Oblivious Coopetitive Analytics Using Hardware Enclaves

Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez and Ion Stoica
ankurd,chester,raluca,jegonzal,istoica@eecs.berkeley.edu
UC Berkeley

Abstract

Coopetitive analytics refers to cooperation among competing parties to run queries over their joint data. Regulatory, business, and liability concerns prevent these organizations from sharing their sensitive data in plaintext.

We propose Oblivious Coopetitive Queries (OCQ), an efficient, general framework for oblivious coopetitive analytics using hardware enclaves. OCQ builds on Opaque, a Spark-based framework for secure distributed analytics, to execute coopetitive queries using hardware enclaves in a *decentralized* manner. Its query planner chooses how and where to execute each relational operator to prevent data leakage through side channels such as memory access patterns, network traffic statistics, and cardinality, while minimizing overhead.

We implemented OCQ as an extension to Apache Spark SQL. We find that OCQ is up to 9.9x faster than Opaque, a state-of-the-art secure analytics framework which outsources all data and computation to an enclave-enabled cloud; and is up to 219x faster than implementing analytics using AgMPC, a state-of-the-art secure multi-party computation framework.

1 Introduction

Distributed analytics frameworks [79] are now widely used, but are designed to operate on data owned by one entity. Federated databases, which span data owned by multiple cooperating parties, have a long history in the database community [27, 28, 70, 80]. This community has focused on the case when the organizations trust each other and can share data.

However, there are many applications in which organizations *cannot share plaintext data* with each other because, for example, they are in business competition, or due to privacy regulations and liability concerns. Nevertheless, collaboration among these competing organizations could enable new applications. For example, banks would like to perform analytics over their aggregate data to better detect money laundering,

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387552

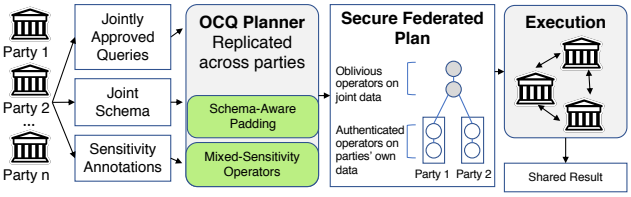


Figure 1. OCQ executes approved federated queries using hardware enclaves. Its query planner and relational operators hide memory and network access patterns and sensitive cardinalities.

but cannot share the data with each other because they are in competition. The Chief Risk Officer of Scotiabank stated that such collaboration "will enhance yields by orders of magnitude" [25]. We are partnering with Scotiabank for this use case. As another example, consider a consortium of hospitals that want to pool their patient data for researchers to access. Researchers need the ability to correlate patients across hospitals, yet regulations prevent the hospitals from sharing raw patient data or letting it leave the premises [17]. Following previous work [82], we refer to this setting of analytics among cooperative and competing parties as *coopetitive* analytics.

In a coopetitive setting, parties agree in advance on a shared schema and a set of allowed queries. They agree to run these queries on their private data and share only the results. In particular, they will not share the actual database of any party or intermediate results in the query computation.

Prior work in the coopetitive setting uses either specialized cryptography or hardware enclaves. Cryptography [1, 7, 42, 51, 64, 82] either offers limited functionality too restrictive for general analytics, as is the case for partially-homomorphic encryption, or introduces enormous overheads, taking hours to months for typical queries, as is the case for secure multi-party computation (MPC), as we show in §8.

Approaches based on hardware enclaves [26, 53, 60, 81] are more promising for performance. While hardware enclaves have many side channels [14, 15, 46, 69, 74, 77], prior work [35] shows that many classes of side channels disappear if one designs the computation to be *oblivious*: memory accesses are independent of sensitive data. In a distributed setting, network traffic patterns also create a side channel [59]. Hence, to use hardware enclaves for coopetitive analytics requires oblivious protocols designed for this setting. The overarching challenge is that such protocols are slow.

The most relevant related work to OCQ is likely Opaque [81], which offers oblivious analytics on data outsourced to a *single* untrusted cloud. However, aggregating multiple parties' sensitive data to a single location suffers from several drawbacks in the coopetitive setting. First,

transferring and maintaining a remote copy of large data incurs significant overhead especially if this data changes frequently. Second, this strategy may run afoul of regulations that forbid a database from being moved out of a certain perimeter. Third, the oblivious computation in Opaque crucially assumes that all communication happens within the same cloud; applying Opaque's algorithms to query execution across different parties connected by a wide-area network results in prohibitive overhead.

In this paper, we propose Oblivious Coopetitive Queries (OCQ), a general framework for coopetitive analytics based on hardware enclaves, overviewed in Figure 1. Rather than requiring multiple parties' data to be aggregated to a single location, OCQ executes queries in a decentralized manner. OCQ develops efficient *oblivious query algorithms* (e.g., oblivious federated join) for the federated setting and a *schema-aware padding* mechanism, which combined prevent data leakage through (1) memory accesses to data inside the enclaves, and (2) network traffic patterns outside the enclaves, both within a local data center and in the wide area. OCQ also contributes an *oblivious planner*, which determines where to execute each operation and how to execute it, to minimize the overhead while maintaining the oblivious security guarantees.

We implemented OCQ as an extension to Apache Spark SQL's Catalyst query planner and execution engine. We evaluate OCQ using SGX-enabled, geographically distributed clusters on a variety of synthetic benchmarks and find that OCQ is up to 9.9× faster than outsourcing all data and computation to a third party running Opaque. Compared to AgMPC [75], a state-of-the-art cryptographic framework for secure multiparty computation, OCQ is up to 219× faster.

OCQ's design addresses the following challenges:

Challenge 1: Oblivious queries in the wide area. The first step of distributed operators such as aggregations and joins is typically to shuffle the entire relation to colocate the appropriate records. For example, Opaque implements aggregation using an initial distributed sort based on the grouping attributes to colocate records that belong to the same group. However, in the coopetitive setting, this incurs record movement across the wide area, requiring the use of expensive security protocols to avoid leaking information.

Approach: Federated and oblivious planner. Our federated planner chooses operators that maximize the computation run at originating parties and ensures that communication across the wide area does not leak information about the input data. For example, while a high-cardinality aggregation that results in many groups would normally be implemented using an initial distributed sort, OCQ's planner instead chooses to compute partial aggregates at each party and shuffle those partial aggregates across the wide area, with padding to hide the number of groups from each party. The worst-case upper bound on the number of groups is often much smaller than the number of rows in the database, for example, for fields containing gender or age. This approach avoids exchanging un-aggregated records between parties. Key to performance is that parts of the computation running within a party's cluster that only touch that party's data need not be oblivious because the data is known to the party. Such local computation will still run inside the enclaves for integrity and authentication.

Challenge 2: Combining data of mixed sensitivities. Queries may consist of operators that combine slices of multiple parties' data. Because OCQ executes these operators at the parties themselves, and because parties may provide table-level sensitivity annotations, many relational operators in OCQ combine data of varying sensitivity levels. For example, one party may execute a join operator between its own data and a slice of sensitive data from other parties. Executing such operators using fully-oblivious algorithms would incur unnecessary overhead.

Approach: Mixed-sensitivity algorithms. OCQ introduces mixed-sensitivity operators, such as a mixed-sensitivity oblivious join algorithm based on the merge phase of bitonic sort that provides up to 2.5x speedup compared to a fully-oblivious join.

Challenge 3: Query planning with sensitive cardinalities. Query planners traditionally rely on statistics to choose among multiple plans. However, such statistics are sensitive in a coopetitive setting, as they reveal information about the distribution of each party's data. The plan chosen can leak information about the input statistics. Additionally, the cardinalities of intermediate relations may leak information, such as the selectivity of a filter. Cardinality leakage poses a further threat in the coopetitive setting than in the outsourced computation setting because a malicious party can manipulate its input to extract information through cardinalities.

Approach: Schema-aware padding. As in previous work on Opaque [81], we take the approach of padding input and intermediate relations to publicly-known bounds to hide sensitive cardinalities. Our contribution is a scheme to refine these bounds by exploiting the likely presence of foreign key relationships between public and private relations in each party's schema to find tighter padding bounds for each operator. For example, a query to find all distinct disease diagnoses across multiple hospitals would typically involve padding to the number of patient diagnoses, while a foreign key relationship to a set of diseases would enable OCQ to pad to the possibly much smaller number of registered diseases. We introduce rules to propagate these bounds through the query plan. Conveniently, padding rules also obviate the problem of leakage via the choice of plan, because the resulting padding bounds provide exact cardinality information without leaking sensitive statistics.

2 Background

OCQ is designed to enable coopetitive analytics using hardware enclaves. Here we describe the coopetitive setting and provide background on the building blocks for OCQ.

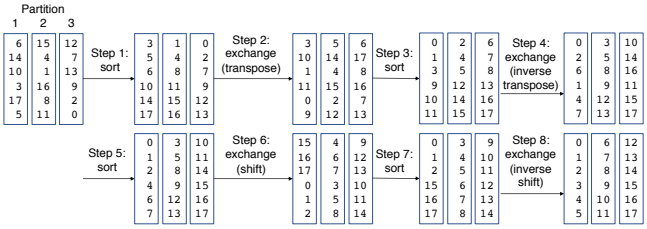


Figure 2. Illustration of column sort algorithm. Each column represents an encrypted partition. Column sort enables oblivious distributed sorting using four intra-partition sorts (steps 1, 3, 5, 7) and four data exchanges (2, 4, 6, 8) in fixed patterns. An attacker only sees exchanges of encrypted records. Since the pattern of exchanges is fixed, it cannot leak data contents.

