Opaque: An Oblivious and Encrypted Distributed Analytics Platform

Wenting Zheng, Ankur Dave, Jethro G. Beekman,
Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica
UC Berkeley

Abstract

Many systems run rich analytics on sensitive data in the cloud, but are prone to data breaches. Hardware enclaves promise data confidentiality and secure execution of arbitrary computation, yet still suffer from *access pattern leakage*. We propose Opaque, a distributed data analytics platform supporting a wide range of queries while providing strong security guarantees. Opaque introduces new distributed oblivious relational operators that hide access patterns, and new query planning techniques to optimize these new operators. Opaque is implemented on Spark SQL with few changes to the underlying system. Opaque provides data encryption, authentication and computation verification with a performance ranging from 52% faster to 3.3x slower as compared to vanilla Spark SQL; obliviousness comes with a 1.6–46x overhead. Opaque provides an improvement of *three orders of magnitude* over state-of-the-art oblivious protocols, and our query optimization techniques improve performance by 2–5x.

1 Introduction

Cloud-based big data platforms collect and analyze vast amounts of sensitive data such as user information (emails, social interactions, shopping history), medical data, and financial data. These systems extract value out of this data through advanced SQL [4], machine learning [25, 15], or graph analytics [14] queries. However, these information-rich systems are also valuable targets for attacks [16, 32].

Ideally, we want to both protect data confidentiality and maintain its value by supporting the existing rich stack of analytics tools. Recent innovation in trusted hardware enclaves (such as Intel SGX [24] and AMD Memory Encryption [19]) promise support for arbitrary computation [6, 34] at processor speeds while protecting the data.

Unfortunately, enclaves still suffer from an important attack vector: *access pattern leakage* [41, 28]. Such leakage occurs at the *memory level* and the *network level*. Memory-level access pattern leakage happens when a compromised OS is able to infer information about the encrypted data by monitoring an application's page accesses. Previous work [41] has shown that an attacker can extract hundreds of kilobytes of data from confidential documents in a spellcheck application, as well as discernible outlines of jpeg images from an image processing application running inside the enclave. Network-level access pattern leakage occurs in the distributed setting because tasks (e.g., sorting or hash-partitioning) can produce network traffic that reveals information about the encrypted data (e.g., key skew), even if the messages sent over the network are encrypted. For example, Ohrimenko et al [28] showed that an attacker who observes the metadata of network messages, such as source and destination (but not their content), in a MapReduce computation can identify the age group, marital status, and place of birth for some rows in a census database. Therefore, to truly secure the data, the computation should be *oblivious*: i.e., it should not leak any access patterns.

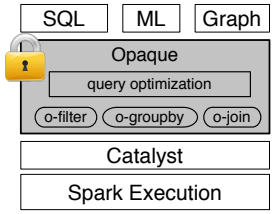


Figure 1: Opaque efficiently executes a wide range of distributed data analytics tasks by introducing SGX-enabled oblivious relational operators that mask data access patterns and new query optimization techniques to reduce performance overhead.

In this paper, we introduce Opaque[1], an oblivious distributed data analytics platform. Utilizing Intel SGX hardware enclaves, Opaque provides strong security guarantees including computation integrity and obliviousness.

One key question when implementing the oblivious functionality is: at what layer in the software stack should we implement it? Implementing at the application layer will likely result in application-specific solutions that are not widely applicable. Implementing at the execution layer, while very general, provides us with little semantics about an application beyond the execution graph and significantly reduces our ability to optimize the implementation. Thus, neither of these two natural approaches appears satisfactory.

Fortunately, recent developments and trends in big data processing frameworks provide us with a compelling opportunity: the *query optimization layer*. Previous work has shown that the relational model can express a wide

[1]The name "Opaque" stands for Oblivious Platform for Analytic QUEries, as well as opacity, hiding sensitive information.

variety of big data workloads, including complex graph analytics [14] and machine learning [17]. We chose to implement Opaque at this layer. While the techniques we present in this paper are general, we instantiate them using Apache Spark [4] by layering Opaque on top of Catalyst, the Spark SQL query optimizer (see Fig. 1). Our design requires no changes to Spark's libraries and requires minimal extensions to Catalyst.

The main challenge we faced in designing Opaque is the question of how to *efficiently* provide access pattern protection. It has long been known in the literature that such protection brings high overheads. For example, the state-of-the-art framework for oblivious computation, ObliVM [22], has an overhead of 9.3×10^6x and is not designed for distributed workloads. Even GraphSC [27], a special-purpose platform for oblivious parallel graph computation, reports a 10^5x slowdown.

To address this challenge, we propose a two-part solution. First, we introduce a set of new distributed relational operators that protect against both memory and network access pattern leakage at the same time. These include operators for joins and group-by aggregates. The contribution of these relational operators is to achieve obliviousness in a *distributed and parallel* setting. One recurring challenge here is to handle boundary conditions (when a value that repeats in rows spans multiple machines) in a way that is efficient and does not leak access patterns. These operators also come with computation integrity guarantees, called *self-verifying computation*, preventing an attacker from affecting the computation result.

Second, we provide novel query planning techniques, both rule-based and cost-based, to further improve the performance of oblivious computation.

- *Rule-based optimization.* Oblivious SQL operators in Opaque consist of fine-grained oblivious computation blocks called Opaque operators. We observe that by taking a global view across these Opaque operators and applying Opaque-specific rules, some operators can be combined or removed while preserving security.
- *Cost-based optimization.* We develop a cost model for oblivious operators that lets us evaluate the cost of a physical plan. This model introduces security as a new dimension to query optimization. We show that it is possible to achieve significant performance gains by using join reordering to minimize the number of oblivious operators. One key aspect used by our cost model is that *not all* tables in a database are sensitive: some contain public information. Hence, we can query such tables using non-oblivious operators to improve performance. Opaque allows database administrators to specify which tables are sensitive. However, sensitive tables can be related with seemingly insensitive tables. To protect the sensitive tables in this case, Opaque leverages a technique in the database literature called inference detection [18, 9] to propagate sensitivity through tables based on their schema information. Additionally, Opaque propagates operator sensitivity as well for all operators that touch sensitive tables.

We implemented Opaque using Intel SGX on top of Spark SQL with minimal modifications to Spark SQL. Opaque can be run in three modes: in *encryption mode*, Opaque provides data encryption and authentication as well as guarantees the correct execution of the computation; in *oblivious mode*, Opaque additionally provides oblivious execution that protects against access pattern leakage; in *oblivious pad mode*, Opaque improves on the oblivious mode by preventing size leakage.

We evaluate Opaque on three types of workloads: SQL, machine learning, and graph analytics. To evaluate SQL, we utilize the Big Data Benchmark [1]. We also evaluated Opaque on least squares regression and PageRank. In a 5-node cluster of SGX machines, encryption mode's performance is competitive with the baseline (unencrypted and non-oblivious): it ranges from being 52% faster to 3.3x slower. The performance gains are due to C++ execution in the enclave versus the JVM in untrusted mode (for vanilla Spark SQL). Oblivious mode slows down the baseline by 1.6–46x. Much of the oblivious costs are due to the fact that Intel SGX is not set up for big data analytics processing; future architectures [8, 21, 35] providing larger and oblivious enclave memory will reduce this cost significantly. We compare Opaque with GraphSC [27], a state-of-the-art oblivious graph processing system, by evaluating both systems on PageRank. Opaque is able to achieve three orders of magnitude (2300x) of performance gain, while also providing general SQL functionality. Finally, while obliviousness is fundamentally costly, we show that our new query optimization techniques achieve a performance gain of 2–5x.

2 Background

Opaque combines advances in secure enclaves with the Spark SQL distributed relational dataflow system. Here we briefly describe these two technologies, as well as exemplify an access pattern leakage attack.

2.1 Hardware Enclaves

Secure enclaves are a recent advance in computer processor technology providing three main security properties: fully isolated execution, sealing, and remote attestation. The exact implementation details of these properties vary by platform (e.g. Intel SGX [24] or AMD Memory Encryption [19]), but the general concepts are the same. Our design builds on the general notion of an enclave, which has several properties. First, *isolated execution* of an enclave process restricts access to a subset of memory such that only that particular enclave can access it. No

other process on the same processor, not even the OS, hypervisor, or system management module, can access that memory. Second, *sealing* enables encrypting and authenticating the enclave's data such that no process other than the exact same enclave can decrypt or modify it (undetectably). This enables other parties, such as the operating system, to store information on behalf of the enclave. Third, *remote attestation* is the ability to prove that the desired code is indeed running securely and unmodified within the enclave of a particular device.

