SMCQL: Secure Query Processing for Private Data Networks

Johes Bater
Northwestern University
johes@u.northwestern.edu

Gregory Elliott
Northwestern University
GregoryElliott2016@u.northwestern.edu

Craig Eggen
Northwestern University
CraigEggen2016@u.northwestern.edu

Satyender Goel
Northwestern University
s-goel@northwestern.edu

Abel Kho
Northwestern University
Abel.Kho@nm.org

Jennie Duggan
Northwestern University
jennie.duggan@northwestern.edu

ABSTRACT
People and machines are recording data at an unprecedented rate. At the same time, progress has been slow in making data available for open science and other research initiatives. Many of these efforts are stymied by privacy concerns and regulatory compliance issues. For example, many hospitals are interested in combining their patient records with those of other healthcare sites for clinical data research, but they cannot disclose the contents of their databases without violating patient confidentiality. We propose a novel generalization of federated database systems called a private data network (PDN), and it is designed for querying over the collective data of mutually distrustful parties. In a PDN, participants do not reveal their raw data, nor do they encrypt and upload it to the cloud. Rather, they perform secure multiparty computation (SMC) with other federation members to produce query results over the data of both parties. In this setting, a user submits their SQL query to an honest broker that plans and coordinates its distributed execution using SMC.

Within SMC, the participating database providers compute a joint function with an output that is only revealed to the user and the honest broker. Thus the databases computing the query learn nothing about the inputs provided by their peers, nor can they see the output of the group’s computation. This capability comes at a high cost—SMC programs typically have runtimes that are orders of magnitude slower than their insecure counterparts. This performance has created a barrier to entry for oblivious computing in the past. We address this challenge with a query planner that automatically identifies the minimal set of coordination points between parties in a given query plan. The planner translates these distributed steps into SMC as needed and feeds the secure code into our query executor. Our framework, SMCQL, plans and executes PDN queries. We are preparing SMCQL for an open-source release.

1. INTRODUCTION
Federated database systems, wherein many autonomous databases are united to appear as a single engine for querying, are seeing a renaissance in “big data” applications. Interestingly, many of these federations contain data sources that are owned by mutually distrustful parties that are interested in having the union of their data analyzed but do not want to reveal individual tuples of their private data. We call a database federation that spans multiple, mutually distrustful sources a private data network or PDN. We have identified use cases for PDNs in medical research, data markets, banking, online advertising, and human rights work. Traditionally, PDNs have either chosen not to share their data or they used one-off privacy preserving algorithms to mine their data—such as one for linear regression \(^{13}\) or a decision tree classifier \(^{3}\). Instead, we posit that users will more widely deploy PDNs if they are permitted to express their analytics as arbitrary SQL queries. In other words, our framework enables PDN users to issue queries over the collective data of mutually distrustful parties without needing to explicitly program how the data of individual parties is accessed and combined.

We first introduce an architecture for PDNs. Here, a user submits their query to the federation’s honest broker, a neutral third-party that plans and orchestrates its execution over the private data providers. The honest broker coalesces the results from its data providers and ships them back to the user. Neither the broker nor any of the data providers are permitted to view private data other than their own. A PDN has an agreed-upon schema shared by all providers. The schema’s attributes are each subject to a security policy that ranges from public, or data that can be viewed by all parties, to private, data that is never divulged outside of its originator.

We execute queries within a PDN using secure multiparty computation (SMC), a subfield of cryptography that creates protocols for two or more parties to jointly compute a function over their data while keeping their individual inputs private. This topic was first studied for two-party computation with Yao’s Millionaire Problem \(^{27}\). He asked, “If Alice and Bob are both millionaires who are interested in determining which one of them is richer, how can they solve for this without either party revealing their net worth?” Informally speaking, the parties calculate their answer using a garbled circuit, or a collection of logic gates (e.g., AND, OR) that compute over encrypted bits. Garbled circuits have proven quite expressive, and can be used to compute any arbitrary function \(^{6}\). This approach has been extended for use in three or more parties \(^{10}\). The garbled circuits in our approach support the semi-honest security model.

Our framework for planning and executing PDN queries is called SMCQL. It automatically translates operators in a
query execution plan into SMC as needed to preserve privacy. In SMCQL, PDN queries use garbled circuits when they are evaluating steps in a query plan that compute over sensitive data from two or more databases. In the prototype presented in this paper, we implement a query planner and executor for two mutually distrustful parties.

This query rewrite framework makes PDN queries execute efficiently with a fine-grained partitioning of the query’s execution plan and its data. We partition the plan by identifying when it requires distributed computing over sensitive data. In addition, we opportunistically slice the data, making it possible to parallelize computation and to reduce the complexity of our SMC circuits.

SMCQL is a substantial departure from previous approaches to secure databases. One existing method of protecting data is to inject controlled levels of noise into query output using differential privacy [8]. In SMCQL, we offer precise query results but only admit queries that conform to a PDN’s security policy. Another technique is to outsource the storage and querying of private data by protecting it with homomorphic encryption [24]. In contrast, our approach is designed to keep data in the hands its originators and using garbled circuits to obfuscate the query’s computation rather than the data itself.

There are numerous existing approaches to making SMC available to untrained users including domain-specific programming languages [15, 25, 20] and extending existing languages [28]. Our approach differs from them in that we do not require the user or programmer to reason explicitly about how to compute over the data of each party in SMC. Instead we translate SQL queries directly into garbled circuits. To the best of our knowledge, this is the first time researchers have combined SMC with an existing language without adding domain-specific extensions.

Our main contributions in this work are:

- Formally defining an architecture and data model for private data networks that enables a federated database to automatically run queries over mutually distrustful parties.
- Creating a query translation engine that automatically generates SMC circuits for relational database query plans.
- Designing and implementing a query rewrite framework for optimized mixed computation between plaintext and SMC.
- A thorough evaluation on real-world data of SMCQL’s query rewrite and execution framework.

The rest of this paper is organized as follows. In Section 2, we describe the basics of SMC and introduce a running example for this work. Section 3 details the architecture of a PDN and its privacy model. After that, we explore SMCQL’s query compilation framework, including how it identifies and translates the minimal set of database operators needed to compute a SQL statement and obey the security requirements of a PDN. After that, we show our experimental evaluation over real-world medical data. Lastly, we survey the relevant literature and conclude.

2. BACKGROUND

At its core, SMCQL offers a query planner and executor that automatically generates and runs SMC over mutually distrustful parties to compute a SQL statement over the collective data of the parties. The executor uses an existing set of SMC primitives. In this section, we briefly describe these building blocks and give an intuition about how they work. Throughout the text, we will refer to our mutually distrustful data providers or parties as Alice and Bob for clarity.

Garbled Circuits For query processing that requires secure coordination between Alice and Bob, we generate and compile garbled circuits for the relevant database operators. Garbled circuits are a series of logic gates (e.g., AND, XOR) that obfuscate the execution trace of an oblivious program. In other words, garbled circuits use the same distribution of execution time, regardless of their inputs, to make it impossible for a party to learn about the inputs of others. In our framework, the only knowledge available to the data providers is the number of tuples provided by each party. To prevent information leaks, the PDN architecture has a security policy so that the honest broker controls the conditions under which a query is admitted for execution.

