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

August 16, 2017

Abstract

We present Sloth[1], a database built on Intel's SGX hardware that provides minimal leakage of cryptographically protected data. Sloth achieves comparable performance to prior work while leaking only the structure of queries and the protected data returned from them. In particular, access patterns to data are hidden. Sloth supports a broad range of SQL queries including groupings and joins and makes use of both ORAM-based oblivious indexes and linear search-based data structures. It also provides a range of oblivious algorithms to execute `SELECT` operations with different result sizes.

1 Introduction

The advent of cloud computing has ushered in hopes of a future where data owners can outsource their databases while retaining data security. In recent years, a smorgasbord of solutions to the problems of cryptographically-protected databases and search over encrypted data has explored a large space of tradeoffs between performance and leakage of private queries and data [21], but performant systems which provide the highest levels of security – that of hiding even access patterns to protected data – remain out of reach. Meanwhile, increasing interest in trusted hardware solutions, lent impetus by the appearance

of Intel's SGX [10] has allowed for dramatic speedups in tasks previously requiring heavy and slow cryptographic operations [18, 29]. While SGX alone partially solves the problem of protected databases [20], preventing leakage of data access patterns requires more work. Several prior works [30, 12, 21] mention the possibility of generically using Oblivious RAM (ORAM) on top of SGX to hide these access patterns but point out that, unfortunately, simply running a generic database application over ORAM adds significant slowdowns.

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

We implement a prototype of Sloth and report on its performance, testing it on both fabricated data of up to 500,000 rows and three real-world data sets of various sizes: domestic US flight data [15], consumer complaints to the consumer financial protection bureau [16], and the NASDAQ stock exchange [17]. We compare Sloth to a baseline implementation where a

---

[1]Much like the data in our system, sloths, who are abundantly present in trees in some parts of the world, are difficult to spot as they are well-hidden amongst the trees where they spend most of their lives.

database index is generically modified to provide data access obliviousness via ORAM and show that Sloth outperforms the baseline by approximately 70% on insertions and deletions and by 60% on SELECT operations. We also compare Sloth to prior work and find that name/ performs comparably to the range search scheme of Demertzis et al [12] which does not hide access patterns and ranges from 6.3x slower to 20x faster than Opaque's Oblivious mode, an SGX-based data analytics platform that does hide access patterns, suggesting that Sloth and Opaque are fundamentally well-suited to different use cases. Moreover, we show that the choices of oblivious data structures and algorithms available in Sloth allow for meaningful optimizations in different data and query settings.

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

2 Architecture and Security Overview

This section summarizes the functionality and architecture of Sloth, the threat model for which it is designed, and the security properties it achieves.

2.1 Sloth Overview

Sloth consists of a trusted code base inside an SGX enclave that provides an interface for users to create, modify, and query tables. Sloth supports tables both with and without indexes, called Indexed and Linear tables, respectively. These tables are stored, encrypted, in unprotected memory and are obliviously accessed as needed by the various supported operators. Indexed tables consist of an ORAM with a B+ tree stored inside, whereas Linear tables rely on accessing every block of the underlying data structure to ensure obliviousness.

Sloth supports oblivious versions of the SQL operators SELECT, INSERT, UPDATE, DELETE, GROUP BY and JOIN as well as the aggregates COUNT, SUM, MIN, MAX, and AVG. Each operator is implemented for both Linear and Indexed tables. Additionally, several different algorithms are included for the SELECT operator, each of which performs better for a different output table size. Our SELECT implementation begins by scanning the table being queried to determine which algorithm to use and then executing the appropriate choice for the expected output size.

We assume a secure channel exists through which an outside user can send messages to the enclave (this is fairly straightforward with SGX), but we did not implement this for our tests, as it is not directly related to the functionality provided by Sloth.

2.2 Threat Model

We assume that the SGX platform and its protected memory pages are secure and do not directly handle side-channels known to affect SGX hardware such as page fault timing attacks [42] and branch shadowing [25]. We note, however, that in many cases the portions of our data structures that need to be stored permanently in enclave memory are small enough to fit in processor registers, and it is likely that Sloth can, without heavy modification, provide security against page fault timing attacks. General solutions to protect against such side channels are also compatible with Sloth (see section 8 for an overview).

The threat model we assume for Sloth is a malicious operating system with power to examine untrusted memory and any communication between the processor and memory. Moreover, the operating system can maliciously schedule processes or interrupt the execution of an enclave. Although we give the operating system the power to maliciously alter the execution of the program, we assume that it faithfully responds to requests for data via OCALLs. This assumption could be removed with additional lightweight safeguards, but we do not fully implement these safeguards. We note that it is always possible for a malicious operating system to launch an indefinite denial of service attack against an enclave, but such an attack does not compromise security and is

outside the scope of the security of SGX for our purposes.

2.3 Security Goals

To carefully describe the level of security provided by Sloth, we follow the convention [21] of describing security in terms of the level of leakage in our scheme. Queries in Sloth reach the highest standard for leakage[2]: to leak only "structural" information about data, queries, and responses, i.e., to only reveal information that can be hidden by padding. This includes, for example, the size of tables in the database, the sizes of queries, and the sizes of responses to queries[3]. Hiding this leakage can only be accomplished with padding at a necessarily high performance cost. In total, Sloth only leaks the structure of queries made on data, the structure of the data returned, and the query plan used to service a query. Structural information regarding intermediate tables created during query execution also leaks. Further details regarding how these leakage properties are achieved can be seen in sections 4 and 5, where the leakage of each operator is explained.

