Shrinkwrap: Efficient SQL Query Processing in Differentially Private Data Federations

Johes Bater
Northwestern University
johes@u.northwestern.edu

Xi He
University of Waterloo
xi.he@uwaterloo.ca

William Ehrich
Northwestern University
wdehrich@gmail.com

Ashwin Machanavajjhala
Duke University
ashwin@cs.duke.edu

Jennie Rogers
Northwestern University
jennie@northwestern.edu

ABSTRACT

A private data federation is a set of autonomous databases that share a unified query interface offering *in-situ* evaluation of SQL queries over the union of the sensitive data of its members. Owing to privacy concerns, these systems do not have a trusted data collector that can see all their data and their member databases cannot learn about individual records of other engines. Federations currently achieve this goal by evaluating queries obliviously using secure multi-party computation. This hides the intermediate result cardinality of each query operator by exhaustively padding it. With cascades of such operators, this padding accumulates to a blow-up in the output size of each operator and a proportional loss in query performance. Hence, existing private data federations do not scale well to complex SQL queries over large datasets.

We introduce Shrinkwrap, a private data federation that offers data owners a differentially private view of the data held by others to improve their performance over oblivious query processing. Shrinkwrap uses computational differential privacy to minimize the padding of intermediate query results, achieving up to a 35X performance improvement over oblivious query processing. When the query needs differentially private output, Shrinkwrap provides a trade-off between result accuracy and query evaluation performance.

PVLDB Reference Format:
Johes Bater, Xi He, Willi Ehrich, Ashwin Machanavajjhala, Jennie Rogers. Shrinkwrap: Differentially-Private Query Processing in Private Data Federations. *PVLDB*, 12(3): 307-320, 2018.
DOI: https://doi.org/10.14778/3291264.3291274

1. INTRODUCTION

The storage and analysis of data has seen dramatic growth in recent years. Organizations have never valued data more highly. Unfortunately, this value has attracted unwanted attention. Data breaches litter the news headlines, creating fear and a hesitance to share data, even among trusted collaborators. Without data sharing, information becomes siloed and enormous potential analytical value is lost.

Recent work in databases and cryptography attempts to solve the data sharing problem by introducing the *private data federation* [4]. A private data federation consists of a set of data owners who support a common relational database schema. Each party holds a horizontal partition (i.e., a subset of rows) of each table in the database. A private data federation provides a SQL query interface to analysts (clients) over the union of the records held by the data owners. Query evaluation is performed securely from multiple data owners without revealing unauthorized information to any party involved in the query and without the assistance of a trusted data curator.

A private data federation must provably ensure the following guarantee: a data owner should not be able to reconstruct the database (or a part of it) held by other data owners based on the intermediate result sizes of the query evaluation. One way to achieve this without a trusted data curator is to use secure computation protocols, which provide a strong privacy guarantee that intermediate results leak *no information* about their inputs.

Current implementations of private data federations that use secure computation have a performance problem: state-of-the-art systems have slowdowns of *4–6 orders of magnitude* over non-private data federations [4, 56]. This slowdown comes from the secure computation protocols used by private data federations to securely evaluate SQL queries without the need of a trusted third party. A query is executed as a directed acyclic graph of database operators. Each operator takes one or two inputs, applies a function, and outputs its result to the next operator. In a typical (distributed) database engine, the execution time needed to compute each intermediate result and the size of that result leaks information about the underlying data to curious data owners participating in the secure computation. To prevent this leakage, private data federations insert dummy tuples to pad intermediate results to their maximum possible size, thereby ensuring that execution time is independent of the input data. With secure evaluation, query execution runs in the worst-case, drastically increasing computation costs as intermediate result sizes grow. While performance is reasonable for simple queries and small workloads, performance is untenable for complex SQL queries with multiple joins.

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291274

Several approaches attempt to solve this performance problem, but they fail to provably bound the information leaked to a data owner. One line of research uses Trusted Execution Environments (TEEs) that evaluate relational operators within an on-chip secure hardware enclave [54, 56]. TEEs efficiently protect query execution, but they require specialized hardware from chip manufacturers. Moreover, current TEE implementations from both Intel and AMD do not adequately obscure computation, allowing observers to obtain supposedly secure data through widely publicized hardware vulnerabilities [9, 17, 28, 31, 49]. Another approach selectively applies homomorphic encryption to evaluate relational operators in query trees, while keeping the underlying tuples encrypted throughout the computation (e.g., CryptDB [45]). While this system improves performance, it leaks too much information about the data, such as statistics and memory access patterns, allowing a curious observer to deduce information about the true values of encrypted tuples [15, 26, 39, 7].

In this work, we bridge the gap between provable secure systems with untenable performance and practical systems with no provable guarantees on leakage using *differential privacy* [14]. Differential privacy is a state-of-the-art technique to ensure privacy, and provides a provable guarantee that one can not reconstruct records in a database based on outputs of a differentially private algorithm. Differentially private algorithms, nevertheless, permit approximate aggregate statistics about the dataset to be revealed.

We present Shrinkwrap, a system that improves private data federation performance by carefully relaxing the privacy guaranteed for data owners in terms of differential privacy. Instead of exhaustively padding intermediate results to their worst-case sizes, Shrinkwrap obliviously eliminates dummy tuples according to tunable privacy parameters, reducing each intermediate cardinality to a new, differentially private value. The differentially private intermediate result sizes are close to the true sizes, and thus, Shrinkwrap achieves practical query performance.