Each garbled circuit securely computes a function, such as $x < y$ in Yao’s Millionaire Problem. As we compute functions with a large series of logic gates, in order to not reveal the inputs of Alice and Bob, we have a blow up in the compute time of a garbled circuit. For example, if we were computing $x < y$ in plaintext, we’d find the most significant bit where Alice and Bob’s inputs differ and use that to get our answer. A naive garbled circuit must instead compare all the bits in order to not disclose the one that produced the result. Similarly, for if-then statements, a SMC program must execute both branches of the statement in order to stay oblivious to the values in the input data.

For boolean circuits such as ours, the garbled function is computed on, and produces, one or more shared secret bits (SSBs) or boolean labels. Thus at the beginning of a circuit, each party selects one of two SSBs to denote whether their input bit is zero or one. The parties determine their inputs using oblivious transfer. Here, the sender (Alice) has two SSB labels, $s_0$ and $s_1$, and the receiver (Bob) has a bit $b$. Bob wants access to $s_b$, without disclosing $b$ to Alice. At the same time, Alice wants to ensure that the receiver receives only one of the two labels. The output of a garbled circuit is another SSB, computed based on the inputs selected by oblivious transfer. Taken alone by each party, each output SSB has no intrinsic meaning. If we combine the SSBs of Alice and Bob as the output of a SMC function, we produce the plaintext result of the computation. See [7] for more background on garbled circuits. In our setting, we permit neither Alice nor Bob to put together two halves of the shared secrets produced by a SMCQL query. Instead, only the honest broker accesses the plaintext results of a SMC function.

Oblivious RAM In addition to covering the compute traces that are effected by the input of another party, our engine obfuscates the memory addresses accessed by an oblivious program. To do this, we use oblivious RAM (ORAM) [9] to store arrays of tuples as the data moves through the secure query executor. ORAM automatically garbles the data’s bits as we access them and shuffles the array at each read or write. This makes all memory accesses appear identi-
cal, preventing attackers from using access patterns to learn about the underlying data. ORAM enables us to reduce the depth of our garbled circuits by starting each circuit after an oblivious read rather than making each query a single, long circuit. This oblivious data structure makes our system practical, but it comes at a high cost. Each read or write with ORAM takes polylogarithmic time, $O((\log n)^3)$, where $n$ is the number of elements in the array.

**SMC Language** To create garbled circuits in our framework, we use a SMC-specific programming language, ObliVM [18]. This language has a C-style syntax. The ObliVM compiler generates garbled circuits and ORAM accesses when a user builds a program. Among other features, ObliVM offers callable functions, loops, and if-then statements. ObliVM automatically integrates ORAM into its SMC programs. A programmer declares and accesses ORAM in ObliVM using a generalization of the C language’s bracket notation. This language is back-end-agnostic, so if one were to research new SMC algorithms, it is possible to seamlessly plug them into code generated by SMCMQL.

Our framework dynamically creates ObliVM code at runtime to implement the distributed database operators in a query plan. We then compile and run our SMC programs. The details about how our code generator translates database operators into garbled circuits are in Section 4.3

### 2.1 Running Example

Throughout this text, we will use a running example of two hospitals that wish to mine the collective data of their electronic health record systems (EHRS) for research while keeping their individual data private. We use a set of three representative queries based on clinical data research protocols from our colleagues in medicine [12, 22].

**C. Diff** Clostridium difficile, or c. diff, is an infection that is commonly associated with individuals on a prolonged regimen of antibiotics, and it is known for having a high rate of recurrence. Many recurring infections go undetected when patients are treated at multiple hospitals. This is exactly the type of problem that SMCMQL is designed to solve. Our first query identifies a cohort of recurrent c. diff patients whose second infection occurs between 15 and 56 days after an initial diagnosis.

\[
\begin{align*}
\rho_{\text{cd}}(\gamma_{\text{pid}}.\text{time}.\text{row no} (\sigma_{\text{diag} = \text{cdiff}}(\text{diagnoses}))) \\
\delta_{\text{pid}}(\rho_{\text{cd}}(\text{cd})) \bowtie \rho_{\text{cd}}(\text{d.time} \in [15, 56] \text{days}) \quad \rho_{\text{cd}}(\text{cd})
\end{align*}
\]

This query first selects for c. diff diagnoses, followed by a window aggregate over (patient ID, timestamp) that produces a row number. We create this list sorted on patient ID and timestamp to identify sequences of c. diff diagnoses. We then compare the i-th diagnoses to the (i + 1)-th one using a self join to find recurring infections. Finally, duplicate patient IDs are removed, giving all distinct patient IDs that have recurring c. diff. We use this query to examine how fine-grained data partitioning impacts SMCQL query performance.

**Comorbidity** Our next query continues our examination of the c. diff cohort. Here, researchers want to find the most common types of ailments that affect c. diff patients. We find the top ten most common diagnoses for individuals with c. diff.

\[
\begin{align*}
\rho_{\text{cohort}}(\sigma_{\text{pid}}(\sigma_{\text{diag} = \text{cdiff}}(\text{diagnoses}))) \\
\text{limit to}(\tau_{\text{count}}(\gamma_{\text{diag_count}}(s) (\sigma_{\text{pid} \in \text{cohort}} \times \text{diag} \times \text{cdiff} (\text{diagnoses}))))
\end{align*}
\]

In this query, we select the patient IDs in the c. diff cohort over both hospitals. The query then selects the diagnoses of the c. diff cohort, also eliminating the initial c. diff diagnoses. We then count the diagnoses for each comorbid condition and sort on this count to find the most common co-occurring conditions. Using this query, we explore practical techniques for partitioning a query plan to minimize its use of SMC.

**Aspirin Rate** Our final query shifts gears a bit. Here we identify the fraction of heart attack sufferers who were prescribed aspirin after an attack. In this scenario, researchers are investigating the effectiveness of Aspirin in preventing repeated heart attacks [12]. The same issue that we saw in c. diff and comorbidity appears again; patients may receive a heart attack diagnosis and aspirin prescriptions from different care providers, which blocks researchers’ attempts to link the heart attack with its treatment. We calculate the aspirin rate of heart attack patients as:

\[
\begin{align*}
\rho_{\text{d.count}}(\gamma_{\text{count}}(\delta_{\text{pid}}(\sigma_{\text{diag} = \text{M.I.}}(\text{diagnoses})))) \\
\rho_{\text{x.count}}(\gamma_{\text{count}}(\delta_{\text{pid}}(\sigma_{\text{diag} = \text{M.I.}}(\text{diagnoses})))) \\
\delta_{\text{pid} \in \text{m.time} \in [d.time] \times \text{med} \in \text{aspirin}(\text{medications})) \\
\pi_{\text{d}/\text{d}}(\text{rx.count} \times \text{d.count})
\end{align*}
\]

The query selects distinct patient IDs that are diagnosed with ‘M/I’, myocardial infarction, and then counts the number of heart attack sufferers. It renames the result $\text{d.count}$. Next, we count the distinct patients who experienced a heart attack and were prescribed aspirin during or after the time of the attack, $\text{rx.count}$. Finally, we join the two results and project the ratio of patients who received aspirin to heart attack sufferers. We designed this query to test the SMCMQL rewrite framework’s ability to create high-performance query plans over complex sequences of operators with mixed plaintext and secure computation.