Another concern with regards to security is that, however secure the properties of the database management system, the way it is used by an application interacting with it can leak additional information. For example, if a web application makes a second query to a database based on the results of a first query, observing the size of the response to the second query may leak additional information about the first query or its response. This is a direct, if unexpected, consequence of structural leakage, and it falls on application developers to consider performance goals against the ramifications of such leakage in their design process.

---

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

[3]We do not make an effort to hide the number of tables in a database or which table(s) a particular query accesses. Our goals deal only with the security of data within individual tables.

3 Background

In this section we give a basic overview of Intel SGX, ORAM, and B+ trees, the primary tools used in Sloth, providing only sufficient detail for the subsequent sections. For more information on work using these primitives, particularly applications, attacks, and defenses for SGX, see section 8.

3.1 Intel SGX

We will only briefly discuss here the main features of SGX relevant to our work. For more details, consult the SGX tutorial of Costan and Devadas [10] or the Intel's developer reference [1].

SGX provides developers with the abstraction of a secure *enclave* which can verifiably run a trusted code base (TCB) and protects its limited memory from a malicious or compromised operating system. SGX handles the process of entering and exiting an enclave and hiding the activity of the enclave when non-enclave code is being run, albeit imperfectly [25]. Enclave code invariably requires access to operating system resources, so SGX provides an interface between the enclave and the operating system based on *OCALLs* and *ECALLs*. OCALLs are calls made from inside the enclave to the operating system, usually for procedures requiring resources managed by the operating system, such as access to files on disk. ECALLs allow code outside the TCB to call the enclave to execute trusted code.

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

The feature of SGX with which we are most concerned is the protection of memory. SGX provides the developer with approximately 90MB of Enclave Page Cache (EPC), a memory region that is hidden from the OS and cleared whenever execution enters or exits an enclave. This memory can be used to execute trusted code and keep secrets from a malicious operating system who otherwise controls the machine

executing the code.

3.2 ORAM

Oblivious RAM, or ORAM, is a cryptographic primitive first proposed by Goldreich and Ostrovsky [22] that hides access patterns to data in untrusted memory. In the traditional ORAM setting, a small trusted processor uses a larger memory over a bus on which an adversary may examine communications. Merely encrypting the data that travels over the bus still reveals the access patterns to the data being requested and can be used to glean private information about the data or the queries on it [24]. ORAM goes further and shuffles the locations of blocks in memory so repeated accesses and other patterns are hidden from the observing adversary. ORAMs guarantee that any two sets of access patterns of the same length are indistinguishable from each other. More formally, the security of ORAM is defined as follows:

Definition 1 (ORAM Security[39]). *Let $\overrightarrow{y}$:= $((op_M,\ a_M,\ data_M),...,(op_1,\ a_1,\ data_1))$ denote a data request sequence of length M, where each op_i denotes a **read(a_i)** or a **write(a_i, data)** operation. Specifically, a_i denotes the identifier of the block being read or written, and $data_i$ denotes the data being written. Index 1 corresponds to the most recent load/store and index M corresponds to the oldest load/store operation.*

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

1. *For any two data request sequences $\overrightarrow{y}$ and $\overrightarrow{z}$ of the the same length, their access patterns $A(\overrightarrow{y})$ and $A(\overrightarrow{z})$ are computationally indistinguishable by anyone but the client ORAM controller.*

2. *The ORAM construction is correct in the sense that it returns on input $\overrightarrow{y}$ data that is consistent with $\overrightarrow{y}$ with probability $\geq 1 - negl(|\overrightarrow{y}|)$, i.e., the ORAM may fail with probability $negl(|\overrightarrow{y}|)$.*

The scope of the security guarantees provided by ORAM create important consequences for oblivious data structures and algorithms built on top of ORAM for two primary reasons. First, ORAM only makes guarantees of indistinguishability for access patterns of the same length. This means that oblivious algorithms using ORAM must always make the same number of memory accesses or risk leaking access pattern data. Second, the definition of security for ORAM does not address side channels, so implementations using ORAMs must ensure that a program's branching behavior does not leak private information without relying on ORAM.

Although other, older schemes have recently received attention due their practical efficiency in certain practical parameter settings [43], the most efficient ORAM scheme known is the Path ORAM [39]. Path ORAM belongs to a family of schemes known as tree-based ORAMs, which operate by storing the blocks of the oblivious memory in a tree structure. Each block is associated with a leaf in the tree in a position map that guarantees the block will be found somewhere on the path to that leaf. An access to the ORAM involves reading a path down the tree from the root to the leaf corresponding to the desired block. After retrieving the desired block, a second pass is made on the same path where each block is re-encrypted with new randomness and the retrieved block is assigned a new leaf, remaining stored in a small "stash" if the path does not allow space for it to be written back on the path to its new assigned leaf. Although it is not always necessary in practice, the position map holding the assigned leaves for each block of the ORAM can be recursively stored in its own ORAM to reduce the trusted processor memory required by this scheme to a constant.

3.3 B+ Tree

The B+ tree is a generalization of the well-known binary search tree and is a data structure frequently used for database indexes [14, 4]. All data is kept in the leaves of the tree, and each row in the database is associated with a *key*. Each internal node contains a series of ordered *labels* corresponding to potential values of keys in the table, with the number of children varying between fixed minimum and maximum values. There is one child node between every pair

of labels that represents any rows whose key falls between those label values. The algorithm to find a row with a specific key follows the path down the tree to the desired leaf, much like a binary search tree. Each leaf is connnected to the next with a pointer so that it is possible to follow the data sequentially as a linked list once a desired row is found.