2.2 Access pattern leakage attacks

To understand access pattern leakage concretely, consider an example query in the medical setting:

```
SELECT COUNT(*) FROM patient WHERE age > 30
GROUP BY disease
```

The "group by" operation commonly uses hash bucketing: each machine iterates through its records and assigns each record to a bucket. The records are then shuffled over the network so that records within the same bucket are sent to the same machine. For simplicity, assume each bucket is assigned to a separate machine. By watching network packets, the attacker sees the number of items sent to each machine. Combined with public knowledge about disease likelihood, the attacker infers each bucket's disease type.

Moreover, the attacker can learn the disease type for a *specific database record*, as follows. By observing page access patterns, the attacker can track a specific record's bucket assignment. If the bucket's disease type is known, then the record's disease type is also known. A combination of page-based access patterns and network-level access patterns thus gives attackers a powerful tool to gain information about encrypted data.

2.3 Spark background

We implemented Opaque on top of Spark SQL [42, 4], a popular cluster computing framework, and we use Spark terminology in our design for concreteness. We emphasize that the design of Opaque is not tied to Spark or Spark SQL: the oblivious operators and query planning techniques are applicable to other relational frameworks.

The design of Spark SQL [42, 4] is built around two components: master and workers. The user interacts with the master which is often running with the workers in the cloud. When a user issues a query to Spark SQL, the command is sent to the master which constructs and optimizes a physical query plan in the form of a DAG (directed acyclic graph) whose nodes are tasks and whose edges indicate data flow. The conversion of the SQL query into a physical query plan is mediated by the Catalyst query optimizer.

3 Overview

3.1 Threat model and assumptions

We assume a powerful adversary who controls the cloud provider's software stack. As a result, the adversary can observe and modify the network traffic between different nodes in the cloud as well as between the cloud and the client. The attacker may gain root access to the operating system, modify data or communications that are not inside a secure enclave, and observe the content and order of memory accesses by an enclave to untrusted memory (i.e., memory that is not part of a secure enclave). In particular, the adversary may perform a *rollback attack*, in which it restores sealed data to a previous state.

We assume the adversary cannot compromise the trusted hardware, relevant enclave keys, or client software. In particular, the attacker cannot issue queries or change server-side data through the client. Denial-of-service attacks are out of scope for this paper. A cloud provider may destroy all customer data or deny or delay access to the service, but this would not be in the provider's interest. Customers also have the option to choose a different provider if necessary. Side-channel attacks based on power analysis or timing attacks (including those that measure the time spent in the enclave or the time when queries arrive) are also out of scope.

We assume that accesses to the source code of Opaque that runs in the enclave are oblivious. This can be achieved either by making accesses oblivious using tools such as GhostRider [21], or by using an enclave architecture that provides a pool of oblivious memory [8, 21, 35]; the latter need only provide a small amount of memory because the relevant Opaque source code is $\approx$ 1.4MB.

3.2 Opaque's architecture

Figure 2 shows Opaque's architecture. Opaque does not change the layout of Spark and Spark SQL, except for one aspect. Opaque moves the query planner to the client side because a malicious cloud controlling the query planner can result in incorrect job execution. However, we keep the scheduler on the server side, where it runs in the untrusted domain. We augment Opaque with a computation verification mechanism (§4.2) to prevent an attacker from corrupting the computation results.

The Catalyst planner resides in the job driver and is extended with Opaque optimization rules. Given a job, the job driver outputs a task DAG and a unique job identifier JID for this job. For example, the query from §2.2 translates to the DAG shown in Fig. 3. The job driver annotates each edge with an ID, e.g., E1, and each node with a task ID, e.g., task 4. The input data is split in partitions, each having its own identifier.

Oblivious memory parameter. As discussed, the current Intel SGX architecture leaks memory access patterns

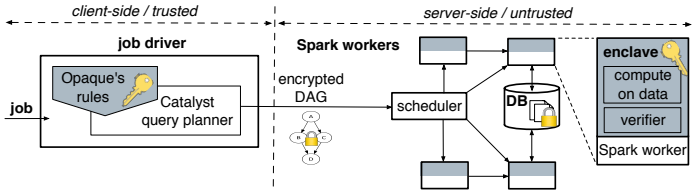


Figure 2: Opaque's architecture overview.

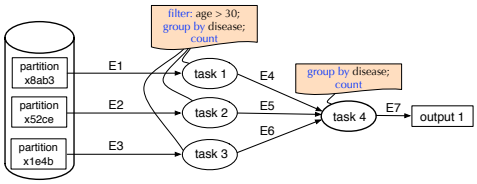


Figure 3: Example task DAG.

both when accessing the enclave's memory (EPC) and the rest of main memory. Nevertheless, recent work, such as Sanctum [8], GhostRider [21], and T-SGX [35], proposes enclave designs that protect against access patterns to the EPC. Hence, such systems yield a pool of oblivious memory, which can be used as a cache to speed up oblivious computations. Since the size of the oblivious pool depends on the architecture used, we parameterize Opaque with a variable specifying the size of the oblivious memory. This parameter can range from as small as the registers (plus Opaque's enclave code size) to as large as the entire EPC [8, 35] or main memory. Bigger oblivious memory allows faster oblivious execution in Opaque. In all cases, Opaque provides oblivious accesses to the non-oblivious part of the EPC, to the rest of RAM, and over the network.

3.3 Security guarantees

Encryption mode. In encryption mode, Opaque provides data encryption and authentication guarantees. Opaque's self-verifying integrity protocol (§4.2) guarantees that, if the client verifies the received result of the computation successfully, then the result is correct, i.e., not affected by a malicious attacker. The proof of security for the self-verifying integrity protocol is rather straightforward, and similar to the proof for VC3 [34].

Oblivious modes. In the two oblivious modes, Opaque provides the strong guarantee of oblivious execution with respect to memory, disk, and network accesses for every *sensitive SQL operator*. As explained in §6.3, these are operators taking as input at least one sensitive table or intermediate results from a set of operators involving at least one sensitive table. Opaque does not hide the computation/queries run at the server or data sizes, but it protects the data content. In oblivious mode, the attacker learns the size of each input and output to a SQL operator and the query plan chosen by Catalyst, which might leak some statistical information. The oblivious pad mode, explained in §5.3, hides even this information by pushing up all filters and padding the final output to a public upper bound, in exchange for more performance overhead. We formalize and prove our obliviousness guarantees in the extended version of this paper, and present only the statement of the guarantees here.

Consider oblivious mode. The standard way to formalize that a system hides access patterns is to exhibit a simulator that takes as input a query plan and data sizes but *not the data content*, yet is able to produce the same trace of memory and network accesses as the system. Intuitively, since the simulator did not take the data as input, it means that the accesses of the system do not depend on the data content. Whatever the simulator takes as input is an upper bound on what the system leaks.

To specify the leakage of Opaque, consider the following (informal) notation. Let $\mathcal{D}$ be a dataset and Q a query. Let $\mathsf{Size}(\mathcal{D})$ be the sizing information of $\mathcal{D}$, which includes the size of each table, row, column, attribute, the number of rows, the number of columns, but does not include the value of each attribute. Let S be the schema information, which includes table and column names in $\mathcal{D}$, as well as which tables are sensitive. Opaque can easily hide table and column names via encryption. The sensitive tables include those marked by the administrator, as well as those marked by Opaque after sensitivity propagation (§6.3). Let $\mathsf{IOSize}(\mathcal{D},Q)$ be the input/output size of each SQL operator in Q when run on $\mathcal{D}$. We define $P = \mathsf{OpaquePlan}(\mathcal{D},Q)$ to be the physical plan generated by Opaque. We define Trace to be the trace of memory accesses and network traffic patterns (the source, destination, execution stage, and size of each message) for sensitive operators.