**HealthLNK** In our results, we evaluate this workload on de-identified medical records from the HealthLNK data repository [23]. This repository contains data from seven different Chicago-area healthcare institutions, each with their own member hospitals, from 2006 to 2012, totaling about 6 million records. The data set is selected from a diverse set of hospitals, including academic medical centers, large county hospitals, and local community health centers. The details of our experimental configuration are in Section 5.1.
2.2 SMC Costs

To get a feel for the performance costs and optimization opportunities associated with secure query execution, we hand-coded the three queries above as oblivious programs. We ran these implementations on a randomly selected subset of tuples that matched the query’s initial selection criteria. For example, the comorbidity test only operated on co-occurring diagnoses from the c. diff cohort. To make our tests complete in a reasonable time, our input samples contained 50 tuples per table.

Our results are shown in Figure 1. We see that the pure SMC performance is 4–5 orders of magnitude slower than plaintext, even with carefully coded optimizations. This is unacceptably slow. Instead of naively applying SMC to an entire query plan, a more nuanced approach is needed. For each query, we identify coordination points where SMC selectively generates and run garbled circuits. In other words, we keep computation within the local database whenever possible.

In this study we also found several opportunities for improving the performance of secure query execution. We had the most speedup by minimizing the number of times we loop over the data and by running SMC over the smallest possible subset of the data. These collections of smaller computations reduce the time required to do lookups in ORAM and the number of comparisons we execute in garbled circuits. In addition, we found that partitioning the input data into smaller chunks allowed us to avoid SMC for certain subsets of the data. Fine-grained partitioning of the data and computation in a PDN query lies at the heart of smcql, and this makes it possible to offer secure execution with practical performance.

3. PRIVATE DATA NETWORKS

We now introduce the architecture within which smcql will run queries over mutually distrustful parties. Private Data Networks, or PDNs, are a generalization of federated databases that enable two or more databases to work together to compute a query without sharing their data with one another. To accomplish this, the framework needs a way to reason about the security requirements of a PDN and the relational algebra used in a query.

3.1 Architecture

The PDN architecture is shown in Figure 2. The honest broker is the entry and exit point of the PDN. It is a trusted, neutral federator that plans and coordinates query evaluation over its data providers. In the figure, Alice and Bob represent the two untrusting data providers in the PDN. After receiving a plaintext query from the user, the honest broker rewrites it into a secure query plan. This query plan contains a partitioned version of the original query, and is comprised of a series of plaintext and secure execution steps that, when taken together, compute a given SQL statement.

To set up a PDN, the data providers begin with a shared set of table definitions. This schema is annotated with the level of protection required for each attribute in the tables. In addition, the configuration denotes whether a table is replicated or partitioned among members. For matching identifiers, such as patient IDs in HealthLNK, the honest broker works with the data providers to carry out secure record linkage [4] [11].

Figure 2: Architecture of a Private Data Network with SMQL.

A PDN query execution begins when a user submits a SQL statement to the honest broker. The broker uses a regular database with the PDN’s table definitions to compile the SQL statement into a directed acyclic graph (DAG) of database operators. Using this query tree, the planner automatically generates a query plan containing a mix of plaintext and SMC execution. It uses the annotated schema to determine the level of security needed for each node in the operator tree based on the attributes it computes on and how it accesses them.

Once the query plan is generated, the honest broker sends a copy of it to both Alice and Bob. This plan consists of a set of steps for executing the mixed computation query. If a step is plaintext, each party runs it locally within their private database. If, however, a step requires coordination over sensitive data, then Alice and Bob compute it securely. For this, Alice uses her inputs to generate a garbled circuit, while Bob uses his data to evaluate that circuit. The output of this garbled circuit generation and evaluation is a set of shared secret bits (SSBs) for both Alice and Bob. After this, Alice and Bob move on to compute the next step in the plan. Once all the plan’s steps are computed, Alice and Bob pass their respective SSBs back to the honest broker, who combines them to reveal the plaintext query results. The honest broker then returns the plaintext query output to the user.

3.2 Privacy Model

PDNs use a simple, but powerful security model to protect the data in a PDN. Recall that a PDN begins with a common data model or schema. The federation offers attribute-level security, allowing PDNs to define the level of protection needed for their data one column at a time. A PDN framework has three levels of protection for attributes: public, protected, and private. By working with stakeholders, a DBA or other administrator works creates an annotated version of the federated database schema to define the conditions under which the data is queryable.

Public attributes are readable by all parties, including the honest broker, all of the data providers, and end users. Attributes of this kind have minimal risk of revealing private information, and may have a low probability of being independently replicable. In HealthLNK, patient IDs, lab results and blood pressure readings are public attributes.

The contents of protected attributes are not visible to other PDN data providers. The honest broker and end
user may see them conditionally. The security policy associated with these attributes is negotiated by the PDN members at configuration time. Any computation involving these attributes must be done securely. For our medical use case, SMCQL uses k-anonymity to control access to these attributes. A data selection is k-anonymous if each of its tuples is indistinguishable in its identifying information from at least k records. This policy is just one of many potential security policies for protected data. In our running example, our protected attributes include diagnosis codes and demographic data such as age and gender. It is possible to configure a PDN with multiple classes of protected attributes, each with a different access control policy.

Private attributes are the most sensitive values in a PDN, they are not disclosed to anyone outside of the initial data provider. This is the highest level of security provided by SMCQL. Any computation done over these attributes must be carried out by the originating database or within SMC. Private attributes may not be present in any results returned to an user. Timestamps and zip codes are examples of private attributes in HealthLNK.

When multiple attributes are accessed in a query together, SMCQL operates at the security level of the most sensitive attribute. Before the framework starts executing a query, it validates the operation against the security policy provided by the annotated schema. The framework uses this policy to plan PDN queries that will run efficiently by minimizing the computation it performs within SMC.

In addition to implementing attribute-level security, a PDN must be configured with an admission control policy. This policy may disallow certain types of queries, such as repeated, but slightly modified ones designed to unmask individuals in a database. It also may build in protection for data providers with requirements such as “at least k databases must contribute data to a secure computation”. This would ensure that each party can hide in the crowd. A PDN may automatically reject queries that do not meet its policy, as specified by the stakeholders of the network, using a system such as DataLawyer 26. We leave the design of such policies to future work.

3.3 Example Application

A clinical data research network or CDRN is a consortium of healthcare sites that agree to share their data for research. CDRN data providers can be mutually distrustful parties. In the absence of a system like ours, data providers in a network will only volunteer their public and protected attributes for querying. Also, all of the PDN’s query processing takes place inside the honest broker rather than distributing the work over members of the network. Clearly, this hub-and-spoke approach will not scale to hundreds of data providers—an issue we are already seeing in the field.