2.1 Hardware enclaves

Hardware enclaves or trusted execution environments (TEEs) such as Intel SGX [50], AMD Memory Encryption [41], Keystone [44], Sanctum [22], MI6 [12] and others [3, 72] enable code to run in an isolated environment where other processes on the same host, including the OS and hypervisor, cannot tamper with its execution or access its memory. Enclaves also provide remote attestation, which allows the enclave to prove to a client that it is running the desired code and to establish a secure channel to a client. We discuss in §4 the enclave threat model we build OCQ on.

2.2 Oblivious algorithms

As we discuss in §4, enclaves suffer from side channels exploiting memory access patterns to data and traffic patterns. OCQ protects against these side channels with oblivious computation and appropriate padding. Oblivious algorithms aim to process data while ensuring that their memory accesses are independent of the contents of the data; this also implies that the network traffic patterns are also independent of data content. For example, basic matrix multiplication is oblivious, because its access pattern depends only on the size of the inputs and not their numerical values. In contrast, quicksort is not oblivious because its access pattern depends on the ordering of the data: in each iteration, records smaller than the pivot are swapped to one memory region, while records larger than the pivot are swapped to another. The choice of where to swap each record depends on the record contents.

Because sorting is at the heart of most database operators, efficient oblivious sorting algorithms are of particular interest. Single-machine oblivious sorting can be done using sorting networks that perform a fixed sequence of compare-exchange operations. Asymptotically more compare-exchange operations are needed for oblivious sorting than for traditional sorting. An oblivious compare-exchange can be implemented via a comparison followed by a conditional swap of two equal-length buffers depending on the result of the comparison.

For data partitioned across multiple machines, oblivious sorting can be accomplished using a two-level sorting algorithm in which each partition is individually sorted using a sorting network, and records are sorted across partitions using

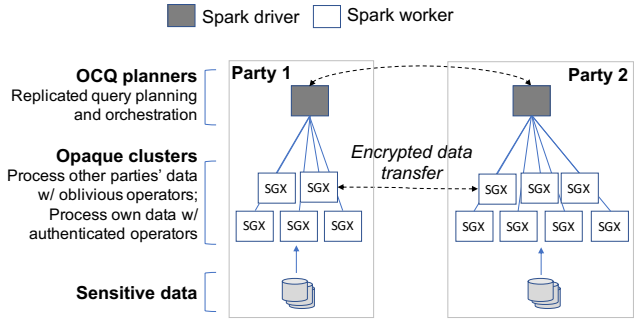


Figure 3. Architecture of OCQ. OCQ's replicated federated planner executes operators on Opaque and Spark clusters at each party. Sensitive data never leaves its originating party in plaintext.

an algorithm called column sort [47]. Column sort consists of a fixed sequence of data exchange and intra-machine sorting that uses only 4 shuffles, compared to $O(n \log^2 n)$ shuffles for a sorting-network-based distributed sort. It is thus well suited to oblivious distributed sorting, where it was previously applied by Opaque [81]. OCQ also uses it for sorting sensitive data; the algorithm is illustrated in Figure 2.

2.3 Spark SQL and Opaque

OCQ's planner and federated execution engine are built on Spark SQL [4, 79], a distributed SQL analytics framework, and Opaque [81], an extension of Spark SQL for secure outsourced computation via hardware enclaves.

Spark SQL offers distributed plaintext query execution. Queries can be written in SQL or an embedded Scala DSL called DataFrames. The user submits queries to the Spark SQL driver, which parses them into logical plan format. Spark SQL's extensible rule-based query planner, Catalyst, which also runs at the driver, optimizes these plans and generates a physical plan for execution on a Spark cluster. Catalyst is primarily rule-based, but offers limited statistics collection and cost-based optimization. The physical plan breaks the query into stages consisting of parallel tasks that are executed on workers. Each worker writes results to distributed storage or returns them to the driver.

Opaque extends Spark SQL to the untrusted cloud setting, where the driver is trusted but the workers are not. By extending Catalyst with encrypted operators which result in tasks that run inside SGX enclaves rather than in plaintext, it enables distributed queries on encrypted data.

OCQ's federated planner is an extension of Catalyst, and OCQ leverages Opaque to process encrypted data within a single party. This enables OCQ to inherit Spark SQL's query languages and optimizations, and Opaque's secure distributed query processing. However, OCQ must implement rules and orchestration logic specific to secure *federated* queries.

3 Architecture

Figure 3 shows OCQ's architecture. Each party maintains a Spark cluster with at least one hardware enclave-enabled machine, on which Opaque tasks are scheduled. OCQ's query planner is deterministic and runs outside the enclave at every

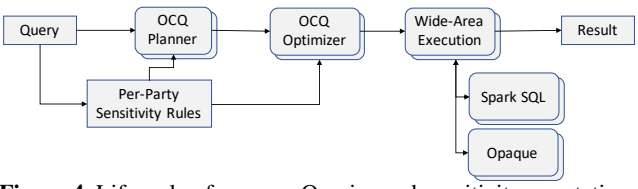


Figure 4. Lifecycle of a query. Queries and sensitivity annotations are submitted to OCQ's federated planner and optimizer. The resulting federated plan contains operators running at different locations across the federation, and satisfies all parties' sensitivity annotations. The plan executes securely, and the user only learns the result and cannot access sensitive intermediate relations. Stacked blocks indicate that a component is present at each party.

party. This is because our query planner builds on Spark SQL's planner, which is a large Scala codebase that would significantly broaden the enclave's attack surface and require heavyweight sandboxing techniques. To reduce coordination between parties at query time, each party runs a replica of OCQ's federated planner; we describe the mechanism for verifying the plan's integrity in §3.1. Each planner replica maintains an audit log of all queries issued by any party.

The role of each party's Opaque clusters is to (1) give assurance that the computation at each party happens correctly, and (2) safely mix multiple parties' data. Therefore it is essential for enclave code to be trusted by all parties. OCQ accomplishes this by granting all parties the ability to invoke pre-approved routines on all enclaves, but ensuring that each enclave verifies that the deployed code is approved by all the parties via remote attestation.

In OCQ, parties own different tables; a logical table consisting of rows from different parties can be implemented using union. Parties annotate their tables as either public or sensitive. The query planner determines the sensitivity level of intermediate results using sensitivity propagation rules discussed in §5.

3.1 Setup phase

Parties must share the cryptographic hash of each encrypted partition of each of their input relations with all other parties. Once the setup phase has completed, no party can change its input data, and the enclaves will ensure this.

Parties setup their enclaves using OCQ's code, and perform remote attestation amongst all these clusters. Consider a logical enclave at each party (which can be implemented via a cluster of enclaves). The result of this stage is:

- Every party and its enclave know the public keys of every party, the schema and sensitivity of every table of each party, and the hashes of each party's encrypted data.
- Each enclave checks that the enclaves at the other parties were correctly setup with OCQ with the same information.
- The enclaves agree on symmetric keys for a secure channel amongst themselves.

Queriers may attempt to submit malicious queries designed to extract sensitive data, and a compromised planner replica may produce a physical plan that reveals sensitive information. We prevent both of these by requiring all parties to agree on the allowed set of queries, the resulting query plans, and size bounds for sensitive input and intermediate data sets. The agreed-upon configuration is represented as a set of DDL statements, queries, and physical plans that is signed by all parties and passed to each party's instance of the OCQ federated planner and enclave. Before signing the configuration, each party should check that it matches expectations.

3.2 Query lifecycle

Figure 4 shows the lifecycle of a query. A user submits a query to a federated query planner replica, which broadcasts it to all the others, one planner instance per party. Each planner first checks that the query conforms to the set of approved queries and then performs the same planning and optimization steps, deterministically generating a federated plan with operators running at different locations. The generated plan satisfies all parties' sensitivity annotations and performs as much computation as possible in plaintext at each party. The party signs the plan, and enclaves will execute only plans signed by all the parties. Each planner runs the relevant operators at its local Spark and Opaque clusters.

Special data movement operators trigger data exchange with other parties across the wide area, or between one party's Spark and Opaque clusters, when allowed by the sensitivity annotations. The final operator collects the results back to the originating replica and returns them to the user.

A party can observe network traffic generated by computation on other parties' data when that computation is run within their Opaque cluster, such as during final aggregation. To prevent size leakage in this case, OCQ automatically determines appropriate public upper bounds for all intermediate results and ensures that each operator pads its output to the appropriate bounds. §5.4 describes this process in more detail.

Our integrity mechanism for ensuring that a malicious party cannot tamper with the integrity of the computation or input data is similar to that of Opaque [81]'s self-verifying computation, so we only describe it at a high level. At query time, the signed physical plans are loaded into the enclaves, which check that all parties signed every plan. When an enclave is requested to execute part of a query plan, it verifies that each of the inputs to the plan fragment were either authentic input data or were generated by the expected child plan, depending on the expected input source. To check that the input is authentic, when scanning a party's input data, the scanning enclave checks the partition's hash against the party's hash from setup to ensure that the party has not tampered with the input data. If the hashes do not match, the enclave signals failure and the query will abort. After executing the plan fragment, the enclave certifies that its output was correctly produced by running the requested plan fragment.