Theorem 1. *For all $\mathcal{D}$, S, where $\mathcal{D}$ is a dataset and S is its schema, and for each query Q, there exists a polynomial-time simulator* Sim *such that, for* $P = \mathsf{OpaquePlan}(\mathcal{D},Q)$,

$$\mathsf{Sim}(\mathsf{Size}(\mathcal{D}),S,\mathsf{IOSize}(\mathcal{D},Q),P) = \mathsf{Trace}(\mathcal{D},P).$$

The existence of Sim demonstrates that access patterns of the execution are oblivious, and that the attacker does not learn the data content $\mathcal{D}$ beyond sizing information and the query plan. The fact that the planner chose a certain query plan over other possible plans for the same query might leak some information about the statistics on the data maintained by the planner. Nevertheless, the planner maintains only a small amount of such statistics that contain much less information than the actual data content. Further, the attacker does not see these statistics directly and does not have the power to change data or queries and observe changes to the query plan.

Oblivious pad mode's security guarantees are similar to the above, except that the simulator no longer takes as input $\mathsf{IOSize}(\mathcal{D},Q)$, but instead only a public upper bound on the size of a query's final output.

Note that Opaque protects most constants in a query using semantic security: for example it hides the constant in "age ≥ 30", but not in "LIMIT 30".

Coupling oblivious accesses with the fact that the content of every write to memory and every network message is freshly encrypted with semantic security enables Opaque to provide a strong degree of data confidentiality. In particular, Opaque protects against the memory and network access patterns attacks presented in [41] and [28].

4 Opaque's encryption mode

In this section, we describe Opaque's encryption mode, which provides data encryption, authentication and computation integrity.

4.1 Data encryption and authentication

Similar to previous designs [6, 34], Opaque uses remote attestation to ensure that the correct code has been loaded into enclaves. A secure communication channel is then established and used to agree upon a shared secret key k between the client and the enclaves.

All data in an enclave is automatically encrypted by the enclave hardware using the processor key of that enclave. Before communicating with another enclave, an enclave always encrypts its data with AuthEnc using the shared secret key k. AuthEnc encrypts data with AES in GCM mode, a high-speed mode that provides authenticated encryption. In addition to encryption, this mode also produces a 128-bit MAC to be used for checking integrity.

4.2 Self-verifying computation

Ensuring computation integrity is necessary because a malicious OS could drop messages, alter data or computation. We call our integrity checking strategy *self-verifying computation* because the computation verifies itself as it proceeds. The mere fact that the computation finished without aborting means that it was not tampered with.

Let us first discuss how to check that the input data was not corrupted. As in VC3 [34], the identifier of a partition of input data is its MAC. The MAC acts as a *self-certifying* identifier because an attacker cannot produce a different partition content for a given ID. Finally, the job driver computes $C \leftarrow \text{AuthEnc}_k(\text{JID}, \text{DAG}, P_1, \ldots, P_p)$, where $P_1, \ldots, P_p$ indicates the identifiers of the partitions to be taken as input. Every worker node receives C. Opaque's verifier running in the enclave decrypts and checks the authenticity of the DAG in C.

Then, to verify the integrity of the computation, each task needs to check that the computation up to it has proceeded correctly. First, if $E_1, \ldots, E_t$ are edges incoming into task T in the DAG, the verifier checks that it has received authentic input on each edge from the correct previous task and that it has received input for *all* edges. To ensure this invariant, each node producing an output o for an edge E encrypts this output using $\text{AuthEnc}_k(\text{JID}, E, o)$. The receiving node can check the authenticity of this data and that it has received data for *every* edge in the DAG. Second, the node will run the correct task T because the enclave code was set up using remote attestation and task T is integrity-verified in the DAG. Finally, each job ends with the job driver receiving the final result and checking its MAC. The last MAC serves as a proof of correct completion of this task.

This protocol improves over VC3 [34], which requires an extra stage where all workers send their inputs and outputs to a master which checks that they all received complete and correct inputs. Opaque avoids the cost of this extra stage and performs the verification during the computation, resulting in negligible cost.

Rollback attacks. Spark's RDDs combined with our verification method implicitly defend against rollback attacks, because the input to the workers is matched against the expected MACs from the client and afterwards, the computation proceeds deterministically. The computation result is the same even with rollbacks.

4.3 Fault tolerance

In Spark, if the scheduler notices that some machine is slow or unresponsive, it reassigns that task to another machine. Opaque's architecture facilitates this process because the encrypted DAG is *independent* from the workers' physical machines. As a result, the scheduler can live entirely in the untrusted domain, and does not affect Opaque's security if compromised.

5 Oblivious execution

In this section, we describe Opaque's oblivious execution design. We first present two oblivious building blocks, followed by Opaque's oblivious SQL operator designs.

5.1 Oblivious building blocks

Oblivious sorting is central to the design of oblivious SQL operators. Opaque adapts existing oblivious sorting algorithms for both local and distributed sorting, which we now explain.

5.1.1 Intra-machine oblivious sorting

Sorting networks [7] are abstract networks that consist of a set of *comparators* that compare and swap two elements. Elements travel over wires from the input to comparators, where they are sorted and output again over wires. Sorting networks are able to sort any sequence of elements using a fixed set of comparisons.

Denote by OM, the oblivious memory available for query processing, as discussed in §3.2. In the worst case, this is only a part of the registers. If the total size of the data to be sorted on a single machine fits inside the OM, then it is possible to load everything into the OM,

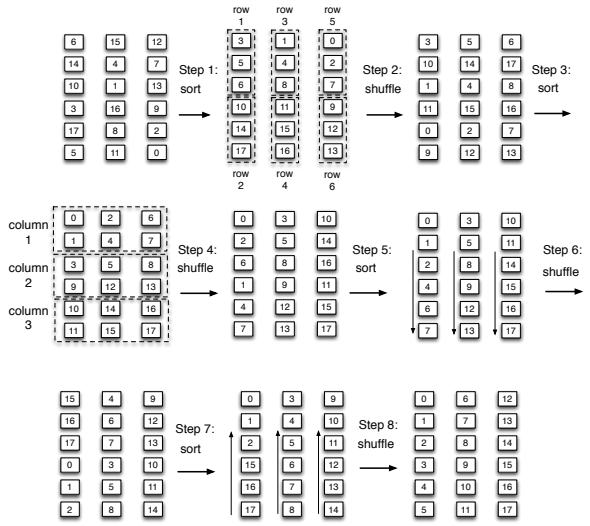


Figure 4: Column sort, used in the distributed setting. Each column represents a single partition, and we assume that each machine only has one partition. The algorithm has eight steps. Steps 1, 3, 5, 7 are sorts, and the rest are shuffle operations.

sort using quicksort, then re-encrypt and write out the result. If the data cannot fit inside the OM, Opaque will first partition the data into blocks. Each block is moved into the OM and sorted using quicksort. We then run a sorting network called *bitonic sort* over the blocks, treating each one as an abstract element in the network. Each comparator operation loads two blocks into the enclave, decrypts, merges, and re-encrypts the blocks. The merge operation only requires a single scan over the blocks.

5.1.2 Inter-machine oblivious sorting

A natural way to adapt the bitonic sorting network in the distributed setting is to treat each machine as an abstract element in the sorting network. We can sort within each machine separately, then run the bitonic sorting network over the machines. However, each level of comparators now corresponds to a network shuffling of data. Given n machines, the sorting network will incur $O(\log^2 n)$ number of shuffles, which is high.

Instead, Opaque uses column sort [20], which sorts the data using a *fixed* number of shuffles (5 in our experiments) by exploiting the fact that a single machine can hold many items. Column sort works as follows: given a sequence of B input items, we split these items into s partitions, where each partition has exactly r items (with padding if necessary). Without loss of generality, we assume that each machine handles one partition. We treat each partition as a column in column sort. The sorting algorithm has 8 steps: the odd-numbered steps are per-column sorts (implemented as intra-machine oblivious sorting), and the even-numbered steps shuffle the data deterministically. Figure 4 gives a visual example of how column sort works. The sorting algorithm has the restriction that $r \geq 2(s-1)^2$, which applies well to our setting because there are many records in a single partition/column.

An important property of column sort is that, as an oblivious operator, it preserves the *balance* of the partitions. This means that after a sort, a partition will have exactly the same number of items as before. Partition balance is required to avoid leaking any information regarding the underlying data's distribution. However, balanced partitioning is incompatible with co-locating all records of a given group. Instead, records with identical grouping attributes may be split across partitions. Operators that consume the output of column sort must therefore be able to transfer information between adjacent partitions *obliviously* and *efficiently*. We address this challenge in our descriptions of the oblivious operators.