To the best of our knowledge, Shrinkwrap is the first system for private data sharing that combines differential privacy with secure computation for query performance optimization. The main technical contributions in this work are:

- A query processing engine that offers controlled information leakage with differential privacy guarantees to speed up private data federation query processing and provides tunable privacy parameters to control the trade-off between privacy, accuracy, and performance
- A computational differential privacy mechanism that securely executes relational operators, while minimizing intermediate result padding of operator outputs
- A novel algorithm that optimally allocates, tracks, and applies differential privacy across query execution
- A protocol-agnostic cost model that approximates the large, and non-linear, computation overhead of secure computation as it cascades up an operator tree

The rest of this paper is organized as follows. In Section 2 we define private data federations, outline our privacy goals and formally define secure computation and differential privacy. Section 3 describes the problem addressed by Shrinkwrap. Our end-to-end solution, Shrinkwrap, and its privacy guarantees, is described in Section 4. We show how to optimize the performance of Shrinkwrap in Section 5.

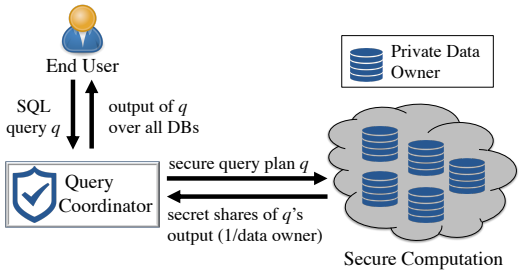


Figure 1: Private data federation architecture

Section 6 describes how we implement specific secure computation protocols on top of Shrinkwrap's protocol agnostic design. We experimentally evaluate our system implementation over real-world medical data in Section 7. We conclude with a survey of related work and future directions.

2. PRIVATE DATA FEDERATION

In this section, we formally define a private data federation (PDF), describe privacy goals and assumptions, and define two security primitives – secure computation and differential privacy.

A private data federation is a collection of autonomous databases that share a unified query interface for offering *in-situ* evaluation of SQL queries over the union of the sensitive data of its members without revealing unauthorized information to any party involved in the query. A private data federation has three types of parties: 1) two or more data owners $\text{DO}_1, \ldots, \text{DO}_m$ that hold private data $D_1, \ldots, D_m$ respectively, and where all D_i share a public schema of k relations $(R^1, \ldots, R^k)$; 2) a query coordinator that plans and orchestrates SQL queries q on the data of the data owners; and 3) a client that writes SQL statements to the query coordinator. The set of private data $\bar{D} = (D_1, \ldots, D_m)$ owned by the data owners is a horizontal partition of every table in the total data set D.

As shown in Figure 1, the client first passes an SQL statement $q(\cdot)$ to the query coordinator, who compiles the query into an optimized secure query plan to be executed by each data owner. Once the data owners finish execution, they each pass their secret share of the output to the query coordinator, who assembles and returns the result to the client.

2.1 Privacy Goals and Assumptions

Table 1 summarizes the privacy goals for all parties involved in this process and the necessary assumptions required for achieving these goals.

Privacy Goals: Data owners require a privacy guarantee that the other parties (data owners, query coordinator and the client) are not able to reconstruct the private data they hold based on intermediate results of the query execution. In particular, we require data owners to have a differentially private view over the inputs of other data owners (formally defined in Def. 2). We aim to support two policies for clients. Clients may be trusted (output policy 1) and allowed to see true answers to queries but must not learn any other information about the private inputs held by data owners. Clients may be untrusted (output policy 2), in which case they only are permitted a differentially private view over the inputs. The query coordinator is an extension of the client and has the same privacy requirements.

Table 1: Privacy Goals and Assumptions

	Client Sees True Answers	**Client Sees Noisy Answers**
Privacy Goal for DOs	Differentially private (DP) view over data held by other DOs	
Privacy Goal for Client	Learns only query answer	DP view over data held by DOs
Client & DO Assumption	Semi-honest & Computationally Bounded	
DOs Colludes with Client	Not allowed	Allowed
Client Colludes with some DOs	Not allowed	Allowed

Assumptions: To achieve the privacy goals presented above, a private data federation assumes that all parties are computationally bounded and work in the semi-honest setting. Hence, the query coordinator honestly follows the protocols and creates a secure plan to evaluate the query. All parties faithfully execute the cryptographic protocol created by the query coordinator, but may attempt to infer properties of private inputs held by other parties by observing the query instruction traces and data access patterns. The query coordinator is also assumed to be memory-less as it destroys its contents after sending query results back to the client. When the client sees the true answers (output policy 1), we assume that the client is trusted to not collude with data owners; otherwise, if the client shares the true answer with a data owner (or publishes the true answer), the data owner who colludes with the client can use their own private data and the true query answer to infer the input of the other data owners. Similarly, we assume the data owners do not share their true input with the client; otherwise, the client can infer the other data owners' input and gain more information than just the query answer. When the client is not trusted, we assume that the client may collude with (as many as all but one) data owners. Our guarantees hold even in that extreme case.

2.2 Security Primitives

Secure Computation: To securely compute functions on distributed data owned by multiple parties, secure computation primitives are required. Let $f : \mathcal{D}^m \rightarrow \mathcal{R}^m$ be a functionality[1] over $\bar{D} = (D_1, \ldots, D_m) \in \mathcal{D}^m$, where $f_i(\bar{D})$ denotes the i-th element of $q(\bar{D})$. Let Π be an m-party protocol for computing f and $\text{VIEW}_i^{\Pi}(\bar{D})$ be the view of the i-th party during an execution of Π on $\bar{D}$. For $I = \{i_1, \ldots, i_t\} \subseteq [m]$, we let $D_I = (D_{i_1}, \ldots, D_{i_t})$, $f_I(\bar{D}) = (q_{i_1}(\bar{D}), \ldots, q_{i_t}(\bar{D}))$, and $\text{VIEW}_I^{\Pi}(\bar{D}) = (I, \text{VIEW}_{i_1}^{\Pi}(\bar{D}), \ldots, \text{VIEW}_{i_1}^{\Pi}(\bar{D}))$. We define secure computation with respect to semi-honest behavior [19] as follows.