4 Threat model and security guarantees

4.1 Abstract enclave model

OCQ considers an attacker who can see memory accesses to data and/or messages sent over the network. Such an attacker arises from a large class of attacks: attacks leveraging page faults [16, 74, 77], the branch predictor [46], cache timing [14, 69], the memory bus [43], network traffic patterns [59], and others. Oblivious algorithms are impressively effective against such a large variety of attacks [35] because they address the core leakage of these channels: memory accesses based on sensitive data. OCQ contributes oblivious algorithms for the federated analytics setting to thrwart such attacks. We group the attackers in two categories:

– A **network attacker** that sees all network traffic but has no access to the machines. In particular, this attacker does not see any memory accesses inside any of the machines.

– A **malicious party attacker** that sees all memory accesses made by enclaves in addition to the network traffic. For this party we assume the oblivious row-level conditional-exchange primitive from §2.2.

OCQ builds on top of an *abstract* enclave model; OCQ's design is not tied to Intel SGX even if our prototype implementation in §7 builds on it. There are many different proposals for hardware enclaves, such as Intel SGX [50], AMD Memory Encryption [41], Keystone [44], Sanctum [22], MI6 [12] and others [3, 72], with more upcoming as researchers make rapid progress towards more secure hardware enclaves. OCQ assumes that the attacker cannot access or undetectably modify data or code inside the hardware enclave, that it cannot exploit side-channels different from the memory addresses discussed above, subvert the remote attestation process or otherwise the integrity or secrecy of the enclave; if the attacker could, OCQ does not protect against such attacks. Such side-channel attacks indeed exist in some enclave implementations (such as in Intel SGX); addressing them should be done at a different level of abstraction than OCQ operates at, for example, through better enclave design. For instance, Intel SGX suffers from various vulnerabilities or side channels, such as attacks based on speculative execution [15, 19, 68], power consumption [56, 71], rollback [63], intra cache line memory accesses [54, 78], denial-of-service attacks [32, 38], and others (e.g., [45, 76]). Many defenses or mitigations have been proposed, such as for closing hyperthreading-based attacks [20, 61], rollback defenses [13, 49], and importantly, improved enclave designs that remove a wide array of the attacks above, such as KeyStone [44] or MI6 [12] (the latter even protects against speculative-execution attacks). We hope that OCQ's design will be ported to better and better enclaves as research progresses on this front.

We remark that our implementation prototype of OCQ described in §7 focuses on obliviousness with respect to *data* at the *cache-line* granularity. While OCQ's oblivious algorithms are oblivious even when it comes to accesses to OCQ's pseudocode, we have not ensured that our implementation and the generated binary preserve this property. Also, while OCQ's oblivious algorithms could be applied at the intra-cache-line granularity [54, 78], our prototype implementation does not implement them at this level. Both of these can be addressed in the implementation with existing mechanisms (e.g., [20, 23, 31, 48, 61]) at a performance cost.

4.2 Party threat model

In the coopetitive setting, a malicious party could attempt to tamper with the other parties' data during joint computation, or it could attempt to inspect other party's data using the attacker capabilities we discussed in §4.1. A party can also observe and modify network traffic generated by such outsourced computation. Given the assumption of an abstract enclave model, OCQ guarantees integrity of outsourced operators and data.

Each party is free to input data of its choice, and OCQ does not protect against low-quality or maliciously-crafted data. OCQ ensures the integrity of each party's computation and data after the data has been inputted. The parties, if they wish to, could include checks for each other's data in the queries given to OCQ.

Query results often leak some information about the data from which they were computed. This is why we require the parties to agree on what queries they permit to run on their data and whose results they are agreeing to release. OCQ ensures that all parties agreed to running some query before running that query and releasing the result. Deciding which queries are safe to run is outside the scope of our system, and is largely an unsolved research problem. Nevertheless, existing work in differentially private analytics [8, 39, 40] (discussed further in §9.4) and inference detection [24, 34, 65] could aid parties to transform the queries into a safer form or to detect particularly revealing queries.

4.3 Security guarantees

OCQ guarantees obliviousness for sensitive tables, namely that there is no information leakage about the sensitive data content from the trace of memory accesses and traffic messages other than size, schema, and query information. We use a standard definition of obliviousness, which states that the *trace* of memory accesses and messages can be simulated without access to the data, while only knowing a bound on the data size, the schema, parties' configuration, and the query plans to execute.

The proof that OCQ meets this guarantee follows from the observation that for all our protocols (query planner, overall query plan, and individual operators), all the accesses to memory and the schedule of messages on the network are performed according to a *predefined schedule*, fixed ahead of time before the data is input, and which is determined according to our planner's rules based on data size, schema, query and party's configuration.

Like in much prior work in oblivious computation, our formalism captures only addresses and lengths, and not the actual data content. The reason is that hardware enclaves as used in OCQ and other works, encrypt the content and re-encrypt it upon every access.

We remark that OCQ's query planner runs entirely using non-sensitive information to produce a physical plan; hence, the planning process is oblivious by definition.

We now focus on showing that the execution of each physical plan of a query is oblivious w.r.t. sensitive tables. OCQ's definition of obliviousness differs from that of Opaque because there are multiple parties in addition to the network attacker. A malicious party attacker may see the content of its own sensitive tables and perform non-oblivious computation on them, but not the content of other parties' sensitive tables. We formalize this by constructing two types of simulators:

- $\text{Sim}_{\text{party}}$ has access to a party's sensitive data, but does not have access to the sensitive data of other parties, yet must simulate this party's memory and network access patterns.
- Sim_{net} does not have access to any party's sensitive data and must simulate the network communication patterns among all parties.

The fact that the simulators can simulate the memory and network patterns without seeing all the sensitive tables implies that these patterns do not leak information about these sensitive tables.

Let D be the set of tables of all parties. As we discussed in §3, tables are of two types: sensitive and public. Let S_i be the set of tables of party i that are sensitive (including those OCQ marks as sensitive after propagating sensitivity as discussed in §5). Let $\text{Public}(D)$ be all the non-sensitive information about the datasets, such as the content of all public tables and publicly known metadata about the sensitive tables, such as the names and owners of the tables, the names and schema of columns, an upper bound on the number of rows in each table, the set of unique key and foreign key constraints, and others. This metadata does not include the actual values in any column.

Let P be the set of publicly known cluster configurations of the parties, including the number of workers and their IP addresses, the untrusted memory size and EPC size of each machine, the oblivious sorting block size, etc. Let q be a query with any user-specified padding bounds (§5.5). Let $\text{Trace}_i(D, P, q)$ be the access pattern trace visible to the malicious attacker at party i: an ordered sequence of memory accesses of the form `(read/write, addr, length)` and network messages of the form `(dst, length)`. These do not contain timestamps because OCQ does not protect against timing attacks. $\text{Trace}_{\text{net}}(D, P, q)$ is the ordered sequence of network messages across all parties.

Theorem 4.1. *For all parties i, datasets D, with party i's sensitive tables S_i, for all cluster configurations P, for all queries q there exist polynomial-time simulators* $\text{Sim}_{\text{party}}$ *and* Sim_{net} *such that*

$$\forall i, \text{Sim}_{\text{party}}(\text{Public}(D), i, S_i, P, q) = \text{Trace}_i(D, P, q),$$
$$\text{Sim}_{\text{net}}(\text{Public}(D), P, q) = \text{Trace}_{\text{net}}(D, P, q).$$

We provide a proof sketch in §6.

5 Query Planning

We next explore OCQ's federated query planner, which finds query plans that satisfy security properties while minimizing the scope of expensive oblivious operators.

5.1 Overview

The federated query planner accepts sensitivity annotations on each party's base tables. It supports two sensitivity levels:

- *Public*: the table can be processed at any site with integrity verification. Confidentiality is not required, and data contents and cardinality may safely be exposed.
- *Sensitive*: if exported from the originating site, the table must be processed with confidentiality and integrity verification, including protection against access pattern side channels and cardinality leakage.

When executing a query, the planner propagates sensitivities through the plan to determine the sensitivity level of each intermediate result, such as a set of partial aggregates. OCQ uses the same two-part sensitivity propagation scheme as Opaque. First, since tables could be correlated via their relationships (e.g., foreign keys), OCQ uses second-path analysis [34] on the user-provided sensitivity annotations to capture these correlations: user-specified base tables are initialized as sensitive, and all tables reachable from a sensitive table via primary–foreign key relationships are recursively marked as sensitive as well. Second, OCQ assigns each operator a sensitivity level determined by the sensitivity levels of its inputs. Operators that process more than one input relation (e.g., join) receive the highest sensitivity level of all their inputs.