5.2 Oblivious operators

In this section, we show how to use the oblivious building blocks to construct oblivious relational algebra operators. The three operators we present are filter, group-by, and join. Opaque uses an existing oblivious filter operator [3], but provides new algorithms for the join and group-by operators, required by the distributed and parallel setting.

In what follows, we focus only on the salient parts of these algorithms. We do not delve into how to make simple structures oblivious like conditionals or increments, which is already known (e.g., [21]).

5.2.1 Oblivious filter

An oblivious filter ensures that the attacker cannot track which encrypted input rows pass the filter. A naïve filter that streams data through the enclave to get rid of unwanted rows will leak which rows have been filtered out because the attacker can keep track of which input resulted in an output. Instead, the filter operator [3] used in Opaque first scans and marks each row with a "0" (record should be kept) or a "1" (record should be filtered), then obliviously sorts all rows with "0" before "1", and lastly, removes the "1" rows.

5.2.2 Oblivious Aggregate

Aggregation queries group items with equal *grouping attributes* and then aggregate them using an aggregation function. For example, for the query in §2.2, the grouping attribute is `disease` and the aggregation function is `count`.

A naïve aggregation implementation leaks information about group sizes (some groups may contain more records than others), as well as the actual mapping from a record to a group. For example, a reduce operation that sends all rows in the same group to a single machine reveals which and how many rows are in the group. Prior work [28] showed that an attacker can identify age group or place of birth from such protocols.

Opaque's oblivious aggregation starts with an oblivious sort on the grouping attributes. Once the sort is complete,

all records that have the same grouping attributes are located next to each other. A single scan might seem sufficient to aggregate and output a value for each group, but this is incorrect. First, the number of groups per machine can leak the number of values in each group. A further challenge (mentioned in §5.1.2) is that a set of rows with the same grouping attributes might span multiple machines, leaking such information. We need to devise a parallel solution because a sequential scan is too slow.

We solve the above problems by designing a distributed group-by operator that reveals neither row-to-group mapping nor the size of each group. The logical unit for this algorithm is a partition, which is assumed to fit on one machine. The intuition for this algorithm is that we want to *simulate* a global sequential scan using per-partition parallel scans. If all records in a group are in one partition, the group will be aggregated immediately. Once the last record in that group has been consumed in the scan, the aggregation result is complete. If records in a group are split across partitions, we want to pass information across partitions efficiently and obliviously so that later partitions have the information they need to finish the aggregation.

High-cardinality aggregation. This aggregation algorithm should be run when the number of groups is large.

Stage 1 [sort]: Obliviously sort all records based on the grouping attributes.

Stages 2–4 are the *boundary processing* stages. These stages solve the problem of a single group being split across multiple machines after column sort. Figure 5 illustrates an example.

Stage 2 [per-partition scan 1]: Each worker scans its partition once to gather some statistics, which include the partition's first and last rows, as well as partial aggregates of the last group in this partition. In Figure 5, each column represents one partition. Each worker calculates statistics including R_i, the partial aggregate. In partition 0, $R_0 = (C, 2)$ is the partial aggregate that corresponds to the last row in that partition, C.

Stage 3 [boundary processing]: All of the statistics from stage 2 are collected into a single partition. The worker assigned this partition will scan all of the statistics and compute one global partial aggregate (GPA) per partition. Each partition's GPA should be given to the next partition.

Figure 5's stage 3 shows an example of how the GPA is computed. The first partition always receives a dummy GPA since it is not preceded by any other partition. Partition $P1$ receives $(C, 2)$ from $P0$. With this information, $P1$ can correctly compute the aggregation result for group C, even though the records are split across $P0$ and $P1$.

Stage 4 [per-partition scan 2]: Each partition receives a GPA, which will be used to produce the final aggregation results. Figure 5's stage 4 shows that $P1$ can aggregate groups C, D and E using $R_1^{'}$. Note that one record needs

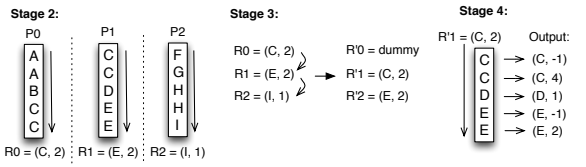


Figure 5: Stages 2 - 4 of oblivious aggregation

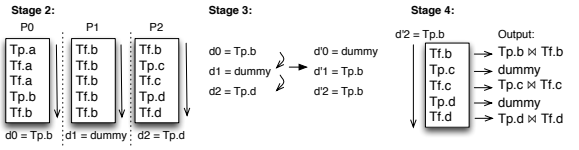


Figure 6: Stages 2–4 of oblivious join.

to be output for every input record, and output rows are marked as dummy if necessary (e.g., returning -1 for the count result).

Stage 5 [sort and filter]: Obliviously sort the dummy records after the real records, and filter out the dummies.

Low-cardinality group-by. If the number of groups is small (e.g., age groups, states), Opaque provides an alternative algorithm that avoids the second oblivious sort, which we describe in the extended version of this paper.

5.2.3 Oblivious sort-merge join

Regular joins leak information about how many and which records are joined together on the same join attributes. For example, a regular primary-foreign key join may sort the two tables separately, maintain a pointer to each table, and merge the two tables together while advancing the pointers. The pointer locations reveal information about how many rows have the same join attributes and which rows are joined together.

We developed an oblivious equi-join algorithm based on the sort-merge join algorithm. While our algorithm presented below focuses on primary-foreign key join, we can also generalize the algorithm to inner equi-join, which we describe in our extended paper. Let T_p be the primary key table, and T_f be the foreign key table.

Stage 1 [union and sort]: We union T_p with T_f, then obliviously sort them together based on the join attributes. We break ties by ordering T_p records before T_f records.

As with oblivious aggregation, stages 2–4 are used to handle the case of a join group (e.g., a set of rows from T_p and T_f that are joined together) that is split across multiple machines. We use Figure 6 to illustrate these three stages.

Stage 2 [per-partition scan 1]: Each partition is scanned once and the last row from T_p in that partition, or a dummy (if there is *no* record from T_p on that machine) is returned. We call this the boundary record.

Figure 6 explains stage 2 with an example, where Tp.x indicates a record from the primary key table with join

attribute x, and Tf.x indicates a record from the foreign key table with join attribute x. In partition $P0$, Tp.b is the last record of T_p in that partition, so the boundary record is set to Tp.b. $P1$ does not contain any row from T_p, so its boundary record is set to a dummy value.

Stage 3 [boundary processing]: In stage 3, we want to generate primary key table records to give back to each data partition so that all of the foreign key table records in each partition (even if the information spans across multiple machines) can be joined with the corresponding primary key record. We do so by first collecting all of the boundary records to one partition. This list is scanned once, and we output a new boundary record for every partition. Each output is set to the value of the most recently encountered *non-dummy* boundary.

For example, Fig. 6's stage 3 shows that three boundary records are collected. Partition 0 will always get a dummy record. Record Tp.b is passed from partition 0 to partitions 1 and 2 because d_1 is a dummy. This ensures that any record from T_f with join attribute b (e.g., the first record of partition 2) will be joined correctly.

Stage 4 [per-partition scan 2]: Stage 4 is similar to a normal sort-merge join, where the worker linearly scans the tables and joins primary key records with the corresponding foreign key records. There are some variations to preserve obliviousness. First, the initial record in the primary key table should come from the boundary record received in stage 3 (except for the first partition). Second, during the single scan, we need to make sure that one record is output for every input record, outputting dummy records as necessary.

Figure 6's stage 4 shows how the algorithm works on partition 2. The boundary record's value is Tp.b, which is successfully joined with the first row of partition 2. Since P_2's second row is a new record from T_p, we change the boundary record to Tp.c, and a dummy is output.

Stage 5 [sort and filter]: Oblivious sort to filter out the dummies.

5.3 Oblivious pad mode

Oblivious execution provides strong security guarantees and prevents access pattern leakage. However, it does not hide the output *size* of each relational operator. This means that in a query with multiple relational operators, the size of each intermediate result is leaked. To solve this problem, Opaque provides a stronger variant of oblivious execution: oblivious with padding.