Although the process of looking up data in a B+ tree is fairly straightforward, there are a long list of rules involved in maintaining the invariants mentioned above while inserting or deleting rows. As these details are not immediately relevant to the work presented here, we refer the interested reader to standard database texts (e.g. [14]) for details.

4 Oblivious Data Structures

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

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

A Linear type table simply stores rows in a series of adjacent blocks with no additional mechanism to ensure obliviousness of memory accesses. This is a "trivial" ORAM where every read or write to the table must involve accesses to every block of the structure in order to maintain obliviousness of access patterns. As such, operators acting on such a tables, as will be seen in section 5, involve a series of linear scans over the entire data structure. This data structure performs best with small tables, tables where operations will typically require returning large swaths of the table, or aggregates that involve reading most or all of the table regardless of the need for obliviousness.

In contrast to the simplicity of Linear tables, Indexed type tables make use of both an ORAM and a B+ tree in order to provide better performance without losing obliviousness for large data sets. The data structure consists of a nonrecursive Path ORAM that holds a B+ tree where the actual data of the table resides. Although the security properties of ORAM guarantee that two access transcripts of the same length will be indistinguishable from each other, it is important in designing oblivious algorithms for operators over Indexed tables to make sure that the total number of accesses or the timing gaps between accesses do not leak any additional private data. These concerns are addressed for each operator in section 5.

5 Oblivious Operators

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

The following notation will be used in subsequent paragraphs: the table being returned will be referred to as O, and the table being selected from will be referred to as T. The number of rows in O is represented by o, the number of rows in T is N. o' and N'

represent the number of blocks in the data structures holding O and T, respectively.

5.1 Linear Tables

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

- Insertion: the first unused block encountered during the linear scan of the table will have the contents of the inserted row written to it instead of a dummy write.

- Deletion: any row matching the deletion criteria will be marked as unused and overwritten with fake data. Deletions and updates support the same kinds of conditions as selection, so any logical combination of conditions on equality or inequality of entries in a row is acceptable.

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

All of the above operations leak nothing about the parameters to the query being executed or the data being operated on except the sizes of the data structures involved because they consist of one linear scan over a table where each encrypted block is read and then written with a fresh encryption.

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

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

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

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

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

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

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

The copy operation reveals no additional information about T or O because it could be carried

out by a malicious OS with no input from the enclave or a user. The process of clearing unselected rows involves a read followed by a write to each block of the table, so it also reveals no information beyond the size of T. This algorithm, in fact, does not even reveal the size of the output set O because the data structure size is padded up to the size of T.

- *Hash*: In the case that none of the preceding special-case algorithms apply, Sloth uses the following generalization of the continuous strategy. The main idea is that we wish to apply the technique used for continuous data on data that may be arbitrarily spread throughout T, not just in one continuous block. Our solution is to resort to a hashing-based solution. For the ith row in T, if the row is to be included in the output, we write the content of the row to the $h(i)$th positin in O, where h is a hash function (we used the last several bits of the SHA256 hash function).

The algorithm as stated above does not exactly represent how Sloth works because a few changes are needed in order to ensure, first, that hash collisions are handled to ensure correctness, and, second, that obliviousness is maintained in handling any collisions. In order to maintain obliviousness, every real or dummy write to O must involve the same number of accesses to memory. This means that if any write resolves in a collision that must be resolved, every write must make as many memory accesses as in the case of a collision. Following the guidance of Azar et al [5], we hash each row number using two different hash functions (prepending 0 or 1 to the input of the SHA256 hash) and have a fixed-depth list of 5 slots for each position in O. This means that for each block in T, there will be 10 accesses to O, 5 for each of the two hash functions.

The modifications above ensure that data access patterns are fixed regardless of the data in the table and which rows are selected by the query. Since the hash is taken over the index of the row in the data structure and not over the actual contents of a row, there is no possibility that any information about the data itself can be leaked by observing the patterns of accesses as rows are written to O. As such, the only leaked information is still the sizes of T and O.

Aggregates & Group By

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

Groupings are handled similarly to aggregates without groupings, except that an array is kept inside the enclave that keeps track of the aggregate for each group. Sloth, as implemented, supports only a large fixed total number of groups (up to as many as can fit in an enclave), but appendix A discusses potential solutions for unbounded numbers of groups.

Join

The Join functionality for Linear tables is implemented as a variant of the standard hash join algorithm [14]. Although we only implement inner joins for Linear tables, other forms of join can be implemented without any additional technical or security-related obstacles. We will refer to the two tables being joined as T_1 and T_2. The Join proceeds by making a hash table out of as many rows of T_1 as will fit in the enclave and then hashing the variable to be joined from each row of T_2 to check for matches. This process repeats until the end of T_1 is reached. When a match occurs between a row of T_1 and T_2, the joined row is written to an ORAM, and if a match does not occur, the ORAM receives a dummy write in order to conceal whether or not there has been a match. The ORAM is important for security in order to hide the rate at which rows are added to the returned table. Once the Join process is completed, the contents of the ORAM are copied to a Linear table. This final copy step reveals only the size of the returned table O, and because of the obliviousness property of the ORAM, the join itself also only reveals the sizes of T_1 and T_2.

