious hash join and a variant of the Opaque join that requires *no oblivious memory*.

Oblivious Hash Join. We implement a variant of the standard hash join algorithm [32]. We refer to the two tables being joined as T_1 and T_2. We make a hash table out of as many rows of T_1 as will fit in the enclave and then hash 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 table. If there is a match, the joined row is written. If not, a dummy row is written to the table at that position. Since each comparison between the tables always results in one write to the next block of the output structure, the memory access pattern of this algorithm is oblivious. Like the traditional join algorithm of the same name, the complexity of our oblivious hash join is $O(|T_1| \cdot |T_2|)$. Since it needs oblivious memory to store the hash table, this algorithm uses whatever quantity of oblivious memory is made available to it. However, as with the Small selection algorithm, reducing the amount of oblivious memory does not affect correctness, only performance. A side effect of this algorithm's obliviousness is that the size of the output table data structure will always be $|T_1| \cdot |T_2|$. Our remaining join algorithms focus on the case of foreign key joins where the maximum output size is at most the greater of $|T_1|$ and $|T_2|$.

Oblivious Sort-Merge Join. We support two sort-merge join algorithms for foreign key joins. First, we re-implement the Opaque join. This algorithm begins by putting the contents of both tables into one new table. Then it uses quicksort to sort chunks of the data that fit inside an enclave's oblivious memory and merges the chunks with a bitonic sorting network. Finally, one linear scan down the new sorted table eliminates rows that do not have matches and merges matching rows to form the output table. In addition to requiring oblivious memory, using quicksort to accelerate the join may open timing side channels as well, a factor that must be considered in choosing a join algorithm for a particular application.

Next, we support a variant of the Opaque join that requires *no oblivious memory* and operates by running a bitonic sort over the rows of both tables according to the join criteria without quicksorting chunks inside of oblivious memory first. The bitonic sort can be implemented obliviously because it always makes the same set of comparisons independent of the data being sorted. As an optimization, when the size of the recursive sort becomes small enough to fit inside of the enclave, we carry out the sort inside the enclave to avoid paying the cost of calls to memory outside the enclave. This has no impact on obliviousness but speeds up memory access by reducing communication between the enclave and untrusted memory while sorting. We call this the *0-OM* join.

We compare the performance of our join algorithms in Section 7 and state their complexities in Table 2. We could reduce the $O(\log^2 n)$ terms in the sorting to $O(\log n)$ using a randomized shellsort [38] (as discussed by Arasu and Kaushik [5]) at the cost of making the correctness of the sorting algorithm probabilistic.

5. QUERY PLANNER

Our query planner picks which selection and join algorithms to use based on statistical information on the input and output table sizes. Our main insight is that we can use the information already leaked by the data structures and output sizes in ObliDB to minimize additional leakage from query planning. In Section 7.2, we find that the planner can improve query performance by $4.6\text{-}11\times$. The query planner is not used in padding mode, where we hide output sizes.

ObliDB runs the query planner at runtime whenever it encounters a selection or join operator. For each selection, the planner begins with a fast scan over the data, during which it keeps track of (1) the number of rows satisfying the predicate and (2) whether those rows are adjacent in the input table. The enclave saves the computed output size to pass into selection operators that pre-allocate output storage. Based on the ratios of number of output rows to available oblivious memory and input table size, the planner decides which variant of the selection operator to use. A precomputed set of thresholds decide when to run each operator. For maximum flexibility, users can also manually choose to force a particular operator.

Note that we cannot simply return the query result in the first scan over the data, as a naïve one-pass algorithm would violate obliviousness. Instead, we must run one of our oblivious operators. Since many of these operators need to know the size of the output table up-front (to allocate memory for the results), the planner's first scan to compute statistics is often "for free."

We adopt a similar approach to choose the appropriate algorithm for foreign key joins, but planning for joins requires even less information than selection. Observe that all the join algorithms in Section 4.3 generate output tables and do computation of the maximum possible size given the input table sizes. As such, the output table, although it may contain many dummy rows that are marked as unused, will reside in a data structure whose size can be calculated directly from the sizes of the input tables. Moreover, this property means that the performance of the join algorithms will depend only on the input table sizes and will otherwise be the same *regardless of the selectivity* of the join. These properties taken together allow us to make effective optimization decisions based only on knowledge of the sizes of the tables joined and the amount of oblivious memory available inside the enclave. Similar to selection, we pick which join algorithm to use based on the ratio of the available oblivious memory to the size of the first input table. If the amount of oblivious memory is large relative to the size of the first table, we always use the hash join. Otherwise, we plug in the table sizes and amount of oblivious memory into expressions denoting the asymptotic runtimes of the join algorithms and choose the smaller result. Section 7.2 shows that this approach works well in practice.

Table 3: Data sets used in the Big Data Benchmark [1].

Table Name	Rows	Notes
USERVISITS	350,000	Server logs for many sites. Data from the Big Data Benchmark [1].
RANKINGS	360,000	URLs, PageRanks, and average visit durations for many sites. Data from the Big Data Benchmark [1].