Definition 1 (secure computation). *We say an m-party protocol securely computes $f : \mathcal{D}^m \rightarrow \mathcal{R}^m$ if there exists a probabilistic polynomial-time algorithm denoted S and* $\text{negl}(\kappa)$ *such that for any non-uniform probabilistic polynomial adversary A and for every $I \subseteq [m]$, every polynomial $p(\cdot)$, every sufficiently large $\kappa \in \mathbb{N}$, every $\hat{D} \in \mathcal{D}^m$ of size at most $p(\kappa)$, it holds that*

$$|\Pr[A(S(I, D_I, f_I(\bar{D})), f(\bar{D})) = 1] - \Pr[A(\text{VIEW}_I^{\Pi}(\bar{D}), \text{OUTPUT}^{\Pi}(\bar{D})) = 1]| \leq \text{negl}(\kappa), \quad (1)$$

where $\text{negl}(\kappa)$ *refers to any function that is $o(\kappa^{-c})$, for all constants c and* $\text{OUTPUT}^{\Pi}(\bar{D})$ *denotes the output sequence of all parities during the execution represented in* $\text{VIEW}_I^{\Pi}(\bar{D})$.

[1]Functionality f can be both deterministic or randomized

Various cryptographic protocols that enable secure computation have been studied [5, 10, 11, 18, 46, 33, 52, 55] and several are applicable to relational operators [4, 8, 50].

Differential Privacy (DP): Differential privacy [14] is an appealing choice to bound the information leakage on the individual records in the input while allowing multiple releases of statistics about the data. This privacy notion is utilized by numerous organizations, including the US Census Bureau [34], Google [16], and Uber [27]. Formally, differential privacy is defined as follows.

Definition 2 ((ϵ, δ)-Differential Privacy). *A randomized mechanism $f : \mathcal{D} \rightarrow \mathcal{R}$ satisfies (ϵ, δ)-differential privacy if for any pair of neighboring databases $D, D' \in \mathcal{D}$ such that D and D' differ by adding or removing a row and any set $O \subseteq \mathcal{R}$,*

$$\Pr[f(D) \in O] \leq e^{\epsilon} \Pr[f(D') \in O] + \delta. \quad (2)$$

The differential privacy guarantee degrades gracefully when invoked multiple times. In the simplest form, the overall privacy loss of multiple differentially private mechanisms can be bounded with sequential composition [13].

Theorem 1 (Sequential Composition). *If M_1 and M_2 are (ϵ_1, δ_1)-DP and (ϵ_2, δ_2)-DP algorithms that use independent randomness, then releasing the outputs $M_1(D)$ and $M_2(D)$ on database D satisfies $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$-DP.*

There exist advanced composition theorems that give tighter bounds on privacy loss under certain conditions [14], but we use sequential composition as defined above as it is general and easy to compute.

3. PROBLEM STATEMENT

Our goal is to build a system that permits efficient query evaluation on private data federations while achieving provable guarantees on the information disclosed to clients and data owners. As discussed in Section 1, prior work is divided into two classes: (i) systems that use secure computation to answer queries, which give formal privacy guarantees but suffer *4–6 orders of magnitude* slowdowns compared to non-private data federations [4, 56], and (ii) systems that use secure hardware [54, 56] or property preserving encryption [45], which are practical in terms of performance, but have no formal guarantee on the privacy afforded to clients and data owners.

While secure computation based solutions are attractive, and there is much recent work in improving their efficiency [6, 24, 25], secure computation solutions fundamentally limit performance due to several factors: 1) a worst-case running time is necessary to avoid leaking information from early termination; 2) computation must be *oblivious*, i.e., the program counters and the memory traces must be independent of the input; 3) cryptographic keys must be sent and computed on by data owners, the cost of which scales with the complexity of the program.

In particular, applying secure computation protocols to execute SQL queries results in extremely slow performance. A SQL query is a directed acyclic graph of database operators. Each operator in the tree takes an input, applies a

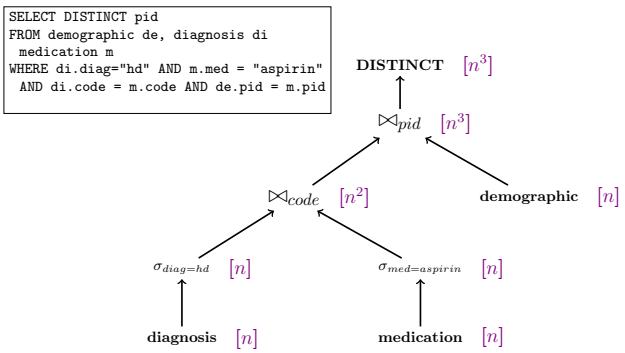


Figure 2: Exhaustive padding of intermediate results in an oblivious query evaluation

function, and outputs its result to the next operator. The execution time needed to calculate each intermediate result and the array size required to hold that result leaks information about the underlying data. To address this leakage, private data federations must insert dummy tuples to pad intermediate results to their maximum possible size and ensure that execution time is independent of the input data. We demonstrate this property in the following example.