7

5.2 Indexed Tables

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

There are a few other differences between the behavior of Sloth on Linear and Indexed tables that largely result from design and implementation decisions. Every query in Sloth results in the generation of a Linear scan table with the response, so responses to queries on Indexed tables still appear in Linear tables. Deletions in Indexed tables are designed to find one row matching the deletion criteria and delete that, whereas deletions for Linear tables delete all rows matching the deletion criteria since performance for deleting one row or deleting all matching rows does not differ in the Linear table regime. The Large strategy for selection is not implemented for Indexed tables because the strategy of copying the whole table is not as applicable where a query is aimed at a small fraction of the table and the data are not stored in consecutive blocks but across multiple nested tree-based data structures. Finally, Joins are implemented differently for Indexed tables. Whereas we implemented `INNER JOIN` functionality for Linear tables, Indexed tables only support `LEFT JOIN`.

Since the rows of an Indexed table are always sorted by the index column in the leaves of the B+ tree, it is possible to efficiently sort-merge join two tables with the same index [14]. T_1 and T_2 are scanned at the same time, and any matching rows are placed in an output ORAM, just as for Linear tables. More specifically, at each step, the next row of each of T_1 and T_2 is read. If the rows match, the pointer on the right table advances and there is a write to the ORAM, and if they do not match, a dummy write takes place and the pointer on the table with the lesser value advances. This process proceeds until pointers reach the end of both tables. Obliviousness holds because each step of the algorithm consists of exactly one read to each of T_1 and T_2 and one write to an ORAM, and the total number of steps is only a function of the sizes of the three data structures involved.

6 Implementation

We implemented and evaluated Sloth on an Intel Nuc box with an 1.9 GHz Intel Core i5-6260U Dual-Core processor and 32GB of RAM running Ubuntu 16.04.2 and the SGX Linux SDK version 1.8 [1]. Our implementation includes the linear scan and oblivious B+ index from section 4 as well as most of the oblivious operator algorithms described in section 5, with the exception of those stated there to be unimplemented. Our final implementation consists of 11,000 lines of code and builds upon the Remote Attestation sample code provided with the SDK and the B+ tree implementation of [4], the latter of which was heavily edited in order to support duplicate labels and the dynamic memory abstraction we built on top of ORAM.

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

There are many parameters in Sloth that, set appropriately, could improve performance that we made no effort to optimize. Most importantly, we set the bucket size of the ORAM to 4 and used a binary tree for the tree structure of the PATH ORAM. We also set the maximum branching factor of the B+ tree to 20. We set the number of rows that can fit in the enclave for the "Small" selection strategy at 5000, a number that we felt would allow for reasonably-sized rows without overflowing the memory available to the enclave.

7 Evaluation

Sloth can handle real-world data sets with reasonable performance comparable to other solutions with more permissive leakage functions. We extensively evaluate Sloth on fabricated data with table sizes of up to 500,000 rows (results in the tables below are for tables of 100,000 rows) as well as three real data sets. NASDAQ [17] consists of the last sale price and other related information for all approximately 3200 stocks on the NASDAQ stock exchange. CFPB [16] contains individual complaints on financial products and services sent to the US consumer financial protection bureau and contains about 106,000 rows. Finally, FLIGHTS [15], a subset of the flight dataset used to evaluate Splinter [40], contains data on arrival/departure points, flight numbers, ticket prices, and delays for 250,000 US domestic flights. A summary of queries executed on real-world data is shown in figure 1. We also compare Sloth to prior work and show that it performs comparably or better.

We will through our evaluation prove the following two claims:

1. Choices of data structures and algorithms offered by Sloth allow for meaningful optimizations in handling queries without increasing leakage.

2. Sloth performs better than a baseline solution that generically uses ORAM and comparably or better than prior work with similar or more permissive leakage.

7.1 Effectiveness of Low-Leakage Optimization

We wish to show that providing both Indexed and Linear tables and optimizing `SELECT` queries based only on information from structureal leakage allows Sloth to make meaningful performance improvements for diverse data sets and queries. To this end, figure 2 compares the performance of Linear and Indexed tables on various `SELECT` algorithms as well as `GROUP BY`, `INSERT`, `DELETE`, and `UPDATE` queries. Linear scans perform better as the amount of data retrieved from a table increases since the cost of the scan is amortized over more rows, but smaller queries perform significantly better using an index. Observe, moreover, in figure 1, that queries perform better on Linear tables when the tables are smaller or involves aggregates over a whole table, but Indexed tables perform better when table sizes become large. This is not a surprising conclusion to draw, but it contrasts with claims in prior work [32] that claims linear scans perform better than ORAM for oblivious memory access. In general, Indexed `INSERT`, `DELETE`, and `UPDATE` queries significantly outperform Linear tables.

Figure 3 demonstrates the effectiveness of Sloth's choice of `SELECT` algorithms, comparing our various algorithms on queries that retrieve 5% and 95% percent of a table. Although the "Hash" algorithm

Data Set	Query	Time (Linear)	Time (Index)
NASDAQ	`SELECT Symbol FROM NASDAQ WHERE Sector="Technology" AND Last_Sale>100 AND Last_Sale<200`	.044	.088
NASDAQ	`SELECT AVG(Last_Sale) FROM NASDAQ GROUP BY Sector`	.022	1.26
CFPB	`SELECT COUNT(*) FROM CFPB WHERE Bank="Bank of America" AND (Product="Credit Card" OR Product="Mortgage") AND Timely_Response="No" AND Date_Received>=2013-05-01 AND Date_Received<=2013-05-31`	.683	1.95
CFPB	`SELECT * FROM CFPB WHERE Date_Received>=2013-05-06 AND Date_Received<=2013-05-10`	1.41	1.86
FLIGHTS	`SELECT AVG(Price) FROM FLIGHTS WHERE Destination_Id=12173 GROUP BY Origin_Id`	1.01	.376
FLIGHTS	`SELECT Price FROM FLIGHTS WHERE Destination_Id=12173`	2.00	.760