Security. Performance improvements due to query planning intrinsically require leakage because the benefits of planning arise from the fact that different algorithms perform better for different data and queries. Our choice of physical operator reveals two pieces of information. First, for selection, is the number of matching rows. Since the non-padded execution mode already reveals the output size of the result, this adds nothing to the overall leakage of the system. Second is whether or not the rows returned by a query form a continuous segment of the table queried. This is revealed by the choice of the Continuous algorithm from Section 4, which occurs if the rows to be returned are continuous. The Continuous algorithm can optionally be disabled, causing optimization to leak *no additional information* beyond what is already revealed through output sizes (this is the configuration used for our comparison to prior work in Section 7). Planning for joins leaks even less, as it relies only on the sizes of the tables being joined and the oblivious memory available.

Our query planner always has the same memory access pattern for selection queries: read each row, update statistics, and perform a table lookup to select an algorithm at the end. As such, the only leakage introduced by the query planner comes from its final choice of which physical operator to run, not the optimization algorithm itself. For joins, the planner only reads the recorded sizes of the input tables and makes no other memory accesses.

6. IMPLEMENTATION

We implemented ObliDB on Intel SGX [26], including the storage methods from Section 3 as well as the oblivious operator algorithms and query planner of Sections 4 and 5. Our implementation consists of over 14,000 lines of code and builds upon the Remote Attestation sample code provided with the SGX SDK [2] and the B+ tree implementation of [8], the latter of which was heavily edited in order to support our ORAM memory allocator. We use SGX SDK libraries for encryption, MACs, and hashing. We instantiate our ORAM scheme with a nonrecursive Path ORAM [73]. See Appendix B for details on this scheme and the oblivious storage and performance implications of recursive vs nonrecursive Path ORAM.

Our current implementation consists only of the core database engine and lacks some components of a full-featured DBMS, e.g. transaction management and persistence to disk. In our evaluation, we compare ObliDB only to in-memory tables on other oblivious systems to avoid giving it an unfair advantage. It would be straightforward to replace ObliDB's external memory with disk storage, as accesses to both ORAM and flat tables are already block-oriented. We discuss options for supporting transactions in Section 3.

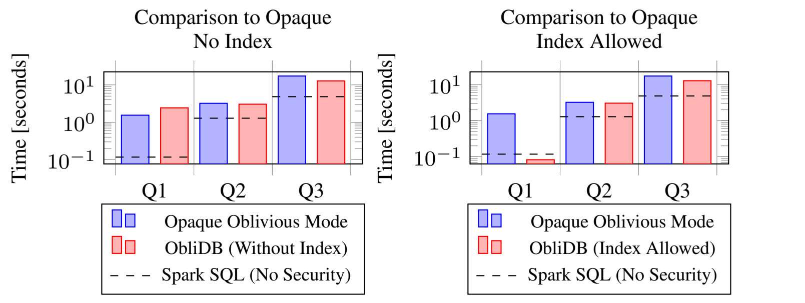


Figure 4: ObliDB outperforms Opaque Oblivious [84] by 1.1-19× and never runs more than 2.6× slower than Spark SQL [7] on Queries Q1-Q3 of the Big Data Benchmark [1]. Even without use of an index, ObliDB performs comparably to Opaque Oblivious.

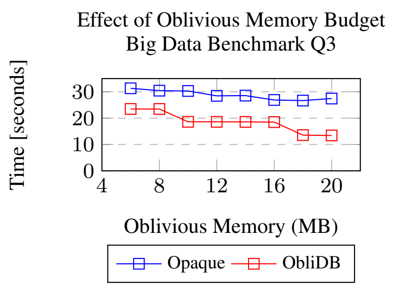


Figure 5: Performance of ObliDB and Opaque [84] on Big Data Benchmark Query 3 as oblivious memory varies.

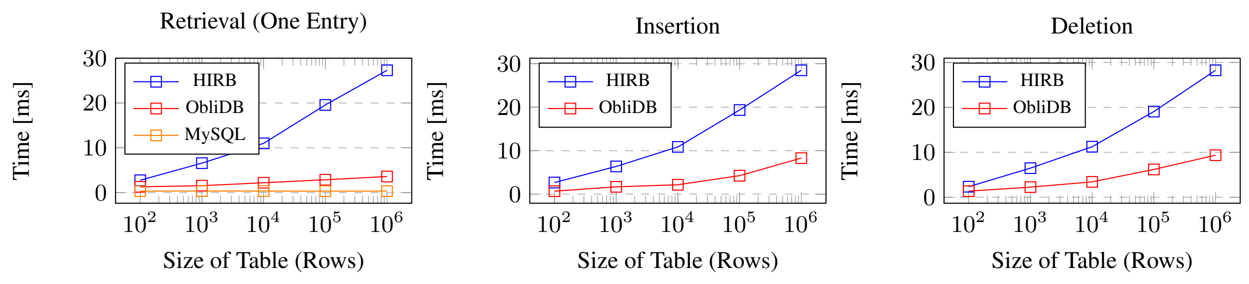


Figure 6: ObliDB's oblivious indexes outperform the HIRB tree + vORAM oblivious map construction.