The federated query planner uses these two sensitivity levels to determine where to execute each operator. Recall that each party has an Opaque site/cluster. Operators may execute in two locations: federated or single-site. The federated operators execute in every site in parallel, whereas the single-site ones execute only at one party. For example, for an aggregation, each site can perform a partial aggregation within their site using the federated operator and the results are then sent to the querying party for final aggregation, which occurs using the single-site operator. OCQ supports the following operator execution modes:

- *Federated.* The operator executes partitioned and encrypted/authenticated in each site's Opaque cluster using SGX. Datasets are encrypted and authenticated for integrity verification, but operators will reveal data cardinality and may leak data contents through side channels.
- *Federated-Oblivious.* The operator executes partitioned and encrypted in each site's Opaque cluster using oblivious algorithms in SGX to hide access pattern side channels.

The operator reveals nothing beyond the cardinality of the input, such as filter and join selectivities.

- *Single-Site-Oblivious.* The operator executes obliviously with Opaque at the site where the query originated. This provides the same security guarantees as Federated-Oblivious and is used for final aggregation on sensitive data.
- *Single-Site.* The operator executes encrypted using Opaque at the site where the query originated. This provides the same security guarantees as the Federated mode and is used for final aggregation on public data.

The query planner ensures data of a particular sensitivity level is never processed using an operator with insufficient protections. For example, Sensitive data may be processed with a Federated operator if it has not left its originating site, but after an exchange, the same data must be processed using Federated-Oblivious or Single-Site-Oblivious operators only. The planner can always produce a secure plan for any supported query by running all operators in the Single-Site-Oblivious mode with naive worst-case padding, but the resulting overhead can be prohibitive so its goal is to find secure plans with much lower overhead when possible.

In the remainder of this section, we describe our query planning algorithm and rules (§5.2, §5.3), then discuss how the planner hides intermediate cardinalities (§5.4). The strict sensitivity levels and the use of padding together simplify physical operator selection. This traditionally depends on accurate cardinality estimation, which is much easier in OCQ than in traditional databases due to OCQ's use of padding. OCQ introduces padding rules that exploit foreign key constraints in the database schema to minimize padding overhead.

5.2 Algorithm

Respecting the constraints described above, the federated query planner applies rules to produce a physical plan that specifies the necessary algorithms and data movement for all three levels of the federation: between parties, within each party, and within each machine. Planning occurs as follows:

1. Obtain the logical plan for the query using Spark SQL.
2. Apply rules to transform the logical plan into a federated physical plan respecting sensitivity annotations and with "ShipTo" operators indicating data movement.
3. Eliminate redundant ShipTo operators.
4. Insert padding operators based on the rules in §5.4.
5. Execute the federated plan and return the results.

The rules used in step 2 apply recursively to the logical plan in bottom-up order starting with the base tables. For each logical operator (e.g., Project, Filter, Join, Aggregate) and its input plans, the rules produce a physical sub-plan that respects the operator's sensitivity level and avoids unnecessary encryption or data movement overhead.

OCQ's rules require operators to be planned in order of execution, because it uses the execution mode chosen for earlier operators in determining the execution mode for later operators. This is implemented using a postorder traversal on subplans referencing sensitive data.

5.3 Query planner rules

We now examine rules for the four common logical operators (Project, Filter, Join, and Aggregate). We express rules in Scala's pattern match syntax, where each rule is a `case` that may match the logical operator being planned. Rules can specify subtyping constraints with `:` syntax. We use these constraints to discriminate based on the execution mode (abbreviated as Fed, FedObl, SSObl, and SS) of an operator's inputs. Each rule ensures the resulting physical operator respects sensitivity levels, so the entire plan also will.

To illustrate how OCQ converts a query into a physical plan, take the example of Query 1 in Table 1. For each logical operator that matches a rule, Scala binds the operator's contents to parameters listed after the `case` keyword. It then executes the body of the case, which comes after the => arrow, and the planner uses the return value as the physical plan for the given logical operator. For example, the logical operator `Filter(diagnosis, diag="c. diff")` in Query 1 (Table 1), will be mapped to OCQ's physical operator `FedEncFilter[diag="c. diff"]`. We do not describe what each one of OCQ's physical operators run (because they are many), but instead describe only the more novel operator algorithms in §6.

Project. The input to the projection is referred to as `child`, and the projected columns as `p`. Because a projection never requires data movement and does not leak access patterns, the resulting operator uses the same execution mode as its input.

```
case Project(p, child: Fed) => FedEncProject(p, child)
case Project(p, child: FedObl) => FedOblProject(p, child)
case Project(p, child: SSObl) => OblProject(p, child)
case Project(p, child: SS) => EncProject(p, child)
```

Filter. The input to the filter is referred to as `child`, and the filter predicate as `f`. As with projection, filtering never requires data movement. However, unlike projection, filtering leaks access pattern information by default, so depending on the execution mode and sensitivity level of the input, we use an oblivious filter operator to hide which input records matched the predicate.

```
case Filter(f, child: Fed) => FedEncFilter(f, child)
case Filter(f, child: FedObl) => FedOblFilter(p, child)
case Filter(f, child: SSObl) => OblFilter(p, child)
case Filter(f, child: SS) => EncFilter(p, child)
```

Join. OCQ currently only supports inner joins. The inputs to the join are referred to as `left` and `right`, and the join columns as `c`. Joining does require data movement, and we choose between a broadcast join where one input is broadcast to all parties and joined separately with each portion of the other input, and a single-site join where both inputs are brought to the querier's cluster. Both types of joins can be executed with or without obliviousness; we additionally implement a mixed-sensitivity broadcast join (§6.1). For brevity,

we only list half the rules for Join; the other half are the same up to swapping left and right. When multiple join rules apply, such as when there is a choice between broadcasting the left side and the right side, the planner currently chooses one of them arbitrarily, but a cost-based planner could consider both and choose the lower-cost option. In addition, we benefit from Spark SQL's existing planner rules. For example, Spark SQL's broadcast exchange reuse ensures that when a plan indicates that the same relation should be broadcast more than once, it will only be shipped over the WAN once.

```
case Join(c, left: Fed, right: Public) =>
  FedEncJoin(c, left, BcastToFed(right))
case Join(c, left: Fed, right: Sensitive) =>
  FedMixedSensJoin(c, left, BcastToFed(right))
case Join(c, left: FedObl, right: Public) =>
  FedMixedSensJoin(c, left, BcastToFed(right))
case Join(c, left: FedObl, right: Sensitive) =>
  OblJoin(c, EncCollect(left), EncCollect(right))
case Join(c, left: SSObl, right: Public) =>
  MixedSensJoin(c, EncCollect(left), EncCollect(right))
case Join(c, left: SSObl, right: Sensitive) =>
  OblJoin(c, left, EncCollect(right))
```

Aggregate. The input to the aggregation is referred to as `child` (which is a query subplan), the grouping attributes as `g`, and the aggregation attributes as `a`. Partial and final aggregates are assigned the same sensitivity level as the input data. The attributes resulting from partial aggregation are referred to using `partial(a)` and the attributes from final aggregation as `final(a)`. The execution mode of partial aggregation is the same as its input, while final aggregation always occurs at the querier's site in SSObl or SS modes. The `with` keyword indicates the child's execution mode as well as its sensitivity.

```
case Aggregate(g, a, child: Fed with Public) =>
  EncAgg(g, final(a),
    EncCollect(FedAgg(g, partial(a), child)))
```

This rule applies to an aggregation over `Public` data which will reside in a `Fed` manner as a result of running the child subplan. Because the data is public, we can execute both the partial and the final aggregation in `Enc` mode (without oblivious operators).

```
case Aggregate(g, a, child: Fed with Sensitive) =>
  OblAgg(g, final(a),
    EncCollect(FedAgg(g, partial(a), child)))
case Aggregate(g, a, child: FedObl) =>
  OblAgg(g, final(a),
    EncCollect(FedOblAgg(g, partial(a), child)))
```

Both rules apply to sensitive federated data. In the first case, each sensitive data slice has not left its originating party, while in the second case, the sensitive data has been commingled with other parties' sensitive data and is protected by obliviousness in addition to encryption. Therefore, in the first case only the final aggregation needs to be oblivious, while in the second case, both the partial and final aggregation must be oblivious. The remaining rules are as follows:

```
case Aggregate(g, a, child: SSObl) => OblAgg(g, a, child)
case Aggregate(g, a, child: SS) => EncAgg(g, a, child)
```

Query planning example. Table 1 shows the results of OCQ's federated planning on two sample medical queries [7]. The queries refer to two tables: a `diagnosis` table containing patient SSNs and the diseases they were diagnosed with, and a `medication` table containing patient SSNs and the medications they were prescribed. Given these two relations, Query 1 computes comorbidity of the disease *c. diff*: the most common diseases that *c. diff* patients are also diagnosed with. Query 2 counts the number of patients with heart disease who were prescribed aspirin. Tables and intermediate results containing identifiable patient information (here SSN) were specified as Sensitive; tables without SSNs but with more than one column as Sensitive to prevent correlation attacks; and tables with only one non-identifiable column as Public.

For Query 1, the planner runs the initial Filter operator in Federated mode. Subsequently, only the `diag2` column of the result is needed for the Aggregate operator, so Spark SQL automatically inserts a projection to drop the other columns. The resulting table is collected to a single site for the final aggregation and sort.

For Query 2, the planner likewise runs the initial filters in Federated mode. As in Query 1, the right side is Public, so the planner uses a mixed-sensitivity broadcast join. The oblivious aggregation returns the number of distinct patients.