EXAMPLE 1. *Figure 2 shows a SQL query and its corresponding operator tree. Tuples flow from the source relations through a filter operator, before entering two successive join operators and ending at a distinct operator. To hide the selectivity of each operator, the private data federation pads each intermediate result to its maximum value. At the filters, the results are padded as if no tuples were filtered out, remaining at size n. This is a significant source of additional computation. If the selectivity of a filter is 10^{-3}, our padding gives a $1000\times$ performance overhead. As the tuples flow into the first join, which now receives two inputs of size n, exhaustive padding produces an output of n^2 tuples. When this result passes through the next join, the result size that the distinct operator must process jumps to n^3. If we once again assume a 10^{-3} selectivity at the joins, we now see a $10^9\times$ computation overhead. Such a significant slowdown arises from the cumulative effect of protecting cascading operators. By exhaustively padding operator outputs, we see ever-increasing intermediate result cardinalities and, as a function of that growth, ever-decreasing performance.*

To tackle this fundamental performance challenge, we formalize our research problems as follows.

- Build a system that uses differentially private leakage of intermediate results to speed up SQL query evaluation while achieving all the privacy goals stated in Section 2.
- Given an SQL operator o and a privacy budget (ϵ, δ), design an (ϵ, δ)-differentially private mechanism that executes the operator with a smaller performance overhead compared to fully oblivious computation.
- Given a sequence of operators $Q = \{o_1, \ldots, o_l\}$, design an algorithm to optimally split the privacy budget (ϵ, δ) among these operators during query execution.

In the following sections, we present our system for efficient SQL query processing, starting with the differentially private mechanisms and overall privacy analysis used for operator evaluation (Section 4), and then our budget splitting algorithm that optimizes performance (Section 5).

4. SHRINKWRAP

In this section, we present the end-to-end algorithm of Shrinkwrap for answering a SQL query in a private data federation, the key differentially private mechanism for safely revealing the intermediate cardinality of each operator, and the overall privacy analysis of Shrinkwrap.

4.1 End-to-end Algorithm

At its core, Shrinkwrap is a system that applies differential privacy throughout SQL query execution to reduce intermediate result sizes and improve performance. The private data federation ingests a query q as a directed acyclic graph (DAG) of database operators $Q = \{o_1, \ldots, o_l\}$, following the order of a bottom up tree traversal. The query coordinator decides the output policy based on the client type and assigns the privacy parameters (ϵ, δ) for the corresponding privacy goals. To improve performance, a privacy budget of $(\epsilon_{1\to l}, \delta_{1\to l})$ will be spent on protecting intermediate cardinalities of the l operators. When the client is allowed to learn the true query output (output policy 1), then $\epsilon_{1\to l} = \epsilon$ and $\delta_{1\to l} = \delta$; when the client is allowed to learn noisy query answers from a differentially private mechanism, then $\epsilon_{1\to l} < \epsilon$ and $\delta_{1\to l} < \delta$, where the remaining privacy budget $(\epsilon - \epsilon_{1\to l}, \delta - \delta_{1\to l})$ will be spent on the query output.

Algorithm 1 outlines the end-to-end execution of a SQL query in Shrinkwrap. Inputs to the algorithm are the query Q, the overall privacy parameters (ϵ, δ), the privacy budget for improving the performance $(\epsilon_{1\to l}, \delta_{1\to l})$, and the public information K, including the relational database schema and the maximum possible input data size at each party.

First, the query coordinator allocates the privacy budget for improving performance $(\epsilon_{1\to l}, \delta_{1\to l})$ among the operators in Q such that

$$\sum_{i=1}^{l} \epsilon_i = \epsilon_{1\to l} \text{ and } \sum_{i=1}^{l} \delta_i = \delta_{1\to l} \tag{3}$$

where (ϵ_i, δ_i) is the privacy budget allocated to operator o_i. If $\epsilon_i = \delta_i = 0$, then the operator is evaluated obliviously (without revealing the intermediate result size). When $\epsilon_i, \delta_i > 0$, Shrinkwrap is allowed to reveal an overestimate of the intermediate result size, with larger (ϵ_i, δ_i) values giving tighter intermediate result estimates. The performance improvement of Shrinkwrap highly depends on how the budget is split among the operators, but any budget split that satisfies Equation 3 satisfies privacy. We show several allocation strategies in Section 5 and analyze them in the evaluation.

Next, the query coordinator compiles a secure query plan to be executed by the data owners. The secure query plan traverses the operators in Q. For each operator o_i, data owners compute secret shares of the true output using secure computation over the federated database and the output of other operators computed from D. We denote this secure computation for o_i by $SMC(o_i, D)$. The true outputs (of size c_i) are placed into a secure array O_i. The secure array is padded (with encrypted dummy values) so that its size equals the maximum possible output size for operator o_i.

Then, Shrinkwrap calls a (ϵ_i, δ_i)-differentially private function $Resize()$ to resize this secure array O_i to a smaller array S_i such that a subset of the (encrypted) dummy records are removed. Resizing is described in Section 4.2. Once all the operators are securely evaluated, if the output policy allows release of true answers to the client (policy 1), the last secure

Algorithm 1: End-to-end Shrinkwrap algorithm

Input: PDF query DAG with operators $Q = \{o_1, \ldots, o_l\}$, the federated database D, public information K, privacy parameters (ϵ, δ), privacy budget for performance $(\epsilon_{1\to l} \leq \epsilon, \delta_{1\to l} \leq \delta)$
Output: Query output R

//Computed on query coordinator
$P = \{(\epsilon_1, \delta_1), \ldots, (\epsilon_l, \delta_l) | \sum_{i=1}^{l} \epsilon_i = \epsilon_{1\to l}, \sum_{i=1}^{l} \delta_i = \delta_{1\to l}\}$
$\leftarrow$ AssignBudget$(\epsilon_{1\to l}, \delta_{1\to l}, Q, K)$ *// (Sec. 5)*