Figure 1: Queries executed on real data. The NASDAQ queries return technology stocks in the $100-200 price range and the average price of stocks in each sector. The CFPB queries find instances where Bank of America did not respond to complaints about credit cards or mortgages in the month of May 2013 and all complaints in the business week of May 6, 2013. The FLIGHTS queries find the average price to a particular destination from all airports as well as the prices of all flights to that destination.

performs the best asymptotically, the figure demonstrates that knowledge gleaned only from Sloth's intended leakage about the results of a query (whether it is small, large, or a continuous set of rows) suffices to pick an algorithm that will perform dramatically better in practice.

7.2 Comparison to Baseline and Prior Work

Sloth's Indexed tables perform better than a baseline implementation (e.g. what could be achieved with a generic tool for converting legacy applications). Fig-

ure 4 shows comparisons between Sloth and a baseline solution where a database index is generically modified to provide obliviousness of data access patterns via an ORAM. Whereas Sloth's Indexed tables include a number of implementation optimizations to minimize the number of ORAM reads and writes needed in any query, our generic baseline makes an ORAM access every time a non-oblivious database would. The `INSERT` query is graphed separately because it runs much faster than `DELETE` and `UPDATE`. `INSERT` only needs to find one place in the tree to insert a row, whereas `DELETE` and `UPDATE` need to find a position in the leaves of the B+ tree and then scan
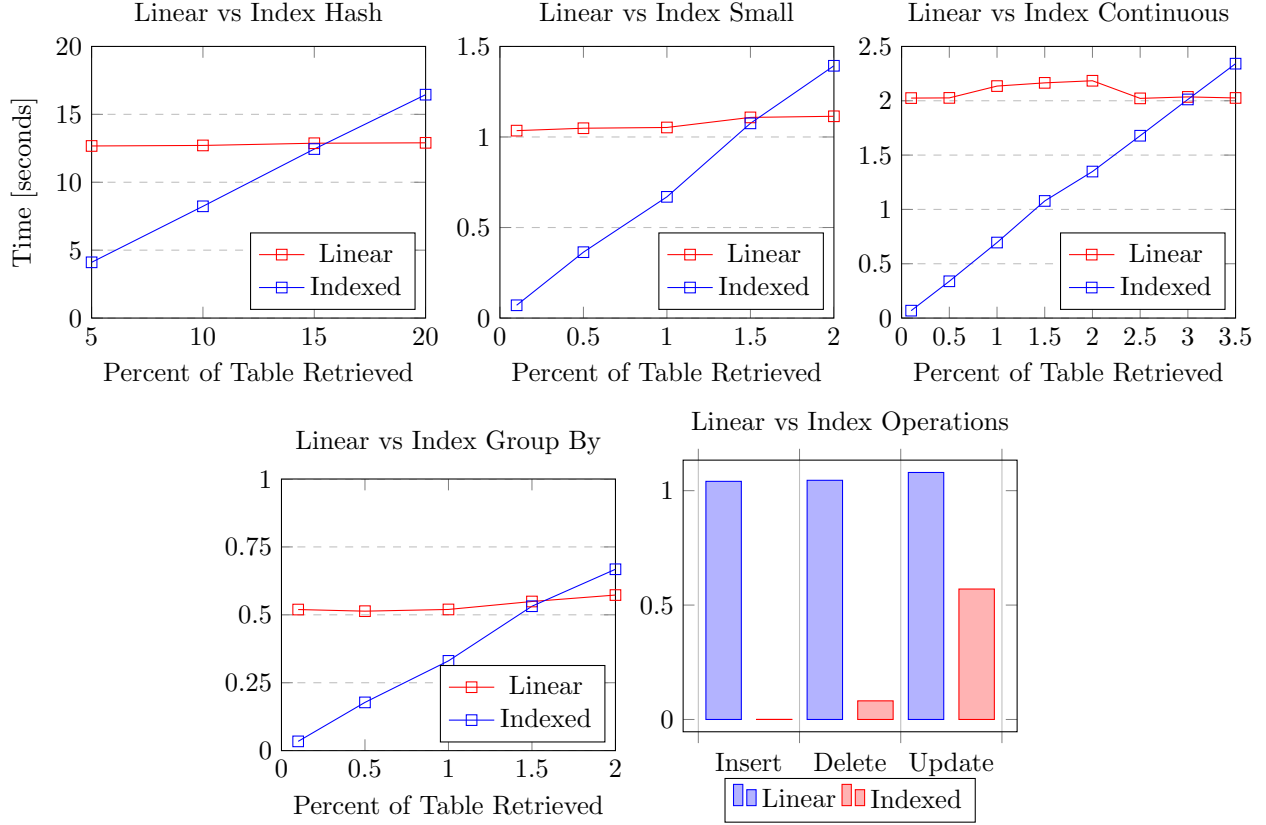
Figure 2: Comparison of Linear and Index versions of various operators. Linear scans do better when most of the data needs to be accessed regardless, but Indexed structures perform far better for small queries. Operations involving modification of the database are far faster for indexed structures.

forward until they find rows to update or delete. On average between the three operations, Sloth outperforms the baseline by approximately 70%.