HealthLNK is a prototype CDRN. More specifically, it is the forerunner for the Chicago Area Patient-Centered Outcomes Research Network (CAPriCORN), which is itself part of a national network in the US, the Patient-Centered Outcome Research Network (PCORnet). CAPriCORN and HealthLNK share the majority of their stakeholders and this group of hospitals and clinics designed this CDRN to meet the needs of clinical researchers, especially ones exploring personalized medicine.

CAPriCORN is a consortium of Chicago area hospitals, containing 481 data sources, each of which include protected health information (PHI) such as gender, timestamps, and patient IDs 15. Each data provider within CAPriCORN may not wish to permit other consortium members to view their protected health information, but they are willing to contribute de-identified data that is combined with other records. This dual need to secure some attributes in a schema, yet allowing users to aggregate over many parties, makes CDRNs an interesting use case for SMcQL. We are investigating deploying a prototype of SMcQL on CAPriCORN.

Looking at our architecture in Figure 2, we see how CAPriCORN can be configured as a PDN. Alice and Bob are two healthcare sites, each containing a database with PHI and data that does not require protection. The honest broker is a trusted coordinator that takes a plaintext query from the user, rewrites it into a SMcQL query plan, and passes that query plan to or more hospitals. From here, the two parties generate and evaluate garbled circuits to jointly compute their own SSBs, which are then returned to the honest broker. The honest broker combines the SSBs to reveal the plaintext results that it sends to the user. This architecture distributes the computation among Alice and Bob, decreasing the maximum load on any one hospital in the network. For CDRNs, this distributed computation is extremely important. If a PDN performs most or all of its computation within the honest broker, this system will not scale to the large number of users who wish to issue queries to consortium members.

4. QUERYING OVER SECURE MULTIPARTY COMPUTATION

We now describe SMcQL’s query planner. The planner translates a SQL statement into a series of fine-grained query execution segments—each of which may execute securely or using a mix of plaintext and secure processing. In this section, we first look at how we analyze a query against a PDN security policy to identify when it must compute using SMC. Next, we examine how SMcQL translates database operators into SMC code. After that we describe how SMcQL partitions a query’s input data to further improve our SMC performance.

4.1 Preliminaries

As we saw in Section 21, SMC incurs a prohibitively high overhead when used for query execution. To address this, our framework generates query execution plans that minimize their usage of SMC. SMcQL accomplishes this by identifying coordination points in the query plan and generating SMC code for them alone. The planner begins with a directed acyclic graph (DAG) of database operators sourced from the database’s explain facility. To get this initial operator tree, the honest broker runs explain over the user’s SQL statement on an empty database that has the PDN’s schema. The planner parses the DAG, matching all of the attributes referenced therein to their security level in the PDN schema. The planner uses the DAG to locate secure leaves or positions in the query tree where the executor must perform distributed query evaluation using SMC. A secure leaf may arise between nodes in the tree. The SMC entry point may also occur within a database operator if its execution is splittable into local and distributed phases. Also,
the planner identifies opportunities to slice the data if it is partitionable using public attributes.

4.1.1 Database Operator Properties for Secure Computing

We have identified a set of database operator characteristics with which the query planner infers the coordination points in a federated database query plan. By reasoning formally about these properties, we can both generate more efficient SMC code and push as much work as possible into the underlying databases in the PDN. These SMC-related properties are shown in Table 1. This approach is a generalization of federated database query planning.

We say that an operator requires coordination if it must perform distributed computing in order to produce output tuples. All of the operators, except scans, combine tuples from both parties to produce their results. A join requires coordination conditionally—it needs to sync with the other party only when both of its input tables are not replicated.

The planner also considers whether an operator is splittable or able to have its execution partitioned into discrete phases, such as local plaintext and distributed secure computation. If a secure leaf is splittable, we partially compute it within the database. For example, if a secure leaf is a COUNT(*), smcql computes a partial count locally on each party and then sums the counts within SMC. This substantially reduces the number of tuples we read in for the secure query segments. Most of the operators that require coordination are splittable. Sort is split into a local ordering on the sort key followed by a secure merge phase. Window aggregates and DISTINCT work similarly to standard aggregation.

Next, we consider whether we can slice the secure query execution—partitioning the data to create smaller units of secure computation. Each sliced operator bins the input data by a slice key. The slice key contains a list of public attributes that are referenced by the operator being sliced. Inputs to an aggregate are sliced by their GROUP BY clause. Similarly, window aggregates are computable once per PARTITION BY group. Likewise, a sort operator is partitionable by some or all of its sort key. Sort is the only operator that can perform sliced computation on partial slice keys; all other operators need the entire key to be public to take advantage of sliced query execution.

It is possible to compute a SMC DISTINCT operator in slice mode if its entire (public) schema is a slice key. At first this might sound paradoxical—why use SMC if the inputs are all public attributes? smcql does its planning on the minimal set of attributes needed to compute a given operator and all operators that rely on its output. Thus attributes may drop out of the query’s schema as tuples move through the operators. An example of this arises in c. diff, where the join has an input schema of (patient id, timestamp) for each input and its outputs tuples have a single patient ID. A secure DISTINCT follows the join and it operates on this formerly public attribute. The attribute must now be private to obfuscate the output of its children that were computed securely.

For code generation, smcql assigns a secure compute order to most operators. This input order for the secure operator minimizes the number of passes the generated SMC code makes over the data. If an operator is a secure leaf, its plaintext inputs are sorted before we ingest the data into

| Operator       | Req. Coord? | Splittable | Slice Key(s) | SMC Order |
|----------------|-------------|------------|--------------|-----------|
| Table Scan     | No          | N/A        | N/A          | N/A       |
| CTE Scan       | No          | N/A        | N/A          | N/A       |
| Join           | Yes         | No         | Equality Predicates | N/A       |
| Aggregate      | Yes         | Yes        | GROUP BY     | GROUP BY  |
| Win. Agg       | Yes         | Yes        | PARTITION BY | PARTITION BY |
| Limit          | Yes         | No         | N/A          | N/A       |
| Distinct       | Yes         | Yes        | schema       | schema    |
| Sort           | Yes         | Yes        | sort key     | N/A       |

Table 1: DB operators and their SMC properties. SMC. Otherwise, if an operator does not have the same compute order as its children the query planner automatically inserts a sort before the operator into the execution DAG.

In many cases, the secure compute order is equal to the slice key. Aggregates are ordered by their GROUP BY clause. The secure operator performs a single pass over the data that combines adjacent group-by values. If we were to implement this as a hash aggregate or some other method that uses complex data structures this would substantially complicate the generated garbled circuits. Likewise, window aggregate operations on values that are already sorted by their PARTITION BY and ORDER BY clauses. If we run a DISTINCT operator without slicing its execution, then its generated code resembles an aggregate; it compares adjacent values and emits the unique ones.

4.2 Minimizing SMC Usage

When planning a PDN query, smcql first assigns an execution mode to each database operator in its DAG. An operator may run in one of three execution modes. We will run in plaintext mode if an operator computes exclusively on public attributes or it requires no coordination. If an operator’s computation is partitionable on public attributes, it is a candidate for sliced mode. Sliced operators in the DAG must have children that are either plaintext or sliced using the same key. All other operators are run in private mode.

smcql generates a logical query plan using Algorithm 1. The planner starts from the bottom of the query’s tree. The DAG’s leaves are all scans and since they require no coordination, they are marked for public execution. Non-leaf nodes in the DAG only switch to secure or sliced mode if they compute on data that is not public. If a plaintext operator requires coordination, we rewrite the query plan to compute over the union of Alice and Bob’s inputs using standard federated database techniques.