7. EVALUATION

We evaluate ObliDB on multiple datasets, comparing to prior private database systems and widely used non-private systems. We use a subset of the data available from the Big Data Benchmark [1], shown in Table 3, as well as larger synthetic data. In addition, we measure the overhead of ObliDB's padding mode, demonstrate the effectiveness of ObliDB's query planner, study the impact of the chosen storage methods, and examine tradeoffs in join algorithms through a series of microbenchmarks. We evaluated ObliDB on an Intel Core i7-6700 CPU @3.4GHz with 8GB of RAM running Ubuntu 16.04 and the SGX SDK version 1.9. The comparison of join algorithms was done on the same machine running Ubuntu 18.04 and the SGX SDK version 2.5.

We find that ObliDB can leverage its indexes to achieve order of magnitude performance improvements over previous private database systems. In particular, ObliDB matches Opaque [84] for scan-based queries on flat tables but can outperform it by 19× when using an index. ObliDB also performs over 7× faster than HIRB [63], an oblivious map scheme, and comes within a factor of 2.6× the performance of the non-private Spark SQL system.

7.1 Comparison to Prior Work

Comparison to Opaque. Figure 4 compares ObliDB with Opaque's oblivious mode [84, 85] and Spark SQL [7], which provides no security guarantees, on queries 1-3 of the Big Data Benchmark [1] on tables of 360,000 and 350,000 rows. We use the same queries and parameters used by Opaque: 1000, 8, and 1980-04-01 are the parameters used for queries 1-3 of the benchmark, which target selection, grouped aggregation, and joins respectively. Opaque also uses an SGX enclave and can be configured in either "encryption" mode or "oblivious" mode, which hides access patterns to data, but by means different from ours. We compare to Opaque's oblivious mode and run it in single node configuration. We limit oblivious memory to 72MB for Opaque (as in its own evaluation) and 20MB for ObliDB, but neither system needed the full oblivious memory allowed. To compare fairly in terms of the leakage, we disable the Continuous selection algorithm in this comparison.

We began by configuring ObliDB to use only the flat storage method, as Opaque does, and found that ObliDB performs comparably to Opaque, slightly worse on query 1 and slightly better on queries 2 and 3. Next, we used the combined storage method. An oblivious index allows ObliDB to outperform Opaque by 19× on query 1 since this query scans a small part of a table whereas Opaque and spark SQL, which primarily handle analytic workloads, scan the entire table. Indexes do not provide a speedup on queries 2 and 3 which scan most of the input anyway. ObliDB is only 2.4× and 2.6× slower than Spark SQL on queries 2 and 3.

We also tested scan-based queries against our indexes to see how ObliDB performs on frequently-updated data too expensive to maintain in flat storage. These queries performed about 2× slower than on flat tables. Thus, unlike prior, flat-only systems, ObliDB performs analytics relatively quickly on "live" tables frequently updated with point insertions and deletions.

Impact of Oblivious Memory Budget. Figure 5 shows the performance of ObliDB and Opaque's oblivious mode on query 3 of the Big Data Benchmark as the quantity of oblivious memory varies from 6MB to 20MB, beyond which the performance of ObliDB remains steady. We chose this query because its performance is most affected by an increase in oblivious memory for both systems. Both systems' performance improves as we add more oblivious memory, but Opaque improves gradually whereas ObliDB decreases in steps as the amount of oblivious memory makes the blocks of the nested loop join large enough to reduce the overall number of scans of the second table being joined. In total, the increase from 6MB oblivious memory to 20MB results in a 1.77× speedup for ObliDB.

Comparison to HIRB. Next, we compare ObliDB's performance to The HIRB Tree + vORAM [63] secure index structure. Unlike ObliDB, HIRB neither supports range queries nor uses hardware enclaves. Source code for other SGX-based oblivious indexes is not yet publicly available, so we cannot compare to them directly,

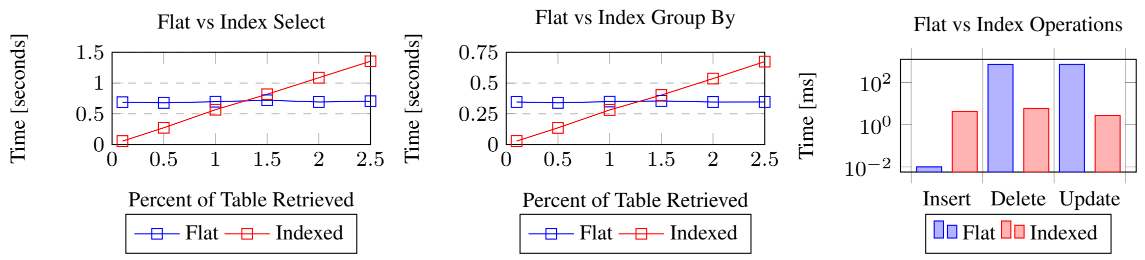


Figure 7: Comparison of flat and indexed versions of operators over 100,000 rows of synthetic data. Flat scans do better when more data needs to be accessed, but the indexed storage method performs far better for small queries.