//Computed on private data owners
for $i \leftarrow 1$ **to** l **do**
  $(O_i, c_i) \leftarrow$ SMC(o_i, D) *//with exhaustive padding*
  $S_i \leftarrow$ Resize$(O_i, c_i, \epsilon_i, \delta_i, \Delta\mathbf{c}_i)$ *// DP resizing (Sec. 4.2)*
end
if $\epsilon = \epsilon_{1\to l}$ *and* $\delta = \delta_{1\to l}$ **then**
  Send query coordinator S_l *//output policy 1*
else
  Query output budget $\epsilon_0 \leftarrow (\epsilon - \epsilon_{1\to l})$, $\delta_0 \leftarrow (\delta - \delta_{1\to l})$
  Send query coordinator $M^{(\epsilon_0, \delta_0)}(S_l)$ *//output policy 2*
end

//Computed on query coordinator
$R \leftarrow$ Assemble(S_l)
return R to client

function Resize(O, c, ϵ, δ, $\Delta\mathbf{c}$)
  $\tilde{c} \leftarrow c +$ TLap$(\epsilon, \delta, \Delta\mathbf{c})$ *// perturb cardinality*
  $O \leftarrow$ Sort(O) *//obliviously sort dummy tuples to the end*
  $S \leftarrow$ new SecureArray$(O[1, \ldots, \tilde{c}])$
  return S

array S_l will be directly returned to the query coordinator; otherwise, the data owners jointly and securely compute a differentially private mechanism on S_l with the remaining privacy budget $(\epsilon_0 = \epsilon - \epsilon_{1\to l}, \delta_0 = \delta - \delta_{1\to l})$, denoted by $M^{(\epsilon_0, \delta_0)}(S_l)$, and return the output of $M^{(\epsilon_0, \delta_0)}(S_l)$ to the query coordinator. For instance, one could use a Laplace mechanism to perturb count queries using a multiparty protocol for generating noise as in [38]. Finally, the query coordinator assembles the final secure array received from the data owners S_l, decrypting the final result R and returning it to the client.

4.2 DP Resizing Mechanism

For each operator o_i, Shrinkwrap first computes the true result using secure computation and places it into a exhaustively padded secure array O_i. For example, as shown in Figure 3, a join operator with two inputs, each of size n, will have a secure array O_i of size n^2. This is the exhaustive padding case shown on the left side of the figure, with the private data federation inserting the materialized intermediate results into the secure array and padding those results with dummy tuples. The Resize() function takes in the exhaustively padded secure array O_i, the true cardinality of the output c_i, the privacy parameters (ϵ_i, δ_i), and the sensitivity of the cardinality query at operator o_i, denoted by $\Delta\mathbf{c}_i$. The cardinality query $\mathbf{c}_i(D)$ returns the output of the operator o_i in the PDF query q. The sensitivity of a query is defined as follows.

Definition 3 (Query Sensitivity). *The sensitivity of a function $f : \mathcal{D} \to \mathbb{R}^*$ is the maximum difference in its output for any pairs of neighboring databases D and D',*

$$\Delta f = \max_{\{D, D' \ s.t. \ |D - D' | \cup |D' - D| = 1\}} \|f(D) - f(D')\|_1$$

Based on these inputs, the Resize() function runs a (ϵ_i, δ_i)-differentially private truncated Laplace mechanism (Def. 4) to obtain a new, differentially private cardinality $\tilde{\mathbf{c}}_i > \mathbf{c}_i$ for the secure array. Next, this function sorts the exhaustively padded secure array O_i obliviously such that all dummy tuples are at the end of the array and then copies the first $\tilde{\mathbf{c}}_i$ tuples in the sorted array into a new secure array S_i of size $\tilde{\mathbf{c}}_i$. This new secure array S_i of a smaller size than O_i is used for the following computations.

Noise Generation: Shrinkwrap generates differentially private cardinalities by using a truncated Laplace mechanism. Given an operator o, with a privacy budget (ϵ, δ) and the sensitivity parameter for the cardinality query $\Delta\mathbf{c}$, this mechanism computes a noisy cardinality for use in Algorithm 1.

Definition 4 (Truncated Laplace Mechanism). *Given a query $\mathbf{c} : \mathcal{D} \to \mathbb{N}$, the truncated Laplace mechanism TLap$(\epsilon, \delta, \Delta\mathbf{c})$ adds a non-negative integer noise $\max(\eta, 0)$ to the true query answer $\mathbf{c}(D)$, where $\eta \in \mathbb{Z}$ follows a distribution, denoted by $L(\epsilon, \delta, \Delta\mathbf{c})$ that has a probability density function $\Pr[\eta = x] = p \cdot e^{-(\epsilon/\Delta\mathbf{c})|x - \eta^0|}$, where $p = \frac{e^{\epsilon/\Delta\mathbf{c}} - 1}{e^{\epsilon/\Delta\mathbf{c}} + 1}$, $\eta^0 = -\frac{\Delta\mathbf{c} \cdot \ln((e^{\epsilon/\Delta\mathbf{c}} + 1)\delta)}{\epsilon} + \Delta\mathbf{c}$.*

This mechanism allows us to add non-negative integer noise to the intermediate cardinality query which corresponding to the number of the dummy records. The noise η drawn from $L(\epsilon, \delta, \Delta\mathbf{c})$ satisfies that $\Pr[\eta < \Delta\mathbf{c}] \leq \delta$, which enables us to show the privacy guarantee of this mechanism.