The idea is to never reduce the output size of a relational operator until the end of the query. This can be easily achieved by using "filter push up." For example, a query that has a join followed by an aggregation will skip stage 5 of the join. After the aggregation, all dummies will be filtered out in a single sort with filter. We also require the user to provide an upper bound on the final result size, and Opaque will pad the final result to this size. In this case, the query plan also no longer depends on data statistics, as we discuss in §6.4.

Note that this mode is more inefficient because Opaque cannot take advantage of selectivity (e.g., of filters), and we provide an evaluation in our extended paper. Therefore, we recommend using padding on extremely sensitive datasets.

6 Query planning

Even with parallelizable oblivious algorithms, obliviousness is still expensive. We now describe Opaque's query planner, which reduces obliviousness overheads by introducing novel techniques that build on rule-based and cost-based optimization, as well as entity-relational modeling. We first formalize a cost model for our oblivious operators to allow a standard query planner to perform basic optimizations on oblivious plans. We then describe several new optimizations specific to Opaque, enabled by a decomposition of oblivious relational operators into lower-level Opaque operators. Finally, we describe a mixed sensitivity setting where a database administrator can designate tables as sensitive. Opaque applies a technique in databases known as second path analysis that uses foreign-key relationships in a data model to identify tables that are *not* sensitive, accounting for inference attacks. We also demonstrate that such sensitivity propagation occurs within a single query plan, allowing us to substantially speed up certain queries using join reordering.

6.1 Cost model

Cost estimation in Opaque differs from that of a traditional SQL database because sorting, the core database operation, is more costly in the oblivious setting than otherwise. Oblivious sorting has very different algorithmic behavior from conventional sorting algorithms because the sequence of comparisons can be constructed based only on the input *size* and not the input data. Therefore, our cost model must accurately model oblivious sorting, which is the dominant cost in our oblivious operators.

Similarly to a conventional sort, the cost of an oblivious sort depends on two factors: the number of input items and the padded record size. Even for datasets that fit in memory, cost modeling for an oblivious sort is similar to that of a traditional external sort because the latency penalty incurred by the enclave for accessing pages outside of the oblivious memory or EPC effectively adds a layer to the memory hierarchy. We therefore use a two-level sorting scheme for oblivious sort, described in §5.1.1, having a runtime complexity of $O(n \log^2 n)$.

We now formalize the cost of oblivious sort and use this to model oblivious join. The costs of other oblivious operators can be similarly modeled.

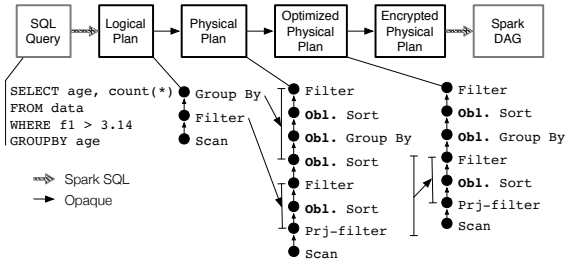


Figure 7: Catalyst oblivious query planning.

Let T be a relation, and r be a padded record. We denote $|T|$ to be the size of the relation T, and $|r|$ to be the size of a padded record. Let |OMem| be the size of the oblivious memory, and K a constant scale factor representing the cost of executing a compare-and-swap on two records. We denote n to be the number of records per block, and B to be the required number of blocks. We can estimate n, B, and the resulting sort cost $C_{\text{o-sort}}$ and join cost $C_{\text{o-join}}$ as follows:

$$n = \frac{|\text{OMem}|}{2\,|R|}, \quad B = |T|/n, \quad C_{\text{o-join}} \approx 2 \cdot C_{\text{o-sort}}$$

$$C_{\text{o-sort}}(|T|,|R|) = \begin{cases} K\,|T|\log|T| & \text{if } |T|\cdot|R| \le |\text{OMem}| \\ K\,[Bn\log n + nB\log B(1+\log B)/2] & \text{otherwise} \end{cases}$$

The number of records n per block follows from the fact that two blocks must fit in oblivious memory at a time for the merge step. The expression for the sort cost follows from the two-level sorting scheme. If the input fits inside the oblivious memory, we bypass the sorting network and instead use quicksort within this memory, so the estimated cost is simply the cost of quicksort. Otherwise, we sort each block individually using quicksort, run a sorting network on the set of blocks and merge blocks pairwise. The sorting network performs $B\log B(1+\log B)/4$ merges, each incurring a cost of $2n$ to merge two blocks. We experimentally verify this cost model in §8.4.

6.2 Oblivious query optimization

We now describe new optimization rules for a sequence of oblivious operators. Our rules operate on the lower-level operations within each oblivious operator, which we call Opaque operators.

6.2.1 Overview of the query planner

Before describing the Opaque operators, we provide an overview of the planning process, illustrated in Fig. 7. Opaque leverages the Catalyst query planner to transform a *SQL query* into an operator graph encoding the *logical plan*. Opaque interposes in the planning process to mark all logical operators that process sensitive data as oblivious. Catalyst can apply standard relational optimizations to the logical plan such as filter pushdown and join reordering.

Catalyst then generates a *physical plan* where each logical operator is mapped to one or more physical operators representing the choice of execution strategy. For example, a logical non-oblivious join operator could be converted to a physical hash join or a broadcast join based on the input cardinalities. Oblivious operators are transformed into physical Opaque operators at this stage, allowing us to express rules specific to combinations of oblivious operators. Similar to Catalyst, generating these physical operators allows Opaque to select from multiple implementations of the same logical operator based on table statistics. For example, if column cardinality is available, Opaque may use it to decide which oblivious aggregation algorithm to use. Catalyst then applies our Opaque rules to the physical plan.

The physical plan is then converted into an *encrypted representation* to hide information such as column names, constants, etc. Finally, Catalyst transforms the encrypted physical plan into a *Spark DAG* containing a graph of RDDs and executes it on the cluster.

6.2.2 Opaque operators

The following is a sampling of the physical Opaque operators generated during planning:

- `SORT`(C): obliviously sort on columns C
- `FILTER`: drop rows if predicate not satisfied
- `PROJECT-f`: similar to `FILTER`, but projects filtered out rows to 1, the rest to 0; preserves input size
- `HC-AGG`: stages 2–4 of the aggregation algorithm
- `SORT-MERGE-JOIN`: steps 2–4 of the sort-merge join algorithm

6.2.3 Query optimization

In this section, we give an example of an Opaque-specific rule:

$$\texttt{SORT}(C_2, \texttt{FILTER}(\texttt{SORT}(C_1, \texttt{PROJECT-f}(C_1)))) = \texttt{FILTER}(\texttt{SORT}(C_1, C_2, \texttt{PROJECT-f}(C_1)))$$

Let us take a look at how this rule would work with a specific query. We use the example query from §2.2, which translates to the following physical plan:

```
LC-AGG(disease,
  SORT(disease, FILTER(dummy,
    SORT(dummy_col, PROJECT-f(age, patient)))))
```

The filter will first do a projection based on the column age. To preserve obliviousness, the projected column is sorted and a real filter is applied. Since a sort-based aggregation comes after the filter, we need to do another sort on disease.

We make the observation that the second sort can be combined with the first sort into *one* oblivious sort on multiple columns. Since `PROJECT-f` always projects a column that is binary (i.e., the column contains only "0"s and "1"s), we can first sort on the binary column, then on

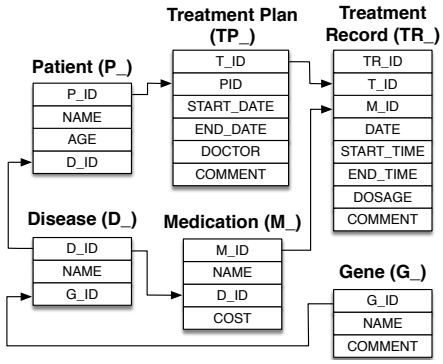


Figure 8: Example medical schema.

the second sort's columns (in this example, the disease column). Therefore, the previous plan becomes:

```
LC-AGG(disease, FILTER(dummy_col,
  SORT({dummy_col, disease},
    PROJECT-f(age, patient))))
```

This optimization is rule-based instead of cost-based. Furthermore, our rule is different from what a regular SQL optimizer applies because it pushes *up* the filter, while a SQL optimizer pushes *down* the filter. Filter push-down is unsafe because it does not provide obliviousness guarantees. Applying the filter before sorting will leak which records are filtered out.