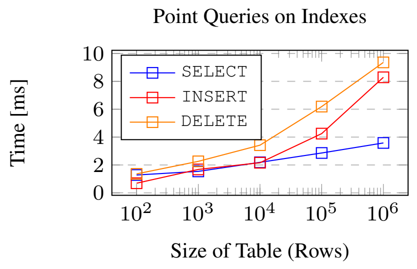
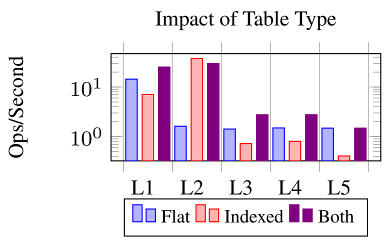


Figure 8: Point queries for various table sizes. Query time is polylogarithmic in table size.



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

Figure 9: Flat, indexed, and combined representations of a 100,000 row table for five workloads. Point reads access 1 row, small reads access 50, and large reads access 5% of the table.

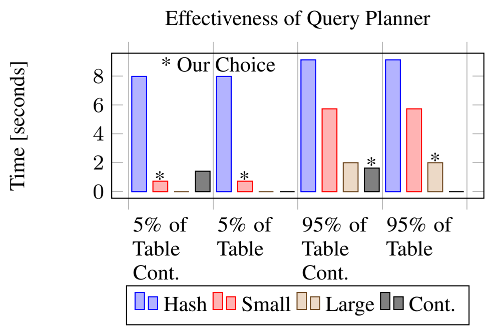


Figure 10: Our query planner picks the best algorithm for `SELECT` queries. Bars omitted when an algorithm is not applicable.

although reported numbers for Oblix [49] and POSUP [40] appear to be within several milliseconds of ours. Despite its reduced functionality and differing security assumptions, HIRB provides a good point of comparison as a practical system attempting to solve similar problems. We compare against it with a replication of the performance experiment in its original paper.

Figure 6 compares the point query performance of ObliDB's oblivious indexes with a HIRB tree + vORAM oblivious map [63] and MySQL. Although ObliDB does not support transactions, we include comparisons of insertion and deletion times over our indexes to demonstrate the performance of the data structure (the comparison is fair since HIRB also implements a key-value store with no notion of concurrency or durability). We instantiated both the table in ObliDB and the HIRB tree with 64-Byte data entries and allocated the underlying vORAM with bucket size 4096, a somewhat larger size than our own ORAM's buckets (HIRB performed worse on smaller bucket sizes). On tables of 1,000,000 rows, ObliDB outperforms HIRB by 7.6$\times$ in point selection and by 3$\times$ on insertions and deletions. While still an order of magnitude slower than MySQL for point queries on larger tables, network latency from user to cloud can be tens of milliseconds, rendering the difference insignificant.

The HIRB construction considers a "catastrophic attack" scenario which compromises the system holding the ORAM client, and they design the HIRB tree to provide history independence and secure deletion even under this attack. Since our work relies on the security of the hardware enclave and keeps the ORAM client inside the enclave, the additional security properties desired by HIRB come for free in our setting, explaining our improved performance. Both our work and the HIRB tree make use of padding for obliviousness, but each uses different optimizations to minimize padding.

7.2 Microbenchmarks

Impact of storage method. Figure 7 compares our storage methods on various queries. Flat scans perform better when more rows are returned, but smaller queries perform much better with an index. Indexed `DELETE` and `UPDATE` queries outperform flat ones, but the fast flat `INSERT` query outperforms the indexed `INSERT`. The flat storage method's performance (outside of constant-time insertions) degrades linearly in table size, but point operations on indexes take polylogarithmic time. Figure 8 shows how point queries scale.

Often a combined table representation that maintains both storage methods for the same data proves effective. Although ObliDB pays insertion and deletion costs for both methods, it can use the better representation for each query, an important benefit because many real-world workloads rely heavily on different kinds of reads. Figure 9 shows ObliDB running various workloads with flat, indexed, or both kinds of tables. One storage method alone sometimes performs well, but a combined representation often performs best.

Impact of query planner. Figure 10 shows ObliDB's choice of `SELECT` algorithms on queries that retrieve 5% and 95% of a 100,000 row table. The "Hash" algorithm is best asymptotically, but we pick an algorithm that performs 4.6-11$\times$ better in practice.

Join algorithm comparison. Table 4 compares the performance of ObliDB's join algorithms on foreign key joins for varying oblivious memory and table sizes. As explained in Section 5, input table sizes and oblivious memory are the only factors that affect join performance. Access to larger amounts of oblivious memory is particularly effective in speeding up the hash join algorithm because

Table 4: Foreign key joins with tables and oblivious memory of varying sizes. The fastest and slowest algorithm in each configuration are shown in blue and red, respectively. Reported number is average of 5 runs, standard deviation is always less than 8% of average. Our planner picks the fastest algorithm for every entry.