5.4 Determining padding upper bounds

Sensitive operators must pad their output to avoid leaking information about the input. For example, a bank may want to hide how many customers it has, so the data sizes of any cross-site communication must be padded to a bound greater than the number of customers. Alternatively, to hide the number of customers with revenue >$1 million, computation downstream of all revenue-based filters must be padded.

The federated query planner ensures queries' intermediate cardinalities do not leak information about sensitive attributes. The final cardinality is handled the same as the intermediate cardinalities, depending on which parties can see the final result. After producing an unpadded plan using the rules above, it adds padding to Sensitive operators that could leak information about the table's contents. This section describes this rule-based process using the same rule syntax as §5.3. Though these rules match physical operators, we use logical plan names in some pattern specifications to denote matching all physical operators implementing a logical operator. Additionally, we refer to certain specific join types such as referential integrity inner equi-joins as a shorthand for pattern specifications that check that these constraints are satisfied. Finally, we use Scala's `@` operator to denote binding the physical operator under consideration to a named variable.

Base tables. Base tables are padded using tiered padding to hide their exact cardinality, encapsulated by the `round` function, specified by the parties. We refer to the table scan

Query 1 (comorbidity of *c. diff*):

```
Sort(
  Aggregate(
    Join(
      Filter(diagnosis, diag="c. diff"),
      Project(diagnosis, diag as diag2),
      col="patientSSN"),
    groupby="diag2", agg="count"), col="count")
```

Federated plan for Query 1:

```
OblSort [diag_count]
  OblAgg groupby[diag2] agg[count(*) as diag_count]
    OblCollect
      FedOblProject [diag2]
        FedMixedSensJoin [patientSSN]
          BcastToFed
            FedEncFilter [diag="c. diff"]
              FedEncProject [diag]
                FedEncScan diagnosis
          FedEncProject [diag as diag2]
            FedEncScan diagnosis
```

Query 2 (aspirin count):

```
Count(
  Aggregate(
    Join(
      Filter(diagnosis, diag="heart disease"),
      Filter(medication, med="aspirin"),
      col="patientSSN"),
    groupby="patientSSN")
```

Federated plan for Query 2:

```
OblCount
  OblAgg groupby[patientSSN]
    OblCollect
      FedOblProject [patientSSN]
        FedMixedSensJoin [patientSSN]
          FedEncFilter [diag="heart disease"]
            FedEncScan diagnosis
          BcastToFed
            FedEncFilter [med="aspirin"]
              FedEncScan medication
```

Table 1. Example queries and resulting federated plans. Queries (top) are specified in Spark SQL's logical plan notation, which is similar to relational algebra. The resulting plans (bottom) are specified in Spark SQL's physical plan notation with OCQ's physical operator names. Nesting indicates a child relationship; sub-plans nested under a physical operator provide input to that operator. Physical operators take parameters, listed on the same line as the operator name. Most operator parameters are expression lists, listed in square brackets.

operator as `t`. The rule wraps `t` with a `Pad` operator that inserts dummy rows to inflate the result cardinality. The base table data must already be padded to an equal or greater bound when writing it to disk to avoid revealing the true cardinality through the file size. This enables this operator to make dummy accesses to the input when generating the dummy output rows.

```
case t @ TableScan() => Pad(round(t.cardinality), t)
```

Filters. Most filters must be padded to the input size to avoid leaking selectivity. The `else` clause of the rule below transforms the filter into a projection that adds a boolean field with the value of the filter predicate using Scala's `:+` operator, which appends this column to the existing columns. Later operators will use this tombstone-like field to determine whether or not to include the record in their operations.

When at most one record is being selected, such as to extract the record of a single patient based on SSN, padding to the input size would be very wasteful. OCQ identifies this case by checking if the predicate is an equality comparison against a unique key using the `uniquelyReferences` method. The `if` clause below pads the result to a cardinality of 1 to avoid leaking whether or not the predicate matched a row. Within this rule, which branch is taken depends only on the schema and query, not the underlying data, so it does not reveal any new information to an attacker.

```
case f @ Filter(pred, child) =>
  if (pred.uniquelyReferences(child.keys)) Pad(1, f)
  else Project(child.cols :+ pred, child)
```

Joins. Arbitrary joins must be padded to the product of table sizes, but common join types have much smaller upper bounds. Unique key equijoins are a very common join type where one side's join attribute is known unique. We pad these joins to the size of the other table; the unique key ensures each record in the latter matches at most one record in the former. For brevity, we omit the code listing for this rule.

Aggregations. Arbitrary aggregations must be padded to the input size, because each input record could belong to a different group. However, foreign key constraints let us refine this bound. For example, an aggregation over patients' diseases can yield at most one row per disease if there is a foreign key constraint between the (sensitive) patient diagnosis table and the (public) disease table, because the foreign key constraint implies that all patients' diseases correspond to a record in the disease table. Without the foreign key constraint, we must pad the aggregate output to the number of patient diagnoses, which could be large. With the constraint, we can instead pad to the number of diseases.

We search for such foreign key constraints using `publicTableKeys` in the rule below. If there is a matching foreign key, the `Some(...)` case determines the appropriate cardinality, using `min` to ensure the new bound is smaller. Otherwise, the `None` case pads to the input cardinality.

```
case a @ Aggregate(groupCols, aggExprs, child) =>
  publicTableKeys.find(groupCols.output) match {
    case Some(tbl, key) =>
      Pad(min(tbl.cardinality(key), child.cardinality), a)
    case None =>
```

```
    Pad(child.cardinality, a) }
```

5.5 Pre-specified padding bounds

Automatic padding bound search cannot always find the optimal bound. First, it assumes that while the exact cardinality of a base table is sensitive, rounded cardinalities are safe to share. Yet, for some tables, even an order-of-magnitude approximation of the cardinality is sensitive. Second, it relies on foreign key constraints to determine padding bounds for individual columns. However, many columns use implicit domains without foreign key constraints, such as ages, genders, letter grades, salaries, and ZIP codes. Third, most relational operators cannot reduce the padding bounds of their output, potentially resulting in large slowdowns.

We therefore support pre-specified padding bounds in schemas and queries. When defining the shared schema, parties can specify cardinality bounds for any table or column. When defining the allowed queries, a query may contain padding bounds for any intermediate result, for example the result of a filter expected to be highly selective. We exposed this functionality using extension methods on Spark SQL's DataFrame API. Below is an example of specifying table and column cardinality bounds and a bound for a filter operator.

```
val diseaseDF = spark.load(".../disease/")
  .sizeBound(70000)
val employeeDF = spark.load(".../employee/")
  .colBounds("salary" -> 20, "addressZIP" -> 42000)
val singleEmployee =
  employeeDF.filter($"name" = "John Doe").sizeBound(1)
```

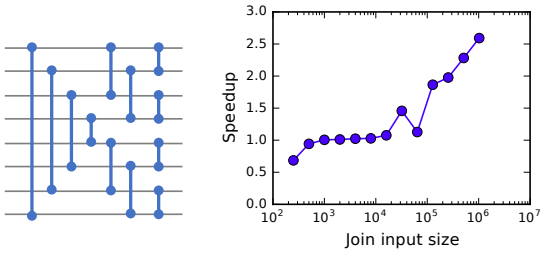
In case a query-specific operator underestimates the actual output cardinality, OCQ silently truncates the result to avoid leakage. If the parties wish to know when this has occurred, they can express this using subqueries.

6 Coopetitive algorithms

The coopetitive setting affects the performance of secure operators because it implies mixed-sensitivity computation and wide-area communication. Mixed-sensitivity computation occurs when a party combines its own data with sensitive data from another party. Wide-area communication occurs for operators such as aggregation and join that require combining data from multiple parties. We describe two algorithms that leverage the coopetitive setting to reduce the amount of oblivious computation needed compared to previous approaches.

6.1 Mixed-sensitivity join

Like Opaque's oblivious join algorithm, OCQ performs oblivious joins using bitonic sorting networks, which have traditionally been used in databases for SIMD parallelism [6] but which we use to protect against access pattern side channels. However, unlike in Opaque, oblivious joins in OCQ are very likely to be of mixed sensitivity. For example, a federated join between two Sensitive relations will be executed as an oblivious partial join at each party between that party's slice of one relation and the entire federation's records for the other



(a) Bitonic merge **(b)** Speedup from mixed-sensitivity join

Figure 5. (a) Bitonic merge network for 8 elements. (b) Speedup from mixed-sensitivity join compared to standard oblivious join for two relations of equal size. The x axis indicates the size of each relation in number of records. For small joins, other costs dominate, but once each relation contains more than 10^5 records, the mixed-sensitivity join shows a speedup of up to 2.5× compared to conventional oblivious join.

relation. In this case, it is unnecessary to handle the former relation obliviously, since the owner of that relation is also the party processing it.

We therefore introduce a mixed-sensitivity join algorithm. We refer to the former relation as the non-sensitive relation and the latter as the sensitive relation. First, the non-sensitive relation is sorted using a conventional external sort. Next, the sensitive relation is obliviously sorted using the algorithm described in §2.2. The two sorted relations are then merged using an oblivious bitonic merge, illustrated in Figure 5a.