6.3 Mixed sensitivity

Many applications operate on a database where not all of the tables are sensitive. For example, a hospital may treat patient information as sensitive while information about drugs, diseases, and various hospital services may be public knowledge (see Figure 8).

6.3.1 Sensitivity propagation

Propagation on tables. In a mixed sensitivity environment, tables that are not marked as sensitive could still be sensitive if they reveal information about other sensitive tables. Consider the example schema in Fig. 8. The Disease, Medication, and Gene tables are public datasets or have publicly known distributions in this example and therefore are not sensitive. Meanwhile the Patient table would likely be marked as sensitive. But what about Treatment Plan and Treatment Record? It turns out these tables are also sensitive because they *implicitly* embed patient information. Each treatment record belongs to a single patient, and each patient's plan may contain multiple treatment records. If an attacker has some prior knowledge, for example regarding what type of medication a patient uses, then observing only the Treatment Record table may allow the attacker to use an *inference attack* to gain further information about that patient such as their treatment frequency and other medication they may be taking.

To prevent such attacks, we use a technique from database literature called second path analysis [18]. The intuition for the inference attack is that information propagates along primary-foreign key relations: since each treatment record belongs to one treatment plan and one patient, the treatment record contains implicit information about patients. The disease table is connected to the patient table as well, except it has a primary key pointing *into* patient. This means that the disease table does not implicitly embed patient information.

Second path analysis accomplishes table sensitivity propagation by first directly marking user-specified tables as sensitive. After this is done, it recursively marks all tables that are reachable from every sensitive table via primary-foreign key relationships as sensitive as well. As in Fig. 8, such relationships are marked in an entity-relationship diagram using an arrow from the primary key table to the foreign key table.

This approach has been generalized to associations other than explicit foreign keys and implemented in automated tools [9]. We do not reimplement such analysis in Opaque, instead referring to the existing work.

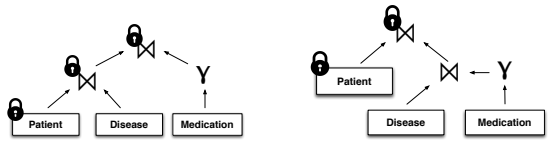
Propagation on operators. Another form of sensitivity propagation occurs when an operator (e.g., join) involves a sensitive and a non-sensitive table. In this case, we must run the entire operator obliviously. Additionally, for every leaf table that is marked sensitive in a query plan, sensitivity propagates on the path from the leaf to the root, and Opaque runs all the operators on this path obliviously.

6.3.2 Join reordering

Queries involving both sensitive and non-sensitive tables may contain a mix of oblivious and non-oblivious operators. Due to sensitivity propagation on operators, some logical plans may involve more oblivious operators than others. For example, a three-way join query where one table is sensitive may involve two oblivious joins if the sensitive table is joined first, or only one oblivious join if it is joined last (i.e., the non-sensitive tables are pushed down in the join order).

Join reordering in a traditional SQL optimizer centers on performing the most selective joins first, reducing the number of tuples that need to be processed. The statistics regarding selectivity can be collected by running oblivious Opaque queries. In Opaque, mixed sensitivity introduces another dimension to query optimization because of operator-level sensitivity propagation and the fact that oblivious operators are much more costly than their non-oblivious counterparts. Therefore, a join ordering that minimizes the number of oblivious operators may in some cases be more efficient than one that only optimizes based on selectivity.

Consider the following query to find the least costly medication for each patient, using the schema in Fig. 8:



(a) SQL join order (b) Opaque join order

Figure 9: Join reordering in mixed sensitivity mode.

```
SELECT    p_name, d_name, med_cost
FROM      patient, disease,
          (SELECT d_id, min(cost) AS med_cost
            FROM medication
            GROUP BY d_id) AS med
WHERE     disease.d_id = patient.d_id
          AND disease.d_id = med.d_id
```

We assume that the Patient table is the smallest, followed by Disease, then Medication ($|P| < |D| < |M|$), as might occur when considering only currently hospitalized patients and assuming there are multiple medications for each disease. The aggregation query reduces the cardinality of Medication to that of Disease and ensures a one-to-one relationship between the two tables.

Figure 9 shows two join orders for this query. A traditional SQL optimizer will execute the most selective join first, joining Patient with Disease, then with Medication. The optimal ordering for Opaque will instead delay joining Patient to reduce the number of oblivious joins. To see this, we now analyze the costs for both join orders.

Let C_{SQL} be the cost of this query using the SQL join order, C_{Opaque} the cost using the Opaque join order, and R the padded row size for all input tables. Note that the size of the Medication aggregate table is $|D|$.

$$C_{\text{SQL}} = 2C_{\text{o-join}}(|P| + |D|, R)$$
$$C_{\text{Opaque}} = C_{\text{join}}(2\,|D|, R) + C_{\text{o-join}}(|P| + |D|, R)$$

Assuming $C_{\text{join}} \ll C_{\text{o-join}}$,

$$\frac{C_{\text{SQL}}}{C_{\text{Opaque}}} \leq \frac{2C_{\text{o-join}}(|P| + |D|, R)}{C_{\text{o-join}}(|P| + |D|, R)} = 2$$

Thus, this query will see at most 2x speedup from join reordering. However, other queries can benefit still further from this optimization. Consider a three-way join of Patient, Disease, and Gene to extract the gene mutation affecting each patient. We assume Gene is a very large public dataset, so that $|P| < |D| < |G|$. Because Disease contains a foreign key into Gene, the three-way join occurs only on primary-foreign key constraints with no need for aggregation. As before, a traditional SQL optimizer would execute $(P \bowtie D) \bowtie G$ while Opaque will run $(G \bowtie D) \bowtie P$. The costs are as follows:

$$C_{\text{SQL}} = C_{\text{o-join}}(|P| + |D|, R) + C_{\text{o-join}}(|P| + |G|, R)$$
$$C_{\text{Opaque}} = C_{\text{join}}(|G| + |D|, R) + C_{\text{o-join}}(|D| + |P|, R)$$

Assuming $C_{\text{join}} \ll C_{\text{o-join}}$ and $|P| < |D| \ll |G|$,

$$\frac{C_{\text{SQL}}}{C_{\text{Opaque}}} = \frac{C_{\text{o-join}}(|P| + |G|, R)}{C_{\text{join}}(|G| + |D|, R)} \approx \frac{C_{\text{o-join}}(|G|, R)}{C_{\text{join}}(|G|, R)}$$

The maximum theoretical performance gain for this query therefore approaches the performance difference between the Opaque and non-oblivious join operators. We demonstrate this empirically in Fig. 12b.

Limitations. Note that sensitivity propagation optimizes efficiently when the large tables in a database are not sensitive. This makes intuitive sense because computation on larger tables contributes more to the query runtime. If the larger tables are sensitive, then join reordering cannot help because any join with these tables must always be made oblivious. Therefore, the underlying schema will have a large impact on the effectiveness of our cost-based query optimizations.

6.4 Query planning for oblivious pad mode

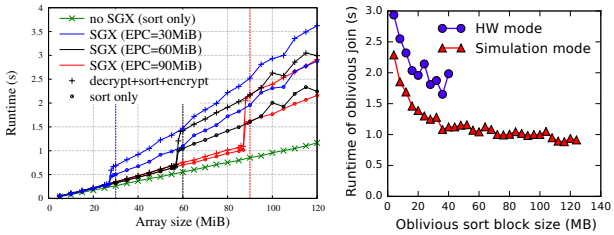
As discussed in §3.3, the fact that the planner chose a query plan over another plan leaks some information about the selectivity of some operators. For example, generalized inner joins' costs depend on join selectivity information. This is not a problem for primary-foreign key joins because these costs can be estimated using only the size of each table: the output size of such a join is always the size of the foreign key table.

Oblivious pad mode does not leak such statistics information. All filters are pushed up and combined together at the end of the query. The optimizer does not need to use selectivity information because the overall size will not be reduced until the very end. Thus, our query planning stage only needs to use publicly-known information such as the size of each table.