Join Performance – 500 Rows Obliv. Mem.						
Table 1	5,000 rows			10,000 rows		
Table 2	Hash	Opaque	0-OM	Hash	Opaque	0-OM
100	0.023s	0.205s	0.404s	0.047s	0.535s	1.017s
1,000	0.141s	0.259s	0.531s	0.274s	0.553s	1.092s
5,000	0.667s	0.529s	1.019s	1.289s	0.822s	1.585s
10,000	1.267s	0.808s	1.581s	2.592s	1.340s	2.497s
25,000	3.300s	2.078s	3.825s	6.540s	3.046s	5.337s

Join Performance – 7,500 Rows Obliv. Mem.						
Table 1	5,000 rows			10,000 rows		
Table 2	Hash	Opaque	0-OM	Hash	Opaque	0-OM
100	0.007s	0.044s	0.202s	0.015s	0.154s	0.533s
1,000	0.016s	0.053s	0.244s	0.031s	0.165s	0.571s
5,000	0.050s	0.149s	0.520s	0.103s	0.335s	0.792s
10,000	0.095s	0.334s	0.794s	0.192s	0.431s	1.282s
25,000	0.241s	0.938s	2.041s	0.479s	1.040s	2.869s

the size of the oblivious memory determines how many times chunks of the first table need to be made into hash tables, which determines the number of scans required of the second table. A large oblivious memory results in a join whose running time is almost linear in the size of the tables. For small oblivious memory, the performance behaves as expected of standard hash and sort-merge join algorithms: the hash join performs better for small tables but rapidly becomes worse than the sort-merge join as table sizes increase.

The Opaque join always outperforms the variant that requires no oblivious memory because the two joins run effectively the same overall algorithm, with the Opaque join using oblivious memory to accelerate sorting. The 0-OM join gets faster as the amount of oblivious memory increases because of our optimization that does oblivious sorting inside the enclave when there is space available without compromising obliviousness (to save on enclave communication costs). As such, the algorithm gets faster with more enclave memory, regardless of whether the memory is oblivious.

Impact of padding mode. Padding mode additionally hides the sizes of tables, intermediate results, and final outputs—comparable to the padding mode described but not evaluated by Opaque [84]. We evaluate this mode by running queries on the CFPB table of 107,000 rows padded to 200,000 rows. Our aggregate query with the flat storage method had a 4.4× slowdown and a select had a 2.4× slowdown. The larger slowdown for aggregates results from the padding algorithm padding to the maximum supported number of groups for aggregates—in this case, 350,000. We did not evaluate padding mode for indexes as the benefit of indexes arises from knowledge of the selectivity of a query, the exact information padding hides. To our knowledge, no comparable system has implemented a pad mode, so we cannot compare to prior work. The results do, however, represent reasonable slowdowns for inflating a table's size by approximately 2× with padding.

8. RELATED WORK

Encrypted Databases. Fuller et al. [35] summarize prior work on cryptographically protected databases. The well-known CryptDB [56] enables a tradeoff between security and performance, encrypting fields differently according to security needs. Arx [55] uses only strong encryption but leverages special data structures to allow search. Other solutions, including Demertzis et al., Sophos, and Diana [15, 16, 30], use searchable encryption. Although all of these systems encrypt *data*, they can leak *access patterns* [20, 41, 50, 83].

SGX Databases. StealthDB [77] is a legacy-compatible, partially-oblivious database that does not provide integrity or hide access patterns to indexes. VeritasDB [71] provides integrity but not privacy. POSUP [40] uses ORAM and SGX to search/update encrypted data and Cui et al. [29] use SGX to speed up search over encrypted data, but both support a more limited range of functionalities than ObliDB. More recently, Oblix [49] builds an oblivious index that requires no obliviousness assumptions inside the enclave, and Obladi [28] considers concurrent ACID transactions but does not support indexes and only processes operations in batches over discrete time epochs. Opaque [84] and Cipherbase [4] support only analytics queries that scan all the data, relying on oblivious sorts of an entire input table.

EnclaveDB [60] is an SGX-based DBMS that does not hide access patterns. TrustedDB [10] uses older trusted hardware designs to build a protected database, but also does not protect access patterns. Many works also implement variations of other analytics systems on SGX [14, 33, 34, 51]. M2R [31] and VC3 [67] provide MapReduce and cloud data analytics functionalities, and HardIDX and LPAD [34, 74] build key-value stores that are not oblivious.

General-Purpose Oblivious Computing. ZeroTrace [66] builds ORAM-based oblivious memory primitives over SGX, Pyramid ORAM [25] builds an efficient ORAM for use in enclaves, and ObliVM [45] compiles oblivious versions of programs. By specializing data structures and operators for ORAM, ObliDB outperforms naïve ORAM translations of database algorithms. Wang et al. [79] optimize data structures over ORAM, focusing on the case of recursive ORAM. Some of their techniques could complement our indexes when using a recursive ORAM position map. Roche et al. [63] build a history-independent "HIRB tree" over an ORAM with variable-sized blocks, but do not support range queries. As seen in Section 7.1, our indexes are up to 7× more efficient.