When joining two relations of equal size, asymptotic analysis shows that mixed-sensitivity join represents a constant-factor improvement. If the two relations contain $\frac{n}{2}$ elements each, an oblivious sort of the union requires $O(n \log^2 n)$ comparisons, while the mixed-sensitivity algorithm requires $O(\frac{n}{2} \log \frac{n}{2})$ comparisons for the conventional external sort, $O(\frac{n}{2} \log^2 \frac{n}{2})$ comparisons for the sensitive-relation oblivious sort, and $O(\frac{n}{2} \log n)$ comparisons for the bitonic merge. Figure 5b demonstrates an empirical speedup due to mixed-sensitivity join of up to 2.5× for large inputs. In addition, when the sensitive relation is relatively small, mixed-sensitivity join becomes arbitrarily faster than a standard oblivious join.

6.2 Coopetitive aggregation

Opaque implements aggregation using an initial distributed oblivious sort based on the grouping attributes to colocate records that belong to the same group. However, in the coopetitive setting, this results in excessive wide-area data movement. Instead, we implement aggregation by computing partial aggregates at each party and sorting only those partial aggregates across the wide area, with padding to hide the exact number of groups from each party. The partial aggregates from each party are padded as described in Sections 5.4 and 5.5. The final aggregation is then performed as in Opaque, with a boundary processing step, a parallel scan over the sorted partial aggregates to produce final aggregates

and dummy records, and, if a user-specified padding bound on the output is provided, an oblivious sort and filter to remove the appropriate number of dummies.

The speedup from this approach comes from avoiding the initial global oblivious sort in favor of an oblivious sort over the partial aggregates. When the bound of the number of groups is small, this produces a substantial performance gain.

6.3 Obliviousness proofs

Now that we presented OCQ's algorithms, we proceed to sketch the proof of its obliviousness, formulated in §4.3.

Proof of Theorem 4.1. In this proof, we will invoke the simulators for the oblivious building blocks that previously existed: bitonic sort, bitonic merge, column sort primitives, and the Opaque operators in oblivious-pad mode. A query q can consist of different tasks. It suffices to prove that the simulators can simulate the trace for each task. Here, we present simulators for the more complex algorithms OCQ contributes: mixed-sensitivity join and and coopetitive aggregation. For each physical operator O in the physical plan, the planner rules ensure that O runs in oblivious mode if the inputs to O contain any sensitive table owned by a party other than i.

Without loss of generality, consider party i. Recall that mixed-sensitivity join occurs between party i's own input A_i and a slice of the other parties' input B_i. If O is a mixed-sensitivity join, it proceeds as follows:

1. $\text{Sim}_{\text{party}}$ sorts A_i using quicksort because it receives A_i as input. Let $\text{Public}(A_{i,\text{sorted}})$ be the public metadata of the sorted result.
2. $\text{Sim}_{\text{party}}$ invokes the simulators for bitonic sort and column sort on $\text{Public}(B)$ to simulate an oblivious distributed sort of B on the equijoin keys. Let $\text{Public}(B_{i,\text{sorted}})$ be the metadata of the sorted result.
3. $\text{Sim}_{\text{party}}$ invokes the simulator for bitonic merge on $\text{Public}(A_{i,\text{sorted}})$ and $\text{Public}(B_{i,\text{sorted}})$. Let $\text{Public}(U_{i,\text{sorted}})$ be the metadata of the sorted union.
4. $\text{Sim}_{\text{party}}$ invokes Opaque's simulator for oblivious padded join on $\text{Public}(U_{i,\text{sorted}})$ with the padding bound specified in the query.

Sim_{net} runs similarly to $\text{Sim}_{\text{party}}$: it performs the last three steps above.

For coopetitive aggregation, if O is the partial phase of coopetitive aggregation at party i, let the padding bound for O be b. $\text{Sim}_{\text{party}}$ for party i extracts the data of the sensitive input A_i and executes a conventional hash aggregation on it, then pads its size to b. Let $\text{Public}(\text{Agg}(A_i))$ be the metadata of the padded partial aggregates.

If O is the final phase of coopetitive aggregation, $\text{Sim}_{\text{party}}$ for party i and Sim_{net} invokes the simulator for the bitonic sort and column sort on $\bigcup_{\forall i} \text{Public}(\text{Agg}(A_i))$. Sim_{net} then invokes Opaque's simulator for the oblivious padded aggregation on this metadata.

□

7 Implementation

We implemented OCQ on top of Intel SGX and as an extension to Apache Spark SQL's Catalyst query planner and execution engine using 2,000 lines of Scala code. OCQ builds on a version of Opaque we modified, which uses 11,000 lines of C++ enclave code and 3,000 lines of Scala code. No code changes to Spark SQL were required; both OCQ and Opaque extend Catalyst only by adding rules and strategies. The schema-aware padding requires that tables be annotated with primary and foreign key hints; since Spark SQL does not natively support key annotations, we added them as a separate extension. We implemented the row-level conditional-exchange primitive in SGX using the x86 conditional move instructions on data in registers, for which we assume that the attacker cannot see accesses to registers inside enclaves. Federated query execution is coordinated by the querying JVM, which maintains a connection to each cluster in the federation using Spark's remote query functionality via the SparkSession.

8 Evaluation

In this section we evaluate OCQ's performance against outsourced and multi-party computation. We measure the overhead of OCQ's security guarantees. We explore an alternative design where SGX-enabled machines are required at only one site, and show that requiring SGX at all sites provides a significant speedup. We explore the speedup of our query planner compared to a traditional query planner unaware of security. Finally, we evaluate the benefit of schema-aware padding versus the conventional filter push-up approach.

8.1 Setup

We performed benchmarks across 5 parties located in AWS us-east-1, AWS us-west-1, AWS eu-west-1, and AWS ap-northeast-1, and in our organization. Each party has approximately 10 MB/s bandwidth to each other party. Our organization's site has an SGX cluster with 5 machines, while the AWS sites use 5-node `r5.large` clusters with Intel's SGX simulation driver. We use a federated query workload derived from previous papers on federated analytics. From SMCQL [7] we use the comorbidity and aspirin count queries described in §5.3. From DJoin [57] we use queries 1–5. The DJoin queries are listed in Table 2. We generated synthetic table data with the following total size per table:

1. `diagnosis` - 1,024,000 rows, 10 GB
2. `medication` - 142,972 rows, 4.3 MB
3. DJoin `A` - 15,000 rows, 15 MB
4. DJoin `B` - 15,000 rows, 15 MB

8.2 Comparison to other systems

Figure 6 compares OCQ to Opaque (outsourced computation) as well as SMCQL and DJoin (secure multi-party computation). For the first, since Opaque's implementation assumes at least 2MB of non-observable memory which speeds up its oblivious protocols considerably, we ran OCQ with the

Q1
```
Count(GroupBy(Join(A, B, "x" == "y"), "x"))
```
Q5
```
Count(GroupBy(Filter(
  Join(A, B, "w"),
  Contains("A.x", "xyz")
    && ("B.x" + "B.y" > 10)
    && ("A.y" > "B.y")), "x"))
```

Table 2. DJoin queries.

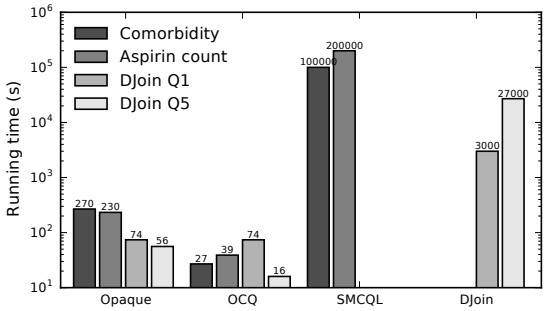


Figure 6. OCQ vs. competing systems. OCQ is orders of magnitude faster than SMCQL and DJoin due to its use of trusted hardware, and is faster than Opaque for most queries because it can execute initial filters in plaintext.

same configuration. For the second, we report the numbers from the SMCQL and DJoin papers because they are too slow to run on our dataset or not open source. They are also insufficient for the coopetitive setting (as explained in §9.1). OCQ is 1–4 orders of magnitude faster than SMCQL and DJoin due to its use of trusted hardware. Meanwhile, OCQ gains a performance advantage over Opaque, which also uses trusted hardware, for queries that begin with a substantially selective filter operation. Because the initial filter requires no communication between parties, it can be executed in plaintext at each party. The cardinality of the input to subsequent oblivious operators is then greatly reduced. For our synthetic data, the initial *c. diff* filter for the comorbidity query has $\approx 1\%$ selectivity, as does the result of the selection and join to find patients with heart disease who were prescribed aspirin for the aspirin count query. For OCQ we specified padding bounds that reflect this selectivity, because we do not wish to treat the selectivity parameter as sensitive. In contrast, Opaque must first perform an oblivious filter over each full relation. Because intermediate relation sizes within our query workload tend to shrink as the query progresses, this initial oblivious filter tends to dominate the running time.