Theorem 2. *The truncated Laplace mechanism satisfies (ϵ, δ)-differential privacy.*

Proof. For any neighboring databases D_1, D_2, let $c_1 = \mathbf{c}(D_1) \geq 0$ and $c_2 = \mathbf{c}(D_2) \geq 0$. W.L.O.G. we consider $c_2 \geq c_1$. It is easy to see that $\Pr[\text{TLap}(D_1) \in (-\infty, c_1)] = 0$ and $\Pr[\text{TLap}(D_2) \in (-\infty, c_2)] = 0$. By the noise property, for any $o \geq c_2 \geq c_1$, it is true that $|\ln \frac{\Pr[\text{TLap}(D_2) = o]}{\Pr[\text{TLap}(D_1) = o]}| \leq \epsilon$. However, when the output $o \in [c_1, c_2)$, $\Pr[\text{TLap}(D_2) = o] = 0$, but the $\Pr[\text{TLap}(D_2) = o] > 0$, making the ratio of probabilities unbounded. Nevertheless, we can bound $\Pr[\text{TLap}(D_2) \in [c_1, c_2)]$ by δ as shown below.

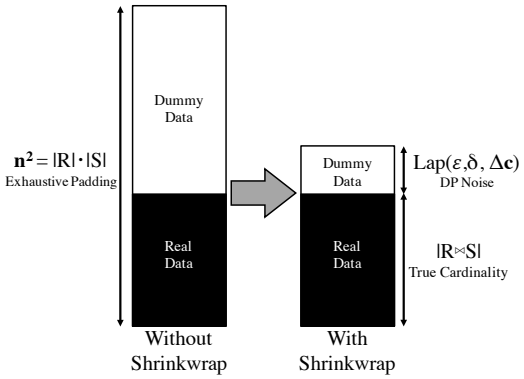


Figure 3: Effect of Shrinkwrap on Intermediate Result Sizes When Joining Tables R and S

Let $O^* = (-\infty, c_2)$. We can show that for any output set O of query $\mathbf{c}()$, we have

$$\begin{aligned} & \Pr[\text{TLap}(D_1) \in O] \\ = \; & \Pr[\text{TLap}(D_1) \in (O \cap O^*)] + \Pr[\text{TLap}(D_1) \in (O - O^*)] \\ \leq \; & \Pr[\text{TLap}(D_1) \in [c_1, c_2)] + e^{\epsilon} \Pr[\text{TLap}(D_2) \in (O - O^*)] \\ = \; & \Pr[\eta_1 < \Delta\mathbf{c}] + e^{\epsilon} \Pr[\text{TLap}(D_2) \in O] \\ = \; & \delta + e^{\epsilon} \Pr[\text{TLap}(D_2) \in O] \end{aligned} \quad (4)$$

Hence, this mechanism satisfies (ϵ, δ)-DP. □

Sensitivity Computation: In order to create our noisy cardinalities using the truncated Laplace mechanism in Definition 4, we must determine the cardinality query sensitivity at the output of each operator in the query tree. An intermediate result cardinality, $\mathbf{c}_i$, is the output size of the sub-query tree rooted at operator o_i. The sensitivity of $\mathbf{c}_i$ depends on the operators in its sub-query tree, and we compute this as a function of the stability of operators in the sub-query tree. The stability of an operator bounds the change in the output with respect to the change in the input of the operator.

Definition 5 (Stability [36]). *A transformation operator is s-stable if for any two input data sets A and B,*

$$|T(A) \ominus T(B)| \leq s \times |A \ominus B|$$

For unary operators, such as SELECT, PROJECT, the stability usually equals one. For JOIN operators, a binary operators, the stability is equal to the maximum multiplicity of the input data. The sensitivity of the intermediate cardinality query $\mathbf{c}_i$ can be recursively computed bottom up, starting from a single change in the input of the leaf nodes.

Example 2. *Given a query tree shown in Figure 4, the bottom two filter operators each have a stability of one, giving a sensitivity ($\Delta\mathbf{c}$) of 1 at that point. Next, the first join has a stability of m for the maximum input multiplicity, which increases the overall sensitivity to m. The following join operator also has a stability of m, which changes the sensitivity to m^2. Finally, the DISTINCT operator has a stability of 1, so the overall sensitivity remains m^2.*

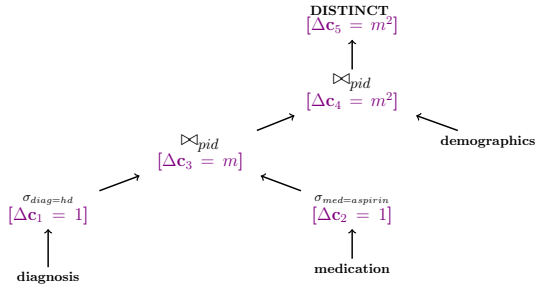


Figure 4: Sensitivity analysis for running example

Using these principles, Shrinkwrap calculates the sensitivity for each operator in a query tree for use in Laplace noise generation during query execution. More details can be seen in PINQ [36] for computing the sensitivity of SQL queries.

Secure Array Operations: Each time Shrinkwrap adds or removes tuples, it normally needs to pay an I/O cost for accessing the secure array. We avoid paying the full cost by carrying out a bulk unload and load. To start the *Resize*() function, Shrinkwrap takes an intermediate result secure array and sorts it so that all dummy tuples are at the end of the secure array. Next, Shrinkwrap cuts off the end of the original secure array and copies the rest into the new, differentially-private sized secure array. Since the new secure array has a size guaranteed to be larger than the true cardinality, we know that cutting off the end will only remove dummy tuples. Thus, we avoid paying the full I/O cost for each tuple. However, we do pay an $O(n \log(n))$ cost for the initial sorting, as well as an $O(n)$ cost for bulk copying the tuples. This algorithm is an extension of the one in [21] and variants of it are in [40, 41, 56].