We use the Path ORAM [73] in our implementation, but any other ORAM could replace it with no other changes to the system. For indexed storage, where ORAM accesses dominate the cost of each operator, using a newer scheme such as Ring ORAM [62] would result in performance improvements corresponding to the approximately 1.5× improvement of Ring ORAM over Path ORAM. Unlike Ring ORAM, other ORAM optimizations designed for systems that provide cloud storage, such as Oblivistore [72], CURIOUS [12], and TAOstore [65], focus on reducing communication costs for the remote storage use case, which is less applicable in ObliDB, where the trusted and untrusted memory reside on the same device.

9. CONCLUSION

ObliDB closes the gap between previous enclave-based query processing engines and oblivious indexes by combining new oblivious query processing algorithms with accompanying data structures and an oblivious query planner. While obliviousness has a cost, ObliDB approaches practical performance: it is competitive to 19× faster than Opaque [84] and comes within 2.6× of Spark SQL. It also outperforms HIRB, a previous oblivious index structure, by over 7×, completing point queries on a 1 million row table with 3.6–9.4ms latency. Our open source implementation of ObliDB is available at `https://github.com/SabaEskandarian/ObliDB`.

Acknowledgments

We would like to thank Ankur Dave and Wenting Zheng for their assistance in reproducing the Opaque benchmarks, as well as Henry Corrigan-Gibbs for many helpful conversations. We would also like to thank the anonymous reviewers for several helpful comments and suggestions. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, SAP, Teradata, and VMware—as well as the NSF under CAREER grant CNS-1651570 and other grants from NSF, the DARPA/ARL SAFEWARE project, the Simons foundation, and a grant from ONR. Any views, opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

APPENDIX

A. SECURITY THEOREM

We model privacy by showing there exists a *simulator* such that for all efficient adversaries $\mathcal{A}$, $\mathcal{A}$ cannot distinguish between a real memory trace from ObliDB and a memory trace from the simulator that is given access to query plans and table sizes. Since the simulator only sees what we intend to leak, the adversary cannot have learned any additional information from interacting with ObliDB. In this model, an (informal) theorem similar to that of Opaque [84] also applies to ObliDB. Let $\mathcal{D}$ be a dataset, $\mathcal{S}$ be its schema, and $\mathcal{Q}$ be a query. Moreover, let $\mathsf{OPT}(\mathcal{D}, \mathcal{Q})$ be the choice of algorithms made by ObliDB's query planner for query $\mathcal{Q}$ on data $\mathcal{D}$ and $\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})$ be the distribution of transcripts of memory accesses outside of oblivious memory made by ObliDB while running query $\mathcal{Q}$ on $\mathcal{D}$. Finally, $|\mathcal{D}|$ denotes the size of $\mathcal{D}$ and $|\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})|$ denotes the sizes of the memory traces of running each operator in $\mathcal{Q}$ on $\mathcal{D}$. Since ObliDB stores intermediate tables encrypted outside of the enclave, this includes intermediate table sizes.

THEOREM 1. *For all $\mathcal{D}, \mathcal{S}, \mathcal{Q}$, and security parameter λ, there is a poly-time simulator SIM such that for all PPT adversaries $\mathcal{A}$,*

$$|Pr[\mathcal{A}(\mathsf{SIM}(|\mathcal{D}|, \mathcal{S}, \mathsf{OPT}(\mathcal{D}, \mathcal{Q}), |\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})|)) = 1] - Pr[\mathcal{A}(\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})) = 1]\,| \leq negl(\lambda).$$

The fact that SIM exists means anything that can be learned by looking at the transcript of ObliDB running can also be learned by looking only at the sizes of the data/queries as well as the table schemas and physical operators chosen by the query planner. The theorem for padding mode replaces the data and trace size with a public parameter indicating the size to which we pad all tables.

To argue that SIM exists, we first argue that each operator output by OPT satisfies our obliviousness property. Next, we argue that the query planner's operations are oblivious with its only leakage being that inherent in the final choice of physical operator. We provide these arguments in Sections 4 and 5. With this, we have all the pieces required to explicitly describe SIM that prints an access pattern transcript distributed indistinguishably from $\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})$ because the trace of query $\mathcal{Q}$ on dataset $\mathcal{D}$ consists exactly of the accesses made by running the planner and then the chosen operator(s).

SIM begins by reading $\mathcal{S}$ and $|\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})|$. It uses this information to simulate the access pattern of one scan over $\mathcal{D}$. This is identical to the access pattern of the query planner. Now SIM reads $\mathsf{OPT}(\mathcal{D}, \mathcal{Q})$ to determine which operator to simulate. Using the provided choice of operator, the schema $\mathcal{S}$, and its knowledge of input and output table sizes gleaned from $|\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})|$, it simulates the access pattern described in the body of the paper for the selected operator on $\mathcal{D}$ (i.e. some number of linear scans or ORAM operations). This completes the simulated output which is distributed indistinguishably from that of $\mathsf{TRACE}(\mathcal{D}, \mathcal{Q})$. The simulator SIM' for padding mode behaves analogously to SIM.