The comparison of `SELECT` algorithms is complicated by the fact that a non-oblivious `SELECT`, if generically run over an ORAM, still leaks more than structural information when producing the Linear table that is returned to the user for display. We chose the naive "Small" strategy as a baseline because it is the most natural transformation of a non-oblivious `SELECT` to the oblivious setting. Although this baseline performs comparably to other strategies for queries with very small outputs, it rapidly falls behind the "Hash" and "Continuous" strategies

as queries access more of the table. The figure shows this disparity reaching the point where Sloth outperforms the baseline by 60% when a fifth of the table must be retrieved.

Comparison to Prior Work

We compare Sloth to two recent works: the range search scheme of Demertzis et al [12] and Opaque's Oblivious mode [45]. The range search scheme does not hide access patterns to data and lacks functionality for `JOIN`s. As we were not able to compare it directly to Sloth, the best we can do is to examine the graphs in the original paper and observe that Sloth's performance is comparable or slightly better
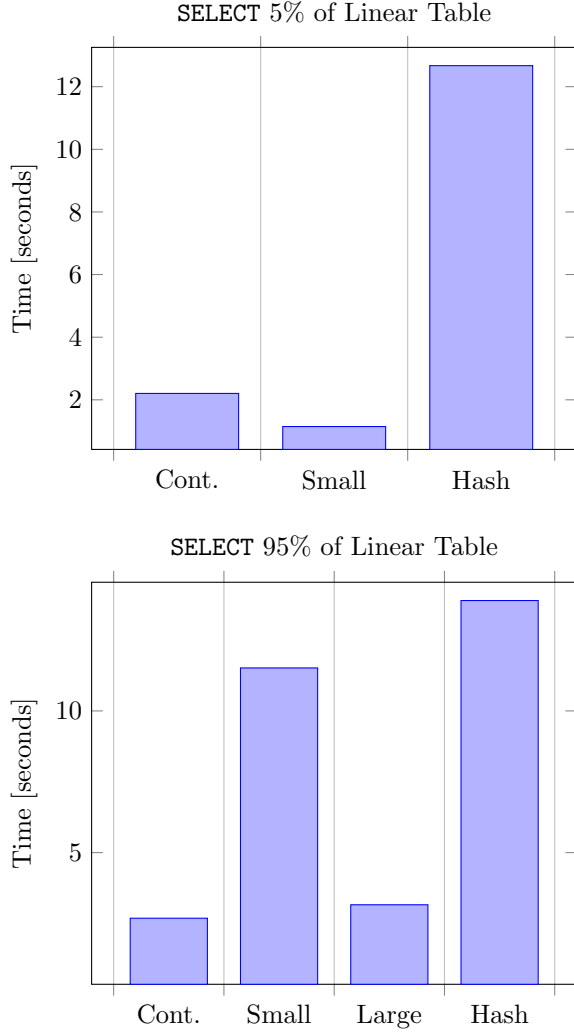
SELECT 5% of Linear Table

SELECT 95% of Linear Table

Figure 3: Comparison of `SELECT` algorithms on Linear table where 5 and 95 percent of a table are retrieved. The Small and Large strategies do well in the 5% and 95% cases, respectively, and the Continuous strategy always performs well when applicable.

than that reported by [12] on a similarly-sized USPS data set with more powerful hardware.

Opaque, on the other hand, can be configured to provide the same security guarantees as Sloth in its OBLIV mode. The difference between Opaque and Sloth is that whereas Opaque is designed with data analytics in the distributed setting in mind and relies heavily on oblivious sorts, Sloth is developed as a standalone database system and makes use of ORAM to provide oblivious indexes. Figure 5 compares the performance of Sloth and Opaque on queries 1 and 3 of the Big Data Benchmark [2] computed over a subset of the original benchmark's dataset (about 350,000 rows in each table). We were unable to compare Sloth and Opaque on queries 2 and 4 of the benchmark because Sloth's high-cardinality aggregate algorithm is unimplemented (see Appendix A) and neither system supports the external scripts needed for query 4. Sloth outperforms Opaque by 20x on query 1 and is 6.3x slower on query 3. The difference highlights the efficacy of Sloth's oblivious indexes, since query 1 is dramatically expedited by having an index that removes the requirement of examining all the data in order to fulfill the query. Query 3, on the other hand, involves an `INNER JOIN` that is executed over a Linear table in Sloth, where Opaque's clever oblivious analytics algorithms give it the upper hand. We conclude from this that Sloth and Opaque are fundamentally well-suited to different kinds of problem domains, since Sloth excels when it can take advantages of its Indexed tables and Opaque performs well on queries that involve scanning over most or all of a table's contents.