4.3 Overall Privacy Analysis

Given a PDF query DAG q with operators $\{o_1, \ldots, o_l\}$ and a privacy parameter (ϵ, δ), Algorithm 1 achieves the privacy goals stated in Section 2.

Theorem 3. *Under the assumptions in Section 2, Algorithm 1 achieves the privacy goals: data owners have a (ϵ, δ)-computational differentially private view over the input data of other data owners; when true answers are returned to the client, the client learns nothing else; when noisy answers are returned the client, the client has a (ϵ, δ)-computational differentially private view over the input data of all the data owners.*

Proof. (sketch) We prove that the view of data owners satisfies computational differential privacy [37, 48] by showing that the view of any data owner is computationally indistinguishable from its view in an ideal simulation that outputs *only* the differentially private cardinalities of each operator output.

Let *Sim* be a simulator that takes as input the horizontal partitions held by the data owners $(D_1, ..., D_m)$, and outputs a vector $S_i^{\perp}$ for operator o_i such that: (a) $|S_i^{\perp}|$ is drawn from $\mathbf{c}_i + \text{TLap}()$, and (b) $S_i^{\perp}$ contains encryptions of 0 (of the appropriate arity). It is easy to see that a function computed by *Sim* satisfies $(\epsilon_{1\to l}, \delta_{1\to l})$-DP: (a) at each operator, the release of $|S_i^{\perp}|$ satisfies (ϵ_i, δ_i)-DP, and (b) By sequential composition (Theorem 1) and Equation 3, releasing all the cardinalities satisfies $(\epsilon_{1\to l}, \delta_{1\to l})$-DP. When the output itself is differentially private, there is an additional privacy loss to data owners (since the client could release the output to the data owners), but the overall algorithm still satisfies (ϵ, δ)-DP by sequential composition.

The real execution outputs secure arrays S_i at each operator o_i using secure computation. This ensures that the view of a data owner in the real execution is computationally indistinguishable from the execution of *Sim*, which satisfies (ϵ, δ)-DP. Thus, we can show that the view of any data owner satisfies (ϵ, δ)-computational DP. The privacy guaranteed to the client and query coordinator can be analogously argued. □

4.4 Multiple SQL Queries

The privacy loss of Shrinkwrap due to answering multiple queries is analyzed using sequential composition (Theorem 1). There are two approaches to minimize privacy loss across multiple queries: (i) applying advanced composition theorems that give tighter bounds on privacy loss under certain conditions [14] or (ii) optimizing privacy budget allocation across the operators of a workload of predefined SQL

queries (e.g. using [35]). These two approaches are orthogonal to our work and adapting these ideas to the context of secure computation is an interesting avenue for future work.

5. PERFORMANCE OPTIMIZATION

A key technical challenge is to divide the privacy budget for performance $(\epsilon_{1\to l}, \delta_{1\to l})$ across the different operators so the overall runtime of the query execution is minimized.

PROBLEM 1. *Given a privacy budget $(\epsilon_{1\to l}, \delta_{1\to l})$, a PDF query DAG q with operators $Q = \{o_1, \ldots, o_l\}$, and public information K which includes the sensitivities of the intermediate cardinality queries $\{\Delta\mathbf{c}_1, \ldots, \Delta\mathbf{c}_l\}$ and the schema information, compute the share (ϵ_i, δ_i) assigned to o_i such that: (a) the assignment ensures privacy, i.e., $\sum_{i=1}^{l} \epsilon_i = \epsilon_{1\to l}$, and (b) the overall run time of the query is minimized.*

5.1 Baseline Privacy Budget Allocation

We first consider two baseline strategies for allocating the privacy budget to individual operators.

Eager: This approach allocates the entire budget $(\epsilon_{1\to l}, \delta_{1\to l})$ to the first operator in the query tree. In Figure 2, this equates to minimizing the output cardinality of the filter on diagnoses and executing the rest of the tree obliviously. A benefit of this approach is that since the intermediate results of the first operator flow through the rest of the tree, every operator will see the benefit of minimizing the output cardinality. On the other hand, in some trees, certain operators, like joins, have an output-sized effect on the total query performance. Not allocating any privacy budget to those operators results in a missed optimization opportunity.

Uniform: The second approach splits the privacy budget $(\epsilon_{1\to l}, \delta_{1\to l})$ uniformly across the query tree, resulting in equal privacy parameters for each operator. The benefit here is that every single operator receives part of the budget, so we do not lose out on performance gains from any operator. The drawback of the uniform approach, like the one-time approach, is that some operators produce larger effects, so by not allocating more of the budget to them, we lose out on potential performance gains.

5.2 Optimal Privacy Budget Allocation

We design a third approach which uses an execution cost model as an objective function and applies convex optimization to determine the optimal budget splitting strategy.

Cost Model: The execution cost of a query in Algorithm 1 mainly consists of (a) the cost for securely computing each operator and (b) the I/O cost of handling each intermediate result by Shrinkwrap. We model these costs as functions of the corresponding input and output data sizes. The cost of secure computation at operator o_i is represented by a function $cost_{o_i}(\mathbf{N}_i)$, where $\mathbf{N}_i$ is the set of the cardinalities of the input tables. The I/O cost of Shrinkwrap handling each intermediate result of operator o_i is mainly the cost to sort the exhaustively padded array of size n' and the cost of copying the array of size n to the new array of size n', denoted by $cost_s(n)$ and $cost_c(n, n')$ respectively. In particular, the size of the exhaustively padded array also depends on the sizes of the input tables, represented by a function $o_i(\mathbf{N}_i)$. Most of the time, the size of the exhaustively padded array is the product of the input table sizes. For example, if operator o_i takes one input table of size n_1, then $o_i(\mathbf{N} = \{n_1\}) = n_1$; if operator o_i takes two input tables of size n_1 and n_2, then $o_i(\mathbf{N} = \{n_1, n_2\}) = n_1 \cdot n_2$. The number of tuples n' copied to the new array depends on the noisy output size n_i.