B. ORAM

Oblivious RAM (ORAM), a cryptographic primitive first proposed by Goldreich and Ostrovsky [37], hides access patterns to data in untrusted storage. For our purposes, an ORAM consists of a small trusted *client* which resides inside an enclave and performs reads/writes to untrusted memory accessible by the OS. Merely encrypting data still reveals access patterns to the data being requested or written, which can leak private information about the data [41]. ORAM shuffles the locations of blocks in memory so repeated accesses to the same block and other patterns are hidden from the adversary. Specifically, ORAM guarantees that any two access patterns *of the same length* are computationally indistinguishable.

Implementing ORAM. ObliDB uses the Path ORAM [73], which operates by storing encrypted blocks of memory in a tree structure. Every read or write to a block (reads and writes are indistinguishable) reads a path from the root to a leaf, and then writes the same path again, regardless of where in the path the desired block sits. The contents of every node in the path are decrypted, read, and re-encrypted. To prevent leaking statistical information about repeated accesses to the same address, a block is randomly reassigned to a new part of the tree after each access. This causes ORAM reads and writes to incur an $O(\log N)$ overhead, where N is the ORAM's size in blocks. If the tree lacks space to store some node in its designated place, the node is kept in an off-tree *stash* until it can find space in a future operation. Path ORAM guarantees that the stash stays quite small with overwhelming probability.

Recursive vs Nonrecursive ORAM. One feature of Path ORAM requires further discussion. In order to know which path down the tree to read to find a given block, the ORAM client keeps a *position map* that maps each block of memory to a leaf in the tree that identifies the path where it can be found. Since the size of the position map is a fixed fraction of the size of the raw data, Path ORAM recursively stores the position map in a second ORAM and repeats until the client storage requirement becomes sufficiently small. We call an ORAM with no recursion a *nonrecursive* ORAM and an ORAM that recursively uses a second ORAM a *recursive* ORAM. In practice, because the size of an entry in a position map is many times smaller than a block of data, at most one layer of recursion suffices to store large quantities of data. For example, a 10MB position map in our implementation can support 1.1 million records (regardless of record size), and a 20MB position map can store twice as many records. Adding a second layer of recursion, where each of those 1.1 million records represent another 1.1 million records, comes at an approximately 2× performance overhead but allows the same 10MB position map to support 1.2 *trillion* records.

Segmenting ORAM. In addition to optimizing ORAM to minimize storage costs, we can also optimize to reduce computational costs. One way to do this is to separate one ORAM into multiple smaller ORAMs in a way that the choice of which ORAM is written to by a given operation does not leak any additional information. Although ORAM's computation costs scale logarithmically in the size of a given ORAM, dramatically reducing the size of an ORAM can still have a significant impact on performance. For example, ObliDB uses a separate ORAM for each table because it does not hide which tables a query reads or modifies. This optimization could be taken further by using a separate ORAM for an index structure and the data for each table, or even using a separate ORAM for each level of a B+ tree (where padding would happen on a per-level basis rather than for the whole tree). These optimizations do not compromise obliviousness because the access patterns between levels of a B+ tree in a read, insert, or delete operation, once padded to the worst case scenario, are publicly known. The ORAM only needs to hide *which entry* in a given level is accessed to preserve obliviousness.

10. REFERENCES

[1] Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[2] 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.

[3] M. A. Abdelraheem, T. Andersson, and C. Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. *IACR Cryptology ePrint Archive*, 2017:24, 2017.

[4] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[5] A. Arasu and R. Kaushik. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 26–37, 2014.

[6] ARM TrustZone, 2017. https://www.arm.com/products/security-on-arm/trustzone.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. 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*, pages 1383–1394, 2015.

[8] A. F. Aviram. Interactive B+ tree (C). http://www.amittai.com/prose/bplustree.html, 2016.

[9] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[10] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 205–216, 2011.

[11] J. Bater, X. He, W. Ehrich, A. Machanavajjhala, and J. Rogers. Shrinkwrap: Efficient SQL query processing in differentially private data federations. *PVLDB*, 12(3):307–320, 2018.

[12] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 837–849, 2015.

[13] C. Bing. Atos, IT provider for winter olympics, hacked months before opening ceremony cyberattack, 2018. https://www.cyberscoop.com/atos-olympics-hack-olympic-destroyer-malware-peyongchang/.

[14] A. Bittau, Ú. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnés, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 441–459, 2017.

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

[16] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. *IACR Cryptology ePrint Archive*, 2017:31, 2017.

[17] E. Boyle, K. Chung, and R. Pass. Oblivious parallel RAM and applications. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 175–204, 2016.

[18] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.*, 2017.

[19] B. Butler. NSA spying fiasco sending customers overseas, 2013. https://www.computerworld.com/article/2484894/cloud-computing/nsa-spying-fiasco-sending-customers-overseas.html.