7 Implementation

Opaque is implemented on top of Spark SQL, a big data analytics framework. Our implementation consists of 7000 lines of C++ enclave code and 3600 lines of Scala.

We implemented the Opaque operators and query optimization rules from §6 by extending Catalyst using its developer APIs with minimal modifications to Spark. Our operators are written in Scala and execute in the untrusted domain, making trusted calls to the enclave when necessary through JNI. For example, the `SORT` operator performs inter-machine sorting using an RDD-based implementation of distributed column sort in the untrusted domain (§5.1.2). Within each partition, the `SORT` operator serializes the encrypted rows and passes them using JNI to the worker node's enclave, which then performs the local sort in the trusted domain (§5.1.1). Our implementation currently does not support arbitrary user-defined functions (UDFs) due to the difficulty in making them oblivious.



(a) Varying EPC size (b) Oblivious sort block size

Figure 10: Sort microbenchmarks. (a) Non-oblivious sort in SGX. Exceeding EPC size causes a dramatic slowdown. (b) Oblivious sort in SGX. Larger blocks improve performance until the EPC limit in HW mode, or indefinitely in simulation mode.

Opaque encrypts and integrity-protects data on a block-level basis using AES in GCM mode, which provides data confidentiality as well as integrity. We pad all rows within a table to the same upper bound before encrypting. This is essential for tables with variable-length attributes as it prevents an attacker from distinguishing between different rows as they move through the system.

8 Evaluation

In this section, we demonstrate that Opaque represents a significant performance improvement over the state of the art in oblivious computation, quantify its overhead compared to an insecure baseline, and measure the gains from our query planning techniques.

8.1 Experimental setup

Single-machine experiments were run using SGX hardware on a machine with Intel Xeon E3-1280 v5 (4 cores @ 3.70GHz, 8MiB cache) with 64GiB of RAM. This is the maximum number of cores available on processors supporting SGX at the time of writing.

Distributed experiments were run on a cluster of 5 SGX machines with Intel Xeon E3-1230 v5 (4 cores @ 3.40GHz, 8MiB cache) with 64GiB of RAM.

8.2 Impact of oblivious memory size

We begin by studying the impact of the secure enclave memory size and show that Opaque will benefit significantly from future enclave implementations with more memory. SGX maintains an encrypted cache of memory pages called the Enclave Page Cache, which is small compared to the size of main memory. Once a page is evicted from the EPC, it is decrypted if it was not entirely in CPU cache, re-encrypted under a different key, and stored in main memory. When an encrypted page in main memory is accessed, it needs to be decrypted again. This paging in and out of the EPC introduces a large overhead. Current implementations of SGX have a maximum effective EPC size of 93.5MiB, but this will be significantly increased in upcoming versions of SGX.

Sorting is the core operation in Opaque, so we studied how SGX affected its performance. In Fig. 10a, we benchmark non-oblivious sorting (introsort) in SGX by sorting arrays of 64-bit integers of various sizes using EPCs of various sizes. We also measure the overhead incurred by decrypting input data and encrypting output data before and after sorting using AES-GCM-128. We see that exceeding the EPC size even by just a little incurs a 50 ~ 60% overhead. When below the EPC limit, the overhead of encryption for I/O is just 7.46% on average. The overhead of the entire operation versus the insecure baseline is 31.7% on average.

Having a part of EPC that is oblivious radically improves performance. In §3.2, we discussed existing and upcoming designs for such an EPC. In Fig. 10b, we call this an oblivious block size, and we benchmarked the performance of oblivious sort with varying block sizes (§5.1.1). Within a block, regular quicksort can happen which speeds up performance. The case when only the registers are oblivious (namely an oblivious block of the same size as the available registers) did not fit in the graph: the overhead was 30x versus when the L3 cache (8MB) is oblivious. We see that in hardware mode, more oblivious memory improves performance until a sort block size of 40 MB, when the working set (two blocks for merging) exceeds the hardware EPC size, causing thrashing, as occurred in Fig. 10a near EPC limits. In simulation mode, no thrashing occurs. In sum, Opaque's performance will improve significantly when run with more oblivious memory as a cache.

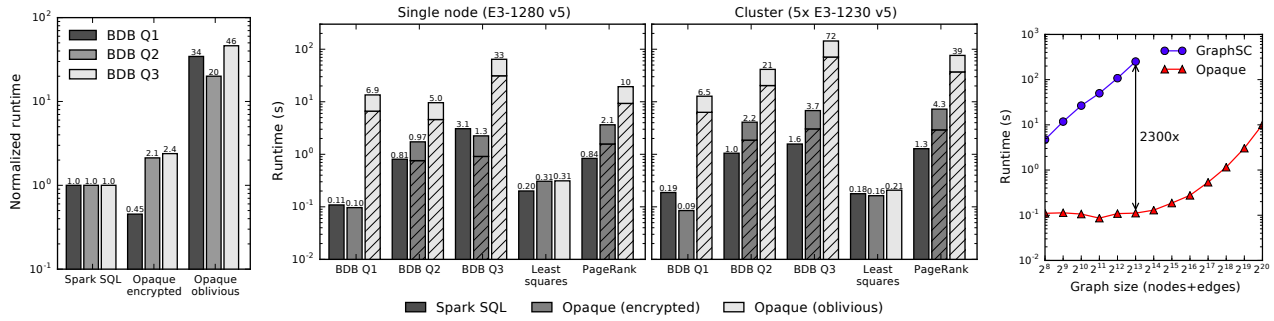
8.3 System comparisons

8.3.1 Comparison with Spark SQL

We evaluated Opaque against vanilla Spark SQL, which provides no security guarantees, on three different workloads: SQL, machine learning, and graph analytics.

For the SQL workload, we benchmarked both systems on three out of four queries of Big Data Benchmark [1], a popular benchmark for big data SQL engines. The fourth query is an external script query and is not supported by our system. The three queries cover filter, aggregation (high cardinality), and join. For the machine learning workload, we chose least squares regression on 2D data; this query uses projection and global aggregation. Finally, we chose to benchmark PageRank for the graph analytics workload; this query uses projection and aggregation.

We show our results in two graphs, Figure 11a and Figure 11b. Figure 11a shows the performance of each of Opaque's security modes on the Big Data Benchmark in the distributed setting. Higher security naturally adds more overhead. Encryption mode is competitive with Spark SQL (between 52% improvement and 2.4x slowdown). The performance gain comes from the fact that



(a) Security mode comparison (b) Comparison to Spark SQL (c) Comparison to GraphSC

Figure 11: (a) Encryption mode is competitive with Spark SQL. Obliviousness (including network and memory obliviousness) adds up to 46x overhead. (b) Comparison across a wide range of queries. Hatched areas represent time spent sorting. (c) Single iteration of PageRank for various graph sizes.

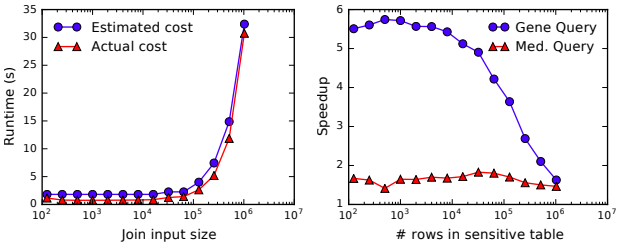
Opaque runs C++ in the enclave, while Spark SQL incurs overhead from the JVM. Opaque's oblivious mode adds 20–46x overhead.

Figure 11b shows Opaque's performance on five queries. Hatched areas show the time spent in oblivious sort, the dominant cost. The left side of Figure 11b shows Opaque running on a single machine using SGX hardware compared to Spark SQL, while the right side shows the distributed setting. In the single-machine setting, Opaque's encryption mode performance varies from 58% performance gain to 2.5x performance loss when compared with the Spark SQL baseline. The oblivious mode (both network and memory oblivious) slows down the baseline by 1.6–62x. The right side shows Opaque's performance on a distributed SGX cluster. Encryption mode's performance ranges from a 52% performance improvement to a 3.3x slowdown, while oblivious mode adds 1.2–46x overhead. In these experiments, Opaque was configured with oblivious memory being the L3 cache and not the bulk of EPC. As discussed in §8.2, more oblivious memory would give better performance, and such hardware proposals already exist (see §3.2).