As the planner works up the query tree, it identifies the secure leaf for each branch of the query plan. If a secure leaf has a public slice key, it is marked for slice mode. Otherwise, the plan switches to secure query evaluation.

As the planner traverses the query tree, the parents of secure leaves infer their execution mode using the modes of their children. Since a query can not go from a more secure mode to a less restrictive one, if an operator has a secure child it too will execute in secure mode. If an operator is sliceable on public attributes and its slice key intersects with all of its children, then the query plan remains in slice mode. Otherwise the plan switches to secure mode so that it will not reveal any information about protected or private attributes to other parties.

1Except when ≥ 1 inputs are replicated
Function inferExecutionMode
Input: DatabaseOperator o
Output: ExecutionMode e // mode in which o will run
if o.children().isEmpty() then
  return e;
else
  for c in o.children() do
    childMode = inferExecutionMode(c);
    if childMode == Secure then
      e = Secure;
    else if childMode == Slice then
      if o.sharesSliceKey(c) and e != Secure then
        e = Slice;
      else
        e = Secure;
    end if
  end for
end if

Algorithm 1: Method for deducing execution mode of operators in a PDN plan.

Physical Query Planning
Once the planner has assigned an execution mode to each operator in the DAG, it groups the operators into segments. Two connected operators belong to the same segment if they share an execution mode and, if sliced, have an overlapping slice key. Segments are the unit of execution in smcql and they crucial to making our secure coordination points fast and scalable. Each segment takes up to two ORAM instances as input and produces a single secure array as output.

By grouping together operators, we only need to initialize our input ORAM once per segment rather than once per operator. Since reads and writes to ORAM are very costly, and initializing ORAM means accessing the data structure $n$ times at a polylogarithmic cost each time, grouping together as many operators as possible is very important for our performance. To give an intuition about how expensive this ORAM setup is, ingesting 400 tuples for comorbidity takes about 200 seconds. If this entire query were run without SMC, it would take 10 seconds. Clearly building ORAM instances once per operator is not a scalable design.

In addition to inferring the execution mode of each operator, the planner also determines how to combine the inputs of Alice and Bob. When a secure leaf ingests data that is not replicated across the hosts, smcql inserts a merge operator to combine the inputs of both parties. This merge creates a single ORAM instance containing the union of the tuples provided by each party ordered by the secure compute order of the secure leaf. In many split operators, the merge will execute logic for combining partially computed operators, such as adding up counts from partial sums.

After this step, the framework has a series of execution segments. Figure 3 shows the execution plan for aspirin rate query. This plan has three secure leaves. The first counts the distinct of heart attack sufferers. Since this branch of the query tree only coordinates on patient ID, a public attribute, we compute it in plaintext.

The latter secure leaves prepare inputs for a join that identifies individuals who received a prescription for Aspirin after a heart attack. The planner sees that the join is sliced on patient ID and that the subsequent DISTINCT operator is sliced on the same. The prescription count calls for a split execution of the COUNT, and the framework performs a secure merge to combine the outputs of the sliced segment. We now switch to secure mode since this aggregate spans multiple patient IDs. After the secure count, the system will ingest the diagnosis counts from the first secure leaf and calculate the aspirin prescription rate. Once we have the execution segments, smcql is ready to generate the garbled circuits needed to run a PDN query.

4.3 Secure Operator Generation

We now explore the process with which we translate secure segments in a PDN plan into SMC. We generate code for a segment one operator at a time. The code generator starts with a template. Recall that we generate the garbled circuits by creating a program in ObliVM, a SMC-specific language. An operator’s template is a parameterized version of ObliVM’s C-style programs for its execution logic. The latter secure leafs prepare inputs for a join that identifies individuals who received a prescription for Aspirin after a heart attack. The planner sees that the join is sliced on patient ID and that the subsequent DISTINCT operator is sliced on the same. The prescription count calls for a split execution of the COUNT, and the framework performs a secure merge to combine the outputs of the sliced segment. We now switch to secure mode since this aggregate spans multiple patient IDs. After the secure count, the system will ingest the diagnosis counts from the first secure leaf and calculate the aspirin prescription rate. Once we have the execution segments, smcql is ready to generate the garbled circuits needed to run a PDN query.

4.3 Secure Operator Generation

We now explore the process with which we translate secure segments in a PDN plan into SMC. We generate code for a segment one operator at a time. The code generator starts with a template. Recall that we generate the garbled circuits by creating a program in ObliVM, a SMC-specific language. An operator’s template is a parameterized version of ObliVM’s C-style programs for its execution logic. The latter secure leafs prepare inputs for a join that identifies individuals who received a prescription for Aspirin after a heart attack. The planner sees that the join is sliced on patient ID and that the subsequent DISTINCT operator is sliced on the same. The prescription count calls for a split execution of the COUNT, and the framework performs a secure merge to combine the outputs of the sliced segment. We now switch to secure mode since this aggregate spans multiple patient IDs. After the secure count, the system will ingest the diagnosis counts from the first secure leaf and calculate the aspirin prescription rate. Once we have the execution segments, smcql is ready to generate the garbled circuits needed to run a PDN query.
$tupleFilter

int$distSize[mw] join(int$distSize[m] lhs, int$distSize[n] rhs) {
    int$distSize[mw] dst;
    int dstIdx = 0;
    bfor(m) (int i = 0; i < m; i=i+1) {
        int$distSize l = lhs[i];
        bfor(nw) (int j = 0; j < n; j=j+1) {
            int$distSize r = rhs[j];
            if($invokeFilter(l, r) == 1) {
                dst[dstIdx] = $writeDst;
                dstIdx = dstIdx + 1;
            }
        }
    }
    return dst;
}

Figure 4: Example template for secure join

The function takes in two tuple-sized integers and uses bitmasks to access the attributes individually. We use the same expression translator to generate the $writeDst variable that resolves any arithmetic expressions and orders the output attributes correctly.

The code uses an oblivious bounded loop, bfor. The bfor is defined with a public loop guard that denotes the maximum number of times it may execute. By running the garbled circuits m times for the outer loop, this makes it impossible for the party executing the code to learn anything about the other party’s inputs or what happened in the join’s child operators. The nested loop’s inner iteration will run m * n times. The inner loop invokes the filter on each potential tuple pairing and appends the matches to the output ORAM, dst.

The code generator has a library of secure code templates. The template from which it will generate when it translates an operator into SMC is selected based on the operator’s properties in Table 1. In doing this, we exploit opportunities to radically simplify the generated code. For example, if a DISTINCT operator runs in sliced mode, then it only needs to check whether the output of its secure child has one or more elements. Otherwise, a secure DISTINCT must iterate over the entire ORAM instance to identify duplicate tuples.

4.4 Slicing Query Data

Now that we have our query plan and have translated the secure operators into garbled circuits, we will examine how we partition the input data of sliced operators for secure processing. There are several reasons why this slicing is important for high-performance SMC execution. First, as described above, sliced operators often have much simpler program logic and thus less depth in their garbled circuits.