8 Related Work

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

Cryptographically-protected database search A testament to the importance of the problem of search over encrypted data in databases lies in the extensive prior work on the subject, summarized and systematized by Fuller et al [21]. Perhaps the most widely known work in this area is CryptDB [31], which implements a tradeoff between security and performance by encrypting each field in a table according to the type of operation expected to be used on the data in that field. Arx [30], a more recent system, keeps all data encrypted at the highest level
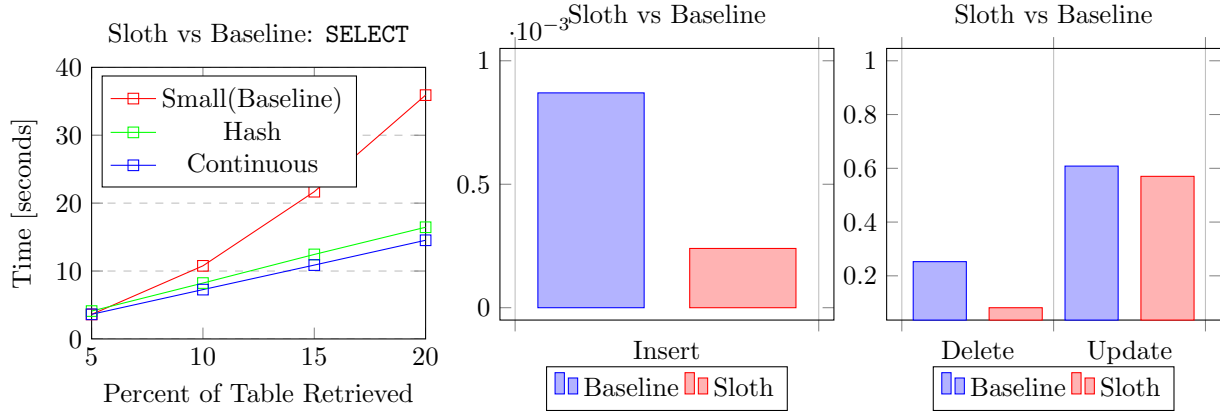
Figure 4: Comparison between Sloth and a baseline solution where a B+ tree is generically ported to use ORAM and naive oblivious algorithms are used. Sloth significantly outperforms the baseline with its Hash and Continuous algorithms as well as its `Insert`, `Delete`, and `Update` operations.
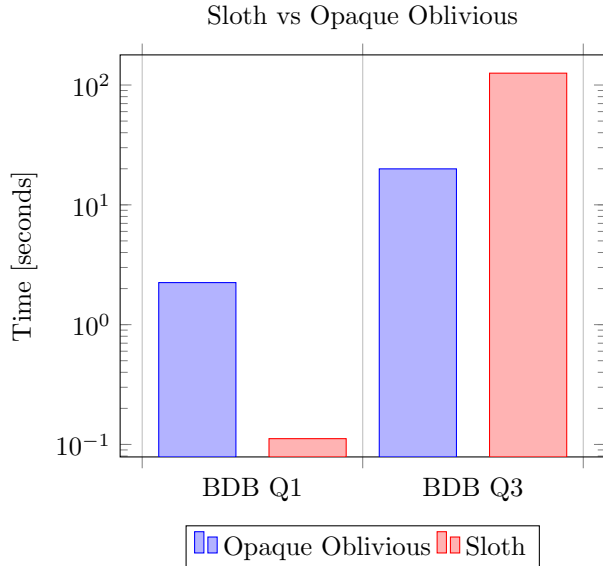

Figure 5: Comparison of Sloth and Opaque Oblivious mode on the Big data benchmark queries 1 and 3. Sloth outperforms Opaque on query 1 where it can leverage that advantage of its oblivious Indexed tables, but Opaque's custom-made oblivious analytics algorithms outperform Sloth when it uses a Linear table to compute the `INNER JOIN` in query 3.

of security and makes clever use of data structures to allow for efficient operations over data. Another common class of solutions are those which use an inverted index to allow searches on stored encrypted data, as exemplified by Demertzis et al [12]. These schemes rely on searchable symmetric encryption (SSE) as a primitive, a recent example of which is Sophos [7], which boasts forward security in the sense that queries authorized in the past do not leak any new information when additional documents are added to an existing database.

The diversity of security goals and varied use cases for which cryptographically protected databases have been designed have led to a plague of attacks which show that real-world applications of schemes proven secure in theoretical models can in fact leak far more data than would be expected from an initial examination of a system's security properties. Initiated by Islam et al [24] and continuing with improved results such as those of Naveed et al [28] and Cash et al [9] to name only a few, such attacks show that inference from known context of the data used, additional correlated public data, or even just the leakage inherent in a scheme itself, can be used to attack various schemes in ways not anticipated in their original security models. Zhang et al [44] show that even schemes with very little leakage are susceptible to attack. In

hiding even the access patterns to data in our solution, we hope to minimize the extent to which Sloth is vulnerable to such techniques.

Trusted hardware

Trusted hardware can be used to achieve security properties that are difficult, impractical, or potentially impossible with traditional cryptographic assumptions. For example, a number of hardware or hardware/software based solutions exist with the explicit goal of rendering programs' memory traces oblivious [11, 26, 27]. Intel SGX, on which we will focus, has in particular been used to implement practical functional encryption [18] and obfuscation [29], both functionalities which can currently only be constructed using heavy cryptographic machinery.

In recent years, a number of generic tools have been designed to provide legacy applications the heightened security available from SGX. Haven [6] shields execution of legacy programs from a malicious OS. Panoply and SCONE [38, 3] provide SGX-protected Linux operating system and container abstractions. In the distributed setting, Ryoan [23] is a sandbox for computation on secret data.

In addition to general tools, applications to securely conduct data analytics or handle data in the cloud represent a compelling practical use case for SGX hardware. In this vein, many works implement variations of existing tools and services rendered secure via SGX. M2R [13] and VC3 [34] provide MapReduce and cloud data analytics functionalities, respectively, and Opaque [45] provides secure support for Spark SQL. SecureKeeper [8] uses SGX to build a confidential version of Apache's ZooKeeper (`zookeeper.apache.org`). More fundamental primitives for databases and oblivious computation in general are provided by HardIDX [20], a database index in SGX, and ZeroTrace [33] which provides oblivious memory primitives based on ORAM as well as an analysis of parameter optimizations for using an ORAM controller in SGX for data storage. None of the solutions above, with the exception of ZeroTrace, use ORAM to hide memory access patterns. Instead, they either make use of memory-oblivious algorithms suited to the tasks they undertake or remain vulnerable to side-channel attacks targeting memory access patterns.