The input tables to an operator can be a mix of base tables and the intermediate output tables. Given a database of k relations $(R_1, \ldots, R_k)$, there will be an additional l number of intermediate output tables generated in a query tree of l operators. The public knowledge K contains the table sizes for the original tables, but the output cardinalities of the operators are unknown. In the private setting, we cannot use the true cardinality for the estimation. Instead, Shrinkwrap uses the naive reduction factors from [47] to estimate the output cardinality of each operator, denoted by $estimate(\mathbf{c}_i, K)$. The noise η_i added to the cardinality depends on the noise distribution of the truncated Laplace mechanism and we use the expectation of the distribution $\text{TLap}(\epsilon_i, \delta_i, \Delta\mathbf{c}_i)$ to model the noise. Putting this together, the output cardinality n_i at operator o_i is estimated by $estimate(\mathbf{c}_i, K) + \mathbb{E}(\text{TLap}(\epsilon_i, \delta_i, \Delta\mathbf{c}_i))$.

With this formulation, the cost of assigning privacy budget $P = \{(\epsilon_i, \delta_i) | i = 1, \ldots, l\}$ to operators $Q = \{o_1, \ldots, o_l\}$ in the query tree is modeled as

$$C(P, K) = \sum_{i=1}^{l} cost_{o_i}(\mathbf{N}_i) + cost_s(o_i(\mathbf{N}_i)) + cost_c(o_i(\mathbf{N}_i), n_i) \tag{5}$$

where $n_i = estimate(\mathbf{c}_i, K) + \mathbb{E}(\text{TLap}(\epsilon_i, \delta_i, \Delta q_i))$.

Optimization: Using this cost model, we find the optimal solution to the following problem as the budget allocation strategy for Algorithm 1.

$$\min_P C(P) \quad s.t. \quad \sum_{i=1}^{l} \epsilon_i \le \epsilon_{1\to l}, \sum_{i=1}^{l} \delta_i \le \delta_{1\to l}, \quad \epsilon_i, \delta_i \ge 0 \ \ \forall i = 1, \ldots l \tag{6}$$

We show the detailed cost model including the unit cost function $cost_{o_i}(\mathbf{N}_i, n_i)$, $cost_s(o_i(\mathbf{N}_i))$, and $cost_c(o_i(\mathbf{N}_i), n_i)$ for two different secure computation protocols in the next section. Using a convex optimization solver, we determine the optimal budget split that minimizes the cost objective function. We evaluate how it performs against the baseline approaches in Section 7.

6. PROTOCOL IMPLEMENTATION

Shrinkwrap is a flexible computation layer that supports a wide range of private data federations and database engines. The only requirements are that the private data federation process queries as operator trees and that the underlying database engine supports SQL queries. Shrinkwrap implements a wide range of SQL operators for PDF queries, including selection, projection, aggregation, limit, and some window aggregates. For joins, we handle equi-joins and cross products. At this time we do not support outer joins or set operations. For output policy 2 where client sees noisy answers released from differentially private mechanisms, we support aggregate queries like `COUNT` for the final operator.

The fundamental design pattern of secure computation protocols is the circuit model, where functions are represented as circuits, i.e. a collection of topologically ordered gates. The benefit of this model is that by describing how to securely compute a single gate, we can compose the computation across all gates to evaluate any circuit, simplifying

the design of the protocol. The downside is that circuits are difficult for programmers to reason about. Instead, the von Neumann-style Random Access Machine (RAM) model handles the impedance mismatch between programmers and circuits by creating data structures, such as ORAM, that utilize circuit-based oblivious algorithms for I/O. With the RAM model, programmers no longer have to write programs as circuits. Instead, they can think of their data as residing in secure arrays that can be plugged into their existing code. The downside of the RAM model is that since the data structures are more general purpose, performance lags behind direct circuit model implementations of the same function. In this work, we use Shrinkwrap with both RAM model and circuit model protocols, providing a general framework that can easily add new protocols to the query executor.

6.1 RAM Model

When integrating a new protocol, we first create an operator cost model that captures the execution cost of each operator. For a RAM based protocol, we model the cost of secure computations $cost_o(\mathbf{N})$ for commonly used relational operators in Table 2, where $\mathbf{N}$ is the set of cardinalities of input tables of operator o. The cost function considered in this work focuses on the I/O cost of each relational operator in terms of reads and writes from a secure array.

We let $c_{read}(n)$ and $c_{write}(n)$ be the unit cost for reading and writing of a tuple from a secure array of size n respectively. Secure arrays protect their contents from attackers by guaranteeing that all reads or writes access the same number of memory locations. Typically, the data structures will shuffle either their entire contents or their access path on each access, with different implementations providing performance ranging from $O(\log n)$ to $O(n \log^2(n))$ per read or write [2, 53]. This variable I/O cost guarantees that an observer cannot look at memory access time or program traces to infer sensitive information.

Given the unit cost for reading and writing, we can represent the cost of secure computation for each operator as the sum of reading cost and writing cost. For example, the cost function for a Filter operator $cost_{Filter}(\mathbf{N} = \{n_1\})$ is the cost of reading n_1 number of input records and writing n_1 number of output records. For a Join operator over two