Second, slicing reduces the cumulative effect of cascades of operators. As query trees grow larger, each secure operator’s input sizes are a function of that of their children. For example, say we have two cascading joins and each join input consists of n tuples. The first oblivious join will produce n² tuples and the subsequent one will produce an ORAM instance with n³ elements. When we slice the inputs, we may tame this complexity.

Lastly, since segments identify groups of sliced-alike operators, this further reduces our ORAM-initialization costs by creating a set of smaller ORAMS once per slice value rather than a single large data structure. Since the ORAM access cost is proportional to the number of tuples in its array, this is a substantial savings. We will now examine how SMQL identifies segments of sliced operators.

In sliced query execution, we first identify the slice values that require coordination between parties in a PDN, such as all of the c. diff patient IDs that appear in both data providers. We use slice values to bin together tuples that need to be evaluated within the same set of garbled circuits. We invoke the segment’s generated code once per slice value to compute its part of the secure query evaluation.

We first create a composite slice key, the union of the attributes that are slice keys on one or more operators in a segment. If a slice key contains join predicates, all of the attributes it references are included in the composite key. This segment key dictates what public values need to be aligned in all of the source data for a slice value to require coordination. The planner then generates a query listing the distinct slice values associated with each attribute in the key.

If a segment’s slice key contains just one attribute, sᵢ, then each party, p probes their database for attribute sᵢ to produce a list of slice values tᵢ,p where p may be equal to A or B for Alice or Bob respectively. The parties each report these values via an encrypted connection to the honest broker to prevent leaking information about a party’s data distribution. If the segment has a single slice key, i, the honest broker finds the slice values for execution as:

\[ I = t_{i,A} \cap t_{i,B} \]

We combine these statistics to tell us which slice values need to be computed within SMC.

If a segment has k slice keys \[ S = \{s_1, ..., s_k\} \], we calculate the key values needed for the entire segment as:

\[ I = \bigcap_{i=1}^{k-1} t_{i,A} \cap t_{i+1,B} \]

We identify the possible join matches for attribute i and attribute i + 1 between Alice and Bob by iterating through the list of slice values associated with each key. Each secure leaf only computes on tuples with slice values in I. In addition to identifying fine-grained slice values that will be evaluated together, we also find out which slice values will not require coordination—the slice complement. Since we know that any tuples with slice values not in I can be processed without coordination, each party p creates a separate plaintext execution track for the slice complement of table i, tᵢ. The planner determines the slice complement once per secure leaf:

\[ P_i = t_{i,p} \setminus I \]

It is the set complement of I. We put this selection criteria into the filter for the table scans and joins that make up the slice complement.

To continue our aspirin rate example, the sliced segment’s key is (d.patient_id, m.patient_id) and its parent is sliced on the latter attribute. Thus the planner collects a list of patient ids for each attribute within each party. It selects patient ids for distributed execution if a diagnoses.patient_id appears in Alice and the same patient also appears in Bob’s aspirin prescriptions or vice versa. The sliced join then tests to see if the MI diagnoses happened before or during an encounter when the patient was prescribed aspirin by comparing the private timestamps associated with each event.
In other words, there has to be a potential distributed join match for a slice value to be run with secure execution. Patient IDs that do not match this criteria for distributed slice execution are evaluated locally on each host in the slice complement. The complement and the distributed slices are merged into a single ORAM for evaluation in the COUNT.

To summarize, each party prepares one ORAM instance per slice value in I and both parties pass their inputs into a merge to prepare the data for processing in SMCQL’s generated operators. In addition, the honest broker generates a plaintext version of each sliced segment for computing over the slice complement, or remaining tuples that do not appear in the intersection set of the slice key over both parties.

5. RESULTS

We now verify that SMCQL produces query plans that run with a manageable level of overhead. We first examine the scalability of SMCQL as it executes over input data of increasing size. After that, we quantify the impact of sliced execution in speeding up SMC computation. Lastly, we compare the execution times of queries in our example workload from Section 2.1 that are run with automatically generated SMC plans to ones that are run in a hypothetical federated database where the parties trust one another.

5.1 Experimental Configuration

We evaluate SMCQL on real-world data sourced from two Chicago-area hospitals in the HealthLNK data repository over one year of data. This dataset consists of 500,000 patient records and totals 15 GB of data. The queries in our workload are based on clinical research protocols.

Our SMC query executor is built on top of PostgreSQL 9.5 on Ubuntu Linux. Our plaintext distributed query evaluation is implemented using PostgreSQL’s foreign data wrapper extension. Thus all computation that does not occur in SMC happens directly in the database, minimizing overhead from data movement and our performance benefits from standard query optimization (e.g., selecting the best join algorithm). Unless otherwise specified, our results show the end-to-end execution time of an experiment, including both the plaintext and secure query evaluation. All of our figures report runtimes using a logarithmic scale.

We conducted our experiments using 8 servers running in pairs. The servers have 64 GB of memory, 7200 RPM NL-SAS hard drives, and are on a dedicated 10 Gb/s network. All of the nodes have Intel Xeon processors running at 2.2 GHz on 6 cores. Our results are reported as the average of three runs per experiment.

5.2 Scalability

We first test how well SMCQL scales as our dataset increases in size. For this experiment, we ran the comorbidity query. We artificially limited the size of the SMIC input—the partial counts of comorbid conditions in the c. diff cohort—to 100, 200, and 400 tuples. Each party ingests an equal number of tuples. In addition, we ran the same experiment with the full dataset; it had nearly 800 diagnoses codes with each party supplying about 650 partial counts. We report the runtimes of these experiments in seconds.

Our results are shown in Figure 5. The plaintext execution time of this query, without the use of SMC, is consistent as we scale out. This is because the database spends the

Figure 5: Query execution times for comorbidity query with varying SMC input sizes.

majority of its execution time reading each table, identifying the cohort, and computing the local count in this query’s plaintext execution. Since this is a sunk cost, and we filter the tuples after this, there is little variance in this baseline.

In contrast, this query’s secure execution shows a substantial increase in duration as the framework computes over more data. This is primarily caused by increasing costs associated with larger ORAM instances for storing and accessing the data. Because the secure array shuffles the data every time we access an element, its cost increases at a rate proportional to the input size. In addition, the secure operators must perform more comparisons among the inputs to sum up the partial counts, further slowing down SMCQL’s execution.

The smallest scale run in this analysis exhibits a modest slowdown; it runs 13X more slowly than its equivalent plaintext execution. With an input size of 400 tuples, the query’s duration jumps to 360X its conventionally executed counterpart. When we run over the entire SMC input, our runtimes go to 11 hours or 2,700X the baseline. Since this query’s slice key is the diagnoses code, a protected attribute, the execution cannot be trivially partitioned into multiple ORAM instances.