Trusted hardware assumption like those underlying the use of SGX as a secure platform fundamentally differ from traditional mathematical assumptions in that, whereas the validity of a mathematical assumption cannot be challenged by the vicissitudes of succeeding implementations, attacks, and side-channels, the legitimacy of a hardware assumption relies directly on the ability of a piece of manufactured hardware to repel practical attacks. As such, the SGX literature includes a number of works that aim to reveal practical side channels in the implementation of SGX and develop techniques to obviate the risks presented by each known family of attacks. Xu et al [42] use page faults and other "controlled channel" side channels to extract images and text documents from protected memory, and Lee et al [25] use the fact that SGX does not clear branch history when leaving an enclave to infer details of branches taken in protected code. In the multithreaded regime, Weichbrodt et al [41] compromise security by leveraging synchronization bugs. Defenses against such attacks include the work of Shinde et al [37] and Raccoon [32] which close side channels by making the memory trace of a program oblivious or obfuscated. SGX-Shield [35] enables address space layout randomization for SGX, and, finally, T-SGX [36] protects against side-channel attacks by using another set of hardware features, Transactional Synchronization Extensions (TSX) to close side channels that could otherwise be exploited by a malicious operating system. Sloth can be generically combined with any of these solutions to provide higher levels of confidence in the security of the enclave.

9 Conclusion

We have presented Sloth, a cryptographically-protected database system based on Intel SGX with little to no leakage beyond structural information about data and responses to queries. We have shown that Sloth handles practical data sets with acceptable performance comparable to other existing solutions with more permissive leakage functions. It is our hope that solutions like Sloth based on SGX

and other techniques that leverage hardware-based advantages can enable rapid advances in the performance and security of solutions to difficult problems related to private databases and search over encrypted data.

Acknowledgements

References

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

[2] Berkeley AMPlab, University of California. Big data benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 689–703, 2016.

[4] Amittai F. Aviram. Interactive b+ tree (c). `http://www.amittai.com/prose/bplustree.html`.

[5] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[6] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, 2015.

[7] Raphael 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.

[8] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R. Pietzuch, and Rüdiger Kapitza. Securekeeper: Confidential zookeeper using intel SGX. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 14, 2016.

[9] David Cash, Paul Grubbs, Jason Perry, and Thomas 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.

[10] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[11] Victor Costan, Ilia A. Lebedev, and Srinivas 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.

[12] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos 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.

[13] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang 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.

15

[14] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (6th Edition)*. Pearson, 2010.

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

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

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

[18] Ben A. Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using intel sgx. *IACR Cryptology ePrint Archive*, 2016.

[19] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frederic Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Conference on Analysis of Algorithms, AofA07*, 2007.

[20] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure index with SGX. In *Data and Applications Security and Privacy XXXI - 31st Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017, Proceedings*, pages 386–408, 2017.

[21] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert 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.

[22] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[23] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 533–549, 2016.

[24] Mohammad Saiful Islam, Mehmet Kuzu, and Murat 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.

[25] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *CoRR*, abs/1611.06952, 2016.

[26] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine 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.

[27] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn 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.

[28] Muhammad Naveed, Seny Kamara, and Charles 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.

[29] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. Hop: Hardware makes obfuscation practical. In *NDSS*.

[30] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.

[31] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, 2012.

[32] Ashay Rane, Calvin Lin, and Mohit 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.

[33] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. *IACR Cryptology ePrint Archive*, 2017:549, 2017.

[34] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark 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.

[35] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*.

[36] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*.

[37] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek 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.

[38] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxeena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*.

[39] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas 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.

[40] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei 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.

[41] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger 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.

[42] Yuanzhong Xu, Weidong Cui, and Marcus 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.

[43] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multiparty computation. In *IEEE Symposium on Se-*

curity and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016, pages 218–234, 2016.

[44] Yupeng Zhang, Jonathan Katz, and Charalampos 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.

[45] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 283–298, 2017.

A Support for Unbounded Numbers of Groups

As mentioned in section 5, our implementation of Sloth supports a large but fixed maximum number of groups. Here we mention ideas that could be implemented to increase this limit to support arbitrary high-cardinality aggregation.

First, if the number of groups is larger than can be kept in the enclave at once, it becomes important to be able to count the total number on a first pass. This can be accomplished with the HyperLogLog algorithm [19]. Once this is done, it is possible to make multiple passes over the table being selected from, each time calculating the aggregate for as many groups as possible. In order to make sure each group is aggregated only once, each pass will cover successive groups in alphabetical order and the enclave will store the name of the last alphabetical group to have been aggregated. To make sure each pass only aggregates groups in alphabetical order, the algorithm will hae to check on each row whether the group to which the row belongs is alphabetically prior to any group permanently being aggregated. If so, that row is given a place in the enclave and the alphabetically last group under consideration is thrown out to be re-examined on a future pass.

If the number of groups were large enough that the above solution becomes impractical because it requires a number of passes linear in the number of groups, another solution would be to, after completing the HyperLogLog scan, create a copy of the table T called T' and sort it obliviously by group. Then, after creating O of size o as determined by the HyperLogLog algorithm, one additional scan over the sorted table could be done, with the aggregate value for each group being calculated as that group is reached in T'.