8.3.2 Comparison with GraphSC

We use the same PageRank benchmark to compare with the existing state-of-the-art graph computation platform, GraphSC [27]. While Opaque is more general than graph computation, we compared Opaque with GraphSC instead of its more generic counterpart ObliVM [22], because ObliVM is about ten times slower than GraphSC.

We used data from GraphSC and ran the same experiment on both systems on our single node machine, with Opaque running in hardware mode with obliviousness. Figure 11c shows that Opaque is faster than GraphSC for all data sizes. For 8K graph size, Opaque is 2300x faster than GraphSC. This is consistent with the ObliVM and GraphSC papers: ObliVM reports a 9.3×10^6x slowdown, and GraphSC [27] a slowdown of 2×10^5x to 5×10^5x.



(a) Accuracy of obl. join cost model (b) Speedup from join reordering

Figure 12: Query planning benchmarks. (a) Our cost model closely approximates the empirical results for oblivious joins across a range of input sizes. (b) Join reordering provides up to 5x speedup for some queries.

Though GraphSC and Opaque share the high-level threat model of an untrusted service provider, they relax the threat model in different ways, explaining the performance gap. Opaque relies on trusted hardware, while GraphSC relies on two servers that must not collude and are semi-honest (do not cheat in the protocol) and so must use garbled circuits and secure two-party computation, which are much slower for generic computation than trusted hardware.

8.4 Query planning

We next evaluate the query planning techniques proposed in §6. First, we evaluate the cost model presented in §6.1 using a single-machine microbenchmark. We run an oblivious join and vary the input cardinality. We then fit the equation from §6.1 to the empirical results. Figure 12a shows that our theoretical cost model closely approximates the actual join costs.

Second, to evaluate the performance gain from join reordering, we run the two queries from §6.3.2. Figure 12b shows the speedup from reordering each query with varying sizes of the sensitive patient table. The medication query sees just under 2x performance gain because two equal-sized oblivious joins are replaced by

one oblivious and one non-oblivious join. The gene query sees a 5x performance gain when the sensitive table is small because the larger oblivious join is replaced with a non-oblivious join. As the sensitive table increases in size, the benefit of join reordering approaches the same level as for the medication query.

9 Related work

9.1 Relevant cryptographic protocols

ORAM. Oblivious RAM [13, 37, 38, 36] is a cryptographic construct for protecting against access pattern leakage. However, ORAM does not fit in Opaque's setting because it has an intrinsically different computation model: serving key-value pairs. We show this problem by devising a simple strawman design using ORAM: put all data items in an in-memory ORAM in Spark.

How can ORAM be utilized if we attempt to sort data, which is an essential operation in SQL? One way to implement sorting on top of ORAM is to simply treat a sorting algorithm's compare-and-swap operation as two ORAM reads and two ORAM writes. This is not viable for three reasons. First, making an ORAM access for each data item is very slow. Second, current ORAM designs are not parallel and distributed, which means that the ORAM accesses will be serialized. Third, we cannot use a regular sorting algorithm because the number of comparisons may be different when run on different underlying data values. This could leak something about the encrypted data and would not provide obliviousness. Therefore, we must use a sorting network anyway, which means that adding ORAM will add an extra polylog(n) factor of accesses.

Other protocols. Fully homomorphic encryption [11, 12] permits computing any function on encrypted data, but is prohibitively slow. Oblivious protocols such as sorting and routing networks [7] are more relevant to Opaque, and Opaque builds on these as discussed in §5.1.

9.2 Non-oblivious systems

A set of database systems encrypt the data so that the service provider cannot see it. These databases can be classified into two types. The first type are encrypted databases, such as CryptDB [33], BlindSeer [31], Monomi [39], AlwaysEncrypted [26], and Seabed [30], that rely on cryptographic techniques for computation. The second type are databases, such as Haven [6], VC3 [34], TrustedDB [5], TDB [23] and GnatDb [40], that require trusted hardware to execute computation.

The main drawback of these systems is that they do not hide access patterns (both in memory and over the network) and hence leak data [41, 28]. Additionally, most of these systems do not fit the distributed analytics setting.

9.3 Oblivious systems

Non-distributed systems. Cipherbase [2] uses trusted hardware to achieve generic functionality for encrypted databases. The base Cipherbase design is not oblivious, but Arasu and Kaushik [3] have proposed oblivious protocols for SQL queries. However, unlike Opaque, their work does not consider the distributed setting. In particular, the proposed oblivious operators are not designed for a parallel setting resulting in sequential execution in Opaque, and do not consider boundary conditions. In addition, Cipherbase's contribution is a design proposal, while Opaque also provides a system and an evaluation.

Ohrimenko et al. [29] provide oblivious algorithms for common ML protocols such as matrix factorization or neural networks, but do not support oblivious relational operators or query optimization. Their focus is not on the distributed setting, and parts of the design (e.g., the choice of a sorting network) and the evaluation focus on single machine performance.

Distributed systems. ObliVM [22] is a platform for generic oblivious computation, and GraphSC [27] is a platform specialized to distributed graph computations built on ObliVM. As we show in §8.3, these systems are three orders of magnitude slower than Opaque. As explained there, they have a different threat model and use different techniques resulting in this higher overhead.

Ohrimenko et al. [28] and M2R [10] provide mechanisms for reducing network traffic analysis leakage for MapReduce jobs. Their solutions do not suffice for Opaque's setting because they do not protect in-memory access patterns. Moreover, they are designed for the simpler setting of a MapReduce job and do not suffice for Opaque's relational operators; further, they do not provide global query optimization of oblivious operators.

10 Conclusion

In this paper, we proposed Opaque, a distributed data analytics platform providing encryption, oblivious computation, and integrity. Opaque contributes a set of distributed oblivious relational operators as well as an oblivious query optimizer. Finally, we show that Opaque is three orders of magnitude faster than state-of-the-art specialized oblivious protocols.

Acknowledgments

We thank the reviewers, the shepherd, Mona Vij and other colleagues from Intel, and Aurojit Panda, for their valuable feedback or discussions. This work was supported by the Intel/NSF CPS-Security grants #1505773 and #20153754, DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, the UC Berkeley Center for Long-Term Cybersecurity, as well as gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

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

[2] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *6th*, Asilomar, CA, Jan. 2013.

[3] 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.

[4] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. 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, Australia, June 2015.

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

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Sorting networks. In *Introduction to algorithms*, chapter 27. MIT Press, 2001.

[8] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*, Aug. 2016.

[9] H. S. Delugach and T. H. Hinke. Wizard: A database inference analysis and detection system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):56–66, Feb. 1996.

[10] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[11] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC)*, Bethesda, MD, May 2009.

[12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099, June 2012.

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

[14] J. E. Gonzalez, D. Crankshaw, A. Dave, R. S. Xin, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[15] Google Research. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. https://www.tensorflow.org/, 2015.

[16] T. Greene. Biggest data breaches of 2015. Network world. http://www.networkworld.com/article/3011103/security/biggest-data-breaches-of-2015.html, 2015.

[17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[18] T. H. Hinke. Inference aggregation detection in database management systems. In *IEEE Symposium on Security and Privacy*, pages 96–106, 1988.

[19] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. White paper, Apr. 2016.

[20] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[21] C. Liu, A. Harris, M. Maas, M. 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, pages 87–101, 2015.

[22] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[23] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000.

[24] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[25] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in apache spark. In *Journal of Machine Learning Research, 17(34):1Ð7*, 2016.

[26] Microsoft. Always encrypted database engine. https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[27] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: parallel secure computation made easy. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[28] 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*, pages 1570–1581. ACM, 2015.

[29] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *Proceedings of the 25th USENIX Security Symposium*, Aug. 2016.

[30] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[31] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[32] F. Pennic. Anthem suffers the largest healthcare data breach to date. HIT consultant. `http://hitconsultant.net/2015/02/05/anthem-suffers-the-largest-healthcare-data-breach-to-date/`, 2015.

[33] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[34] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.

[35] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *In Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[36] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.

[37] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652*, 2011.

[38] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[39] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, pages 289–300, Riva del Garda, Italy, Aug. 2013.

[40] R. Vingralek. GnatDb: A small-footprint, secure database system. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, Aug. 2002.

[41] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2012.