5.3 Sliced Query Execution

We now examine the role of sliced query execution in making SMCQL queries complete more efficiently and with greater scalability. We first examine the performance of aspirin rate in both sliced and unsliced PDN query plans. The performance of both executions for each sliced operator in aspirin rate are shown in Figure 5. All runtimes are reported in milliseconds. In this experiment we observed that this query’s second, secure segment had comparable runtimes between the sliced and unsliced executions. Their runtimes differed by less than a second. This is unsurprising since the generated code for their segments is identical. The plaintext version of aspirin rate ran in 15 seconds. The data contained 15 patients, enabling the query executor to partition the SMC inputs into numerous smaller, more manageable ORAM instances.

The sliced execution of aspirin rate begins by merging the
secure inputs of the two parties. It then performs a join to identify individuals who received an aspirin prescription after a heart attack diagnosis. Lastly, it eliminated duplicates to identify the distinct patients who meet this criteria. This query is sliced by patient ID and by computing this workflow one patient at a time the join performs much fewer comparisons. The DISTINCT only checks whether the number of elements in the join output is greater than zero. It does not iterate over each element comparing it to its neighbors.

The overhead incurred by SMC were remarkably reduced when executing in sliced mode. The sliced execution of this query ran 7X slower than its insecure equivalent. In contrast, the unsliced run of this query took 1,200X as long as the baseline. Clearly, SMCQL benefits substantially from this fine-grained partitioning of private data.

Another important trend in this data is the cumulative effect of oblivious computing over cascades of operators. In order for later operators, such as distinct, to not leak information about the output of its secure children, the engine allocates ORAM instances that are the size of the worst-case scenario for its children. The details of this allocation are described in Section 4.4. It is easy to imagine how the costs of large operator trees explode quickly from this compounding effect of the output size of secure operators.

We also investigated the impact of slicing on the c. diff query, and our results are shown in Figure 7. This query has a single execution segment and it is sliced on patient ID. The performance benefit of slicing is less pronounced for this query because it has only 3 slice values, and it has fewer tuples per slice—for c. diff the median tuples per slice for was 2, whereas it was 4 for aspirin rate. Thus the c. diff joins had much fewer comparisons in both its sliced and unsliced executions.

The merge and window aggregate steps make a simple, tuple-at-a-time pass over the data and they run in about a third of the time that unsliced execution takes. This speedup is a result of working with smaller ORAM instances, thus lessening the time needed to access the data. These two operators are also fast because we do not have the explosion in ORAM size associated with joins. After that, c. diff exe-

cutse a self-join on the numbered rows of its patient cohort. The join executes in 25% of the time of its unsliced version. This speedup is modest compared to the one we saw for aspirin rate, which had an unsliced execution time that was 700X the duration of the sliced run.

Both c. diff and aspirin rate benefited from slice-aware code generation for DISTINCT. Recall that this is because the sliced operator tests just one element per slice to see if the slice will contribute to the final query output saving the executor from iterating over all elements in the ORAM instance. This optimization is the source of the majority of c. diff’s sliced execution speedup.

**Parallelized Slice Execution** Slicing is motivated by bringing down the cost accessing individual tuples in a database operator since this cost in ORAM is proportional to the input size. Slicing also creates sets of tuples that can be evaluated independently. We may further speed up our query executions by parallelizing the evaluation of individual slice values, such as computing SMC over many patient IDs in aspirin rate at the same time.

To estimate this potential performance gain, we simulated the performance of parallelized slice execution by analyzing the per-slice runtime of each segment. We modeled having 4 SMC workers, assuming each has its own CPU core, and assigning the slices to workers in round robin order. We then summed up the runtime of each worker and take that as the duration of a segment. The results of this analysis are in Table 2.

| Test         | Sliced | Sliced Parallel |
|--------------|--------|-----------------|
| Asprin Rate  | 103.72 | 92.77           |
| C. Diff      | 35.88  | 35.04           |

Table 2: Simulation of parallelized slice execution
ing. We leave the question of SMC scheduling algorithms to future work.

Discussion This analysis of parallelism highlights that the main benefit of slicing is reduced SMC time. This reduced SMC time comes about because of shallower garbled circuits, such as a nested loop join comparing fewer tuples. This performance improvement also arises owing to the aforementioned reduced ORAM size.

Depending on the setting, each of these costs (code complexity, ORAM access) may become the bottleneck for performance. For example, if a slice has very few tuples, the cost of setting up ORAM and initially shuffling the data is never amortized in the form of having smaller execution paths of garbled circuits. On the other hand, if the operator has complicated logic—such a window aggregate—over medium-sized data, ORAM is crucial for making the operator scale. An interesting potential future direction for this work is identifying when and how to use ORAM versus deeper garbled circuits circuits.

5.4 Overall Performance

We now evaluate the end-to-end duration of HealthLNK queries run in smcql by comparing them to an equivalent insecure execution in plaintext. Our results are shown in Figure 5. Here, aspirin rate and c. diff run in sliced mode whenever possible, and these experiments do not parallelize slice evaluation. Since comorbidity’s slice key—diagnoses code—is a protected attribute, it is not eligible for sliced evaluation.

SMCQL executes both sliced queries in under 5 minutes each. These queries are accessing a collective 42 million diagnoses and 23 million medication records. By using fine-grained partitions of the underlying dataset, the SQL operators scale well to large volumes of data.

We see noticeably slower execution times for comorbidity. The secure query executes in 11 hours, a dramatic slowdown from its 15-second insecure runtime. The main reason this query runs so slowly is because both parties ingest very large ORAM instances containing hundreds of elements. More work is needed to manage this challenge of securing large in-memory data structures.

One possible solution for scaling SMCQL to large datasets is to partition the work into smaller pieces and use the honest broker to assemble the results. This approach would require an analytical cost model to identify how to finely to partition the work based on the rate at which the honest broker can merge and assemble the final results. It would also require the planner to work adaptively relative to the security policy for protected attributes. In our running example, this would mean making sure that intermediate results assembled by the honest broker adhered to k-anonymity.

In summary, our results demonstrate that SMCQL provides scalable secure query evaluation over SQL for mutually distrusted parties. Our queries incur a moderate overhead in comparison to their plaintext equivalents. In exploring these results, we highlighted numerous opportunities for additional query optimization for PDNs, especially in deciding when and how to use ORAM and opportunities for parallelism in SMCQL’s fine-grained query execution plans.

6. RELATED WORK

There are several efforts to combine relational databases with secure computation in the literature. One such direction centers on privacy-preserving data mining [2]. In this area, the data set is distorted, either by partitioning it into arbitrary groups for computation or injecting noise. This approach gives imprecise query results, which makes it difficult to use in a research environment where precision is important for accurate decision-making. Differential privacy also guarantees the security of underlying databases, but like privacy-preserving databases, its approach trades precision for privacy. Our approach protects the data by making the queries that run over it adhere to a security policy.

Another strategy for privacy is to use homomorphic encryption for outsourced query evaluation, as in CryptDB [24]. Homomorphic encryption enables queries over encrypted data, closing down many routes for attackers to compromise the system. Rather than outsourcing computation, we propose to keep the data in the hands of its source. Thus we can efficiently execute our parts of our queries in plaintext locally providing further performance gains.

Additional work exists to enable database filters and joins through secure multiparty computation [17, 16]. Research in this area uses third-party trusted mining nodes to carry out the computation. In contrast, our work keeps the computation in the database as much as possible and performs SMC at the host where the data originated. This is an important generalization because it enables us to delay SMC until coordination is needed for a query.