Our reported query times include network transfer time, including the time required to transfer the full relations to the cloud for each Opaque query. Figure 7 shows the proportion of query time spent in network transfer. These transfers occur in parallel, so 5-node cluster uses its full aggregate bandwidth to transfer each relation. For the medical queries, this transfer is the dominant factor in Opaque's query times due to the large data sizes involved. Although the uploaded data could

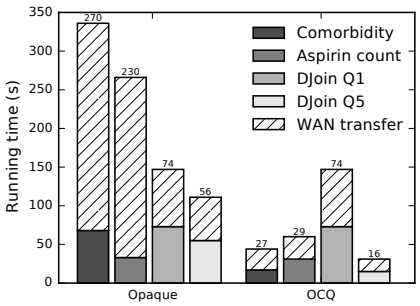


Figure 7. OCQ vs. Opaque, highlighting network transfer time. OCQ retains an advantage even assuming an infinitely fast network because it can execute initial filters in plaintext rather than using oblivious operators on the full inputs.

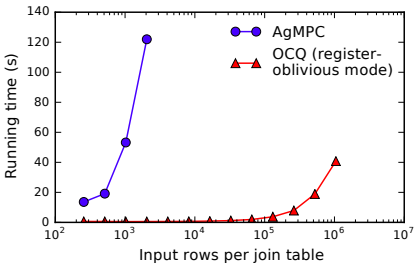


Figure 8. Performance of register-oblivious OCQ versus AgMPC. OCQ provides the same level of memory access pattern protection as AgMPC, but scales much better and is up to 219x faster.

be reused across queries, we include it in the query times for consistency with the other systems. Data reuse is not always possible, for example in case of frequently-changing data.

We also evaluated OCQ without any non-observable memory assumption (namely, the attacker can observe any party table data in any part of memory) to show that its performance remains much better than MPC-based systems. We compared OCQ against AgMPC [75], a state-of-the-art maliciously secure MPC framework (§9.1). We ran a query consisting of a referential integrity inner equi-join on two equal-sized synthetic tables, each containing two 32-bit integers. The first integer in each table was the join key. Figure 8 shows that OCQ is up to 219× faster than AgMPC on this query.

8.3 Overhead of security

Figure 9 compares OCQ to alternative uses of Opaque with varying security guarantees to show the overhead of OCQ's security. "Outsourced Opaque" is as before. "Plaintext federated" refers to an alternative federated configuration where all computation runs in plain text rather than within SGX. This configuration might be suitable for a network of non-competing entities. "Outsourced Spark SQL" refers to a configuration similar to Opaque but where computation is run in plaintext rather than in SGX. This option provides no security guarantees, as a server-side attacker could access the data in full.

Figure 9 shows that OCQ introduces 2.2–12× overhead compared to a plaintext federated configuration due to its

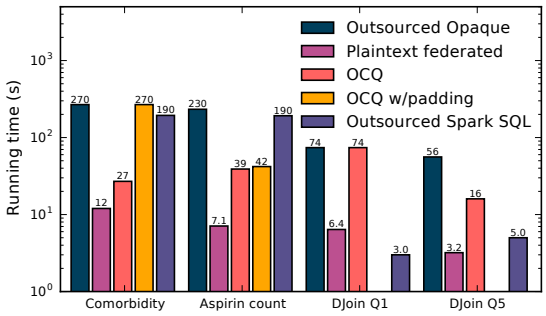


Figure 9. Overhead of OCQ's security. OCQ incurs 2.2–25× overhead compared to the federated and outsourced Spark SQL baselines, which provide no security.

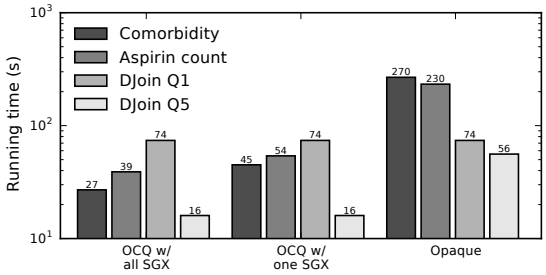


Figure 10. Availability of SGX enclaves at all sites provides 1.3–1.6× speedup compared to having SGX at only one site.

use of oblivious operators. Additionally, outsourced Spark SQL outperforms OCQ by up to 25× for the DJoin queries, which use small relations without any initial filters, and whose complexity is entirely in the core join computation, which is expensive to perform obliviously. However, OCQ outperforms the outsourced configurations for the medical queries, which do contain initial filters. Note that as before, we include network transfer time in query time, including the time to upload full tables for the outsourced configurations.

8.4 Benefit of having SGX at each site

We next explore the design choice in OCQ that each site must have its own local SGX cluster. This represents a constraint to its adoption, since SGX deployments are not widespread. However, we observe that some queries significantly benefit from this choice. Figure 10 compares OCQ to an alternative design ("OCQ w/ one SGX") where operations on multiple parties' data such as joins and aggregations require the data to be collected to a single SGX cluster first. We observe a 1.3–1.6× speedup over the alternative design for the medical queries because the use of broadcast joins allows the components of the broadcast join, namely an oblivious join at each party, to operate on less data. Since all oblivious joins run in parallel, this results in a speedup for the initial join, which dominates the query plans.

8.5 Benefit of schema-aware padding

We compare the performance of our schema-aware padding approach to a baseline query without padding and to an implementation of Opaque's "filter push-up" approach. In the

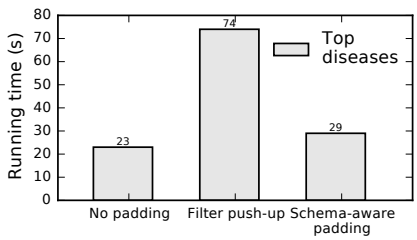


Figure 11. Schema-aware padding provides a 2.5× speedup compared to filter push-up, and is only 26% slower than the baseline plan without padding.

latter, filters and aggregations on sensitive relations always return one output row for each input row, with rows that did not match the predicate or rows other than the first one in each group resulting in a dummy output row. All dummy rows are filtered out at once as the final step of the query, thus hiding intermediate result sizes. This approach results in large costs when intermediate output is transferred over the network.

With appropriate sensitivity step-down hints for the medical dataset, schema-aware padding on the aspirin count and comorbidity queries results in the same plan as a suitably designed filter push-up approach. We therefore introduce a plausible new query on the medical dataset that finds the most costly diseases by grouping the `diagnosis` table on disease id and computing the total cost for each disease:

```
diagnosis.groupBy("disease")
  .agg(sum("cost") as "total_cost")
  .sort("total_cost").select("disease").take(5)
```

This query results in the physical plan without padding:

```
OblLimit 5
  OblProject [disease_id]
    OblSort [total_cost]
      OblAgg groupby[disease_id]
             agg[sum(cost_partial_sum) as total_cost]
        OblCollect
          FedAgg groupby[disease_id]
                 agg[sum(cost) as cost_partial_sum]
            FedEncScan diagnosis
```

Though the top-level aggregation occurs within SGX because its input is sensitive, this plan may leak the cardinality of the partial aggregates through SGX side channels. Our reimplementation of filter push-up results in the following:

```
OblLimit 5
  OblProject [disease_id]
    OblSort [total_cost]
      OblAggPadded groupby[disease_id]
                   agg[sum(cost_partial_sum) as total_cost]
        OblCollect
          FedPad diagnosis.cardinality
            FedAgg groupby[disease]
                   agg[sum(cost) as cost_partial_sum]
              FedEncScan diagnosis
```

This plan pads both levels of aggregation to the input size, hiding their cardinalities, but resulting in two costly oblivious sort operations on padded data. In contrast, our schema-aware padding recognizes that the cardinalities of the partial and final aggregates are bounded by the number of known disease codes, which is much smaller than the cardinality of the input `diagnosis` table and is public knowledge. It generates a nearly identical plan to filter push-up, with the difference that both aggregation operations pad to `disease.cardinality` instead of `diagnosis.cardinality`. Figure 11 compares the performance of all three plans using the input described in §8.1. The lower padding bound and consequently reduced oblivious sort cardinality give our schema-aware padding approach a 2.5× speedup compared to the filter push-up approach, and it is only 26% slower than the baseline plan without padding.

9 Related work

9.1 Cryptographic approaches

SMCQL [7] and Conclave [73] use secure multi-party computation (instead of hardware enclaves) to achieve federated analytics queries. Unlike OCQ, SMCQL and Conclave do not protect against a malicious attacker (the attacker is semi-honest), and their implementation is for only two or three parties. Further, their join scheme relies on joining non-sensitive attributes, unlike OCQ, which can join on sensitive data.

AgMPC [75] is likely the most relevant cryptographic framework to OCQ because it is n-party, maliciously-secure, and supports generic computation. As we show in §8, though, AgMPC is orders of magnitude slower than OCQ.

DJoin [57] and private intersection-sum [37] use multi-party computation to provide certain SQL operators. DJoin supports only count queries over equi-joins, while private intersection-sum supports sum queries over set intersections. Both assume a passive attacker and incur high overhead.

UnLynx [29] and MedCo [66] use partially-homomorphic encryption to provide filter-aggregate queries over multiple parties' data and are secure against malicious queriers. Compared to these systems, OCQ offers a greater range of functionality, but requires trust in hardware enclaves.

Cryptographic approaches have also been used for coopetitive machine learning training and prediction, which often involve secure aggregation [11, 21, 30, 33, 55, 58]. These approaches do not offer general analytics, and they largely assume a passive attacker or two non-colluding servers.