[20] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 668–679, 2015.

[21] A. Chakraborti and R. Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[22] T. H. Chan, K. Chung, and E. Shi. On the depth of oblivious parallel RAM. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 567–597, 2017.

[23] B. Chen, H. Lin, and S. Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 205–234, 2016.

[24] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Stealing Intel secrets from SGX enclaves via speculative execution. *CoRR*, abs/1802.09085, 2016.

[25] M. Costa, L. Esswood, O. Ohrimenko, F. Schuster, and S. Wagh. The pyramid scheme: Oblivious RAM for trusted processors. *CoRR*, abs/1712.07882, 2017.

[26] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[27] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 857–874, 2016.

[28] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 727–743, 2018.

[29] S. Cui, S. Belguith, M. Zhang, M. R. Asghar, and G. Russello. Preserving access pattern privacy in SGX-assisted encrypted search. In *ICCCN 2018*, 2018.

[30] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. N. Garofalakis. 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*, pages 185–198, 2016.

[31] T. T. A. Dinh, P. Saxena, E. Chang, B. C. Ooi, and C. Zhang.

M2R: enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 447–462, 2015.

[32] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems (6th Edition)*. Pearson, 2010.

[33] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. IRON: functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 765–782, 2017.

[34] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi. HardIDX: Practical and secure index with SGX. In *DBSec*, pages 386–408, 2017.

[35] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 172–191, 2017.

[36] M. Giraud, A. Anzala-Yamajako, O. Bernard, and P. Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017.*, pages 200–211, 2017.

[37] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[38] M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, 2011.

[39] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1353–1364, 2016.

[40] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *PoPETs*, 2019(1):172–191, 2019.

[41] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[42] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[43] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *CoRR*, abs/1611.06952, 2016.

[44] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi. 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*, pages 87–101, 2015.

[45] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015.

[46] C. Liu, L. Zhu, M. Wang, and Y. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, 2014.

[47] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. 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*, pages 311–324, 2013.

[48] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: rollback protection for trusted execution. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 1289–1306, 2017.

[49] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy, SP (Oakland)*, 2018.

[50] M. Naveed, S. Kamara, and C. V. Wright. 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*, pages 644–655, 2015.

[51] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal. HOP: Hardware makes obfuscation practical. In *NDSS*, 2017.

[52] K. Nayak and J. Katz. An oblivious parallel RAM with o($\log^2$ N) parallel runtime blowup. *IACR Cryptology ePrint Archive*, 2016:1141, 2016.

[53] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 334–348, 2013.

[54] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1570–1581, New York, NY, USA, 2015. ACM.

[55] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.

[56] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, 2012.

[57] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 157–172, 2014.

[58] N. Porter, J. Garms, and S. Simakov. Introducing Asylo: an open-source framework for confidential computing, 2018. `https://cloudplatform.googleblog.com/2018/05/Introducing-Asylo-an-open-source-framework-for-confidential-computing.html`.

[59] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1341–1352, 2016.

[60] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy, SP (Oakland)*, 2018.

[61] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 431–446, 2015.

[62] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 415–430, 2015.

[63] D. S. Roche, A. J. Aviv, and S. G. Choi. A practical oblivious map data structure with secure deletion and history independence. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 178–197, 2016.

[64] M. Russinovich. Introducing Azure confidential computing, 2017. `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/`.

[65] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 198–217, 2016.

[66] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace : Oblivious memory primitives from Intel SGX. *IACR Cryptology ePrint Archive*, 2017:549, 2017.

[67] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 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*, pages 38–54, 2015.

[68] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-shield: Enabling address space layout randomization for SGX programs. In *NDSS*, 2017.

[69] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[70] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. 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*, pages 317–328, 2016.

[71] R. Sinha and M. Christodorescu. Veritasdb: High throughput key-value store with integrity. *IACR Cryptology ePrint Archive*, 2018:251, 2018.

[72] E. Stefanov and E. Shi. Oblivistore: High performance oblivious distributed cloud data store. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[73] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. 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*, pages 299–310, 2013.

[74] Y. Tang and J. Chen. LPAD: Building secure enclave storage using authenticated log-structured merge trees. *IACR Cryptology ePrint Archive*, 2016:1063, 2018.

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

[76] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also Technical Report Foreshadow-NG: `http://ForeshadowAttack.com`.

[77] D. Vinayagamurthy, A. Gribov, and S. Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *PoPETs*, 2019(3):370–388, 2019.

[78] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. 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*, pages 299–313, 2017.

[79] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 215–226, 2014.

[80] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza. 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*, pages 440–457, 2016.

[81] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell. Privacy-preserving shortest path computation. In *NDSS*, 2016.

[82] Y. Xu, W. Cui, and M. Peinado. 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*, pages 640–656, 2015.

[83] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 707–720, 2016.

[84] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 283–298, 2017.

[85] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque (github repository), 2017. `https://github.com/ucbrise/opaque/tree/c42fe1bb758a93239fae284885c3d64991affddf`.