In [1], the authors proposed an architecture for using SMC for outsourced computation. This vision paper outlined two-party SMC might be used if we encoded the entire database as shared-secret bits. Our approach differs because we do not design for outsourced computation. Also, to the best of our knowledge, this is the first work to implement SQL over SMC. The prior work did not perform any experimental evaluation or method of generating SMC code based on database operators.

Programming languages for SMC, like ObliVM [18], VM-Crypt [19], TASTY [11], and FAIRPLAY [29] generate secure code for procedural programs. These approaches rely on the user explicitly specifying how to route and compare the data from each party. In contrast, SMCQL seamlessly injects SMC into an existing language, SQL. 

![Figure 8: Performance of SMCQL on HealthLNK benchmark queries.](image-url)
7. CONCLUSIONS

In this work, we introduce private data networks (PDNs), a novel generalization of federated database systems for mutually distrustful parties. We then described SMQCL, a framework for translating SQL queries into garbled circuits for secure distributed query evaluation. We stress that this is the first approach that seamlessly integrates SMC into an existing language. SMQCL creates fine-grained partitions of a PDN’s query plan to both minimize the computation that takes place within costly garbled circuits and reduces the size of SMC inputs for scalable secure computation.

Our results demonstrate that by partitioning SQL processing at a fine granularity, we can achieve high-performance SMC queries. Our PDN query workload, which is based on a real medical use case and evaluated on de-identified medical records, runs within minutes for queries with sliced computation and has more modest slowdowns on queries that call for higher levels of protection. We designed this system to be practical. To this end, we are preparing SMQCL for an open source release. In addition, we are collaborating with stakeholders in a clinical data research network to start testing SMQCL in the field.

There are numerous opportunities for future research within a PDN. We are investigating how to generalize SMQCL to three or more parties. Queries within a PDN would benefit from automated techniques for coalescing loops rather than doing one pass over the data per database operator. There is also work to be done in parallelizing SMQCL plans especially in the context of three or more parties. Another interesting future direction to identify automatic SQL rewrite rules that further delay our entry into SMC and get better reuse out of intermediate results within secure a secure query segment. An additional research direction is coming up with SQL-specific SMC algorithms. Extensive research exists on SMC implementations for specific problems, like linear regression and matrix multiplication, but almost nothing has been done on improving SQL operators like the ones we generate in SMQCL.

8. ACKNOWLEDGMENTS

The authors thank Katie Jackson and Jess Joseph Behrens for their guidance and assistance with CAPriCORN and HealthLNK data. We also are grateful to the HealthLNK team for sharing de-identified electronic health record data for this study.

9. REFERENCES

[1] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. CIDR 2005, 2005.
[2] G. Aggarwal, N. Mishra, and B. Pinkas. Secure Computation of the kth-Ranked Element. Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, Mag 2-6, 2004, Proceedings, 3027:40–55, 2004.
[3] R. Agrawal and R. Srikant. Privacy-preserving data mining. Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00, 29(2):439–450, 2000.
[4] A. Al-Lawati, D. Lee, and P. McDaniel. Blocking-aware private record linkage. In Proceedings of the 2nd international workshop on Information quality in information systems, pages 59–68. ACM, 2005.
[5] J. R. Bambauer, K. Muralidhar, and R. Sarathy. Fool’s Gold: an Illustrated Critique of Differential Privacy. Vanderbilt Journal of Entertainment & Technology Law, 16(4):Paper No. 13–47, 2013.
[6] D. Chaum, C. Crépeau, and I. Damgård. Multiparty Unconditionally Secure Protocols. Proceedings of the twentieth annual ACM Symposium on Theory of Computing (STOC), pages 11–19, 1988.
[7] T. Dokos. A Fisher-Price explanation of Yao’s garbled circuits in secure computation. pages 1–5, 2014.
[8] C. Dwork. Differential privacy. Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, pages 1–12, 2006.
[9] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In Proceedings of STOC, pages 182–184, New York, NY, USA, 1987. ACM.
[10] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game. Stoc ’87, pages 218–229, 1987.
[11] W. Henecke, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, et al. Tasty: tool for automating secure two-party computations. In Proceedings of the 17th ACM conference on Computer and communications security, pages 451–462. ACM, 2010.
[12] A. F. Hernandez, R. L. Fleurence, and R. L. Rothman. The ADAPTABLE Trial and PCORNet: shining light on a new research paradigm. Annals of internal medicine, 163(8):635–636, 2015.
[13] A. F. Karr, X. Lin, A. P. Sanil, and J. P. Reiter. Secure Regression on Distributed Databases. Journal of Computational and Graphical Statistics, 14(2):263–279, 2005.
[14] A. Kho, J. Cashy, K. Jackson, A. Pah, S. Goel, J. Boehnke, J. Humphries, S. Kominers, B. Hota, S. Sims, B. Malin, D. French, T. Walunas, D. Melzer, E. Kaleba, R. Jones, and W. Galanter. Design and implementation of a privacy-preserving electronic health record linkage tool in chicago. In The American Medical Informatics Association, 22(5):1072–1080, 2015.
[15] A. N. Kho, D. M. D. Hynes, S. Goel, A. E. Solomonides, R. Price, B. Hota, S. A. Sims, N. Bahroos, F. Angulo, W. E. Trick, and Others. CAPriCORN: Chicago Area Patient-Centered Outcomes Research Network. Journal of the American Medical Informatics Association, 21(4):607–611, 2014.
[16] S. Laur, R. Talviste, and J. Willemsen. From oblivious AES to efficient and secure database join in the multiparty setting. Lecture Notes in Computer Science, 7354 LNCs:84–101, 2013.
[17] S. Laur, J. Willemsen, and B. Zhang. Round-efficient Oblivious Database Manipulation. ISC ’11, pages 262–277, 2011.
[18] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. OblIVM: A Programming Framework for Secure Computation. Oakland, pages 359–376, 2015.
[19] L. Malka. VMCCrypt: modular software architecture for scalable secure computation. In CCS, pages 715–724. ACM, 2011.
[20] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, et al. Fairplay-secure two-party computation system. In USENIX Security Symposium, volume 4. San Diego, CA, USA, 2004.
[21] M. Naveed, C. V. Wright, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 644–655. ACM, 2015.
[22] PCORI. Characterizing the Effects of Recurrent Clostridium Difficile Infection on Patients. IRB Protocol, ORA: 14122, 2015.
[23] PCORI. Exchanging de-identified data between hospitals for city-wide health analysis in the Chicago Area HealthLNK data repository (HDR). IRB Protocol, 2015.
[24] R. Popa and C. Redfield. CryptDB: protecting confidentiality with encrypted query processing. SOSP, pages 85–100, 2011.
[25] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In 2014 IEEE Symposium on Security and Privacy, pages 655–670. IEEE, 2014.
[26] P. Upadhyaya, M. Balasimha, and D. Suciu. Automatic enforcement of data use policies with DataLawyer. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 213–225. ACM, 2015.
[27] A. C. Yao. Protocols for secure computations. 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982), pages 1–5, 1982.
[28] S. Zahur and D. Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015.