Encrypted databases such as CryptDB [64], AlwaysEncrypted [52], and Seabed [62] perform queries over encrypted data, but are not suited for the coopetitive setting.

9.2 Hardware enclave approaches

Systems for single-machine or distributed computation using hardware enclaves include SCONE [5], Graphene [18], Ryoan [36], Haven [9], VC3 [67], Cipherbase [2], and Opaque [81]. These systems assume a setting where all data is controlled by one party and are not designed for the coopetitive setting. Prochlo [10] offers privacy-preserving outsourced computation over many users' data using hardware enclaves. However, it requires centralizing the data, which encounters regulatory and logistical challenges in the coopetitive setting. Ohrimenko et al. [60] provides oblivious machine learning in SGX, but does not consider analytics in the federated setting and query planning. Oblix [53] focuses on oblivious point queries and does not support analytics and the decentralized setting.

9.3 Unencrypted federated databases

Collaborative query planning (CQP) [80] is a proposal for decentralized query planning in a multi-party setting where information sharing policy restricts centralized planning. Queries are instead broken into subqueries, independently planned by each party, and reassembled into a federated plan. CQP shares a setting with OCQ and complements it in the case where query planning information such as statistics must be treated as sensitive. Preference-aware query optimization (PAQO) [27, 28] is a proposal to extend SQL with users' declarations for where their data should be processed in a distributed database, with applications including restricting certain data from untrusted servers. PAQO allows the user to treat these declarations, called intensional descriptions [27], as preferences to be optimized rather than hard constraints. In future work, a similar approach could be applied to OCQ.

9.4 Differential privacy

A complementary and synergetic direction to OCQ are differential-privacy systems like Flex [39] and Chorus [40], which offer differential privacy for SQL queries via query rewriting. The queries they produce can be used as input to OCQ. Hence, one could add differential privacy to a query result before sharing it among the parties in OCQ. Shrinkwrap [8] provides differential privacy specifically for federations, and can be used with OCQ to reduce the amount of padding for intermediate results.

10 Conclusion

In this paper we proposed OCQ, an efficient framework for oblivious coopetitive analytics using hardware enclaves. OCQ's contributions are its query planner design, which supports flexible party-specific sensitivity rules, its mechanism for propagating and refining padding upper bounds based on foreign key constraints, and its mixed-sensitivity algorithms.

Acknowledgements

We are grateful for the helpful comments from the anonymous reviewers and from our shepherd, Manuel Costa. This work has been supported by NSF CISE Expeditions Award CCF-1730628, as well as gifts from the Sloan Foundation, Bakar, Alibaba, Amazon Web Services, Ant Financial, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. 2000. Privacy-preserving data mining. *SIGMOD Rec.* (2000).

[2] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramaratnam Venkatesan. 2013. Orthogonal Security with Cipherbase. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA.

[3] ARM. [n.d.]. TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone.

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[6] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* (2013).

[7] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: Secure Querying for Federated Databases. *Proc. VLDB Endow.* (2017).

[8] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: Efficient SQL Query Processing in Differentially Private Data Federations. *Proc. VLDB Endow.* 12, 3 (2018).

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[10] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[11] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[12] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *IEEE/ACM Internationl Symposium on Microarchicture*.

[13] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*.

[15] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.

[16] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX SECURITY*.

[17] Centers for Medicare & Medicaid Services. 1996. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at http://www.cms.hhs.gov/hipaa/.

[18] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*.

[20] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. 2018. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[21] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[22] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. Usenix Security Symposium. https://eprint.iacr.org/2015/564.

[23] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*.

[24] Harry S. Delugach and Thomas H. Hinke. 1996. Wizard: A Database Inference Analysis and Detection System. *IEEE Transactions on Knowledge and Data Engineering* (1996).

[25] Erez Eizenman. 2019. Scotiabank's chief risk officer on the state of anti-money laundering. In *McKinsey Company*.

[26] Saba Eskandarian and Matei Zaharia. 2017. An Oblivious General-Purpose SQL Database for the Cloud. *CoRR* abs/1710.00458 (2017). arXiv:1710.00458 http://arxiv.org/abs/1710.00458

[27] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. 2011. Don't Reveal My Intension: Protecting User Privacy Using Declarative Preferences During Distributed Query Processing. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS'11)*.

[28] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. 2014. PAQO: Preference-aware query optimization for decentralized database systems. In *2014 IEEE 30th International Conference on Data Engineering*.

[29] David Froelicher, Patricia Egger, João Sá Sousa, Jean Louis Raisaro, Zhicong Huang, Christian Vincent Mouchet, Bryan Ford, and Jean-Pierre Hubaux. 2017. UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. *Proceedings on Privacy Enhancing Technologies* 4 (2017), 152–170. http://infoscience.epfl.ch/record/229308

[30] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. 2017. Privacy-Preserving Distributed Linear Regression on High-Dimensional Data. *PoPETs* 2017, 4 (2017), 345–364.

[31] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*.

[32] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of rowhammer defenses. In *IEEE Symposium on Security and Privacy (SP)*.

[33] Rob Hall, Stephen E. Fienberg, and Yuval Nardi. 2011. Secure Multiple Linear Regression Based on Homomorphic Encryption.

[34] Thomas H. Hinke. 1988. Inference aggregation detection in database management systems. In *Proceedings. 1988 IEEE Symposium on Security and Privacy*.

[35] Tyler Hunt, Zhipeng Jia, Vance Miller, Christopher J. Rossbach, and Emmett Witchel. 2019. Isolation and Beyond: Challenges for System Security. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*.

[36] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association.

[37] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. 2017. Private Intersection-Sum Protocol with Applications to Attributing Aggregate Ad Conversions. Cryptology ePrint Archive, Report 2017/738. https://eprint.iacr.org/2017/738.

[38] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*.

[39] Noah Johnson, Joseph P. Near, and Dawn Song. 2018. Towards Practical Differential Privacy for SQL Queries. *Proc. VLDB Endow.* (2018).

[40] Noah M. Johnson, Joseph P. Near, Joseph M. Hellerstein, and Dawn Song. 2018. Chorus: Differential Privacy via Query Rewriting. *CoRR* abs/1809.07750 (2018). arXiv:1809.07750 http://arxiv.org/abs/1809.07750

[41] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. White paper.

[42] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. 2005. Secure regression on distributed databases. *Journal of Computational and Graphical Statistics* (2005).

[43] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2019. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. arXiv:cs.CR/1912.01701

[44] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *European Conference on Computer Systems (Eurosys)*. https://keystone-enclave.org/.

[45] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*.

[46] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX SECURITY*.

[47] Tom Leighton. 1984. Tight bounds on the complexity of parallel sorting. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM, 71–80.

[48] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. 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)*.

[49] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*.

[50] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

[51] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. 2016. Federated Learning of Deep Networks using Model Averaging. *CoRR* abs/1602.05629 (2016). arXiv:1602.05629 http://arxiv.org/abs/1602.05629

[52] Microsoft. 2019. Always Encrypted Database Engine. https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[53] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (SP)*. 279–296.

[54] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* (2019).

[55] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[56] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[57] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially Private Join Queries over Distributed Databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*.

[58] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *2013 IEEE Symposium on Security and Privacy*.

[59] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. 2015. Observing and Preventing Leakage in MapReduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*.

[60] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium (USENIX Security 16)*.

[61] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[62] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big Data Analytics over Encrypted Datasets with Seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[63] Bryan Parno, Jay Lorch, John (JD) Douceur, James Mickens, and Jonathan M. McCune. 2011. Memoir: Practical State Continuity for Protected Modules. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[64] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*.

[65] Xiaolei Qian, Mark E. Stickel, Peter D. Karp, Teresa F. Lunt, and Thomas D. Garvey. 1993. Detection and elimination of inference channels in multilevel relational database systems. In *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, 196–205.

[66] Jean Louis Raisaro, Juan Troncoso-Pastoriza, Mickael Misbach, Joao Sa Sousa, Sylvain Pradervand, Edoardo Missiaglia, Olivier Michielin, Bryan Ford, and Jean-Pierre Hubaux. 2019. MedCo: Enabling Secure and Privacy-Preserving Exploration of Distributed Clinical and Genomic Data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2019).

[67] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015.

VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society.

[68] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[69] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clementine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.

[70] Michael Stonebraker, Paul M Aoki, Robert Devine, Witold Litwin, and Michael Olson. 1994. Mariposa: A new architecture for distributed data. In *Data Engineering, 1994. Proceedings. 10th International Conference*.

[71] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*.

[72] Komodo: Using verification to disentangle secure-enclave hardware from software. 2017. Andrew Ferraiuolo and Andrew Baumann and Chris Hawblitzel and Bryan Parno. In *SOSP*.

[73] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: Secure Multi-party Computation on Big Data. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*.

[74] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*.

[75] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-Scale Secure Multiparty Computation. Cryptology ePrint Archive, Report 2017/189. https://eprint.iacr.org/2017/189.

[76] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. Springer.

[77] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEESP*.

[78] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* (2017).

[79] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.

[80] Mingyi Zhao, Peng Liu, and Jorge Lobo. 2015. Towards Collaborative Query Planning in Multi-party Database Networks. In *Data and Applications Security and Privacy XXIX*, Pierangela Samarati (Ed.).

[81] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[82] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2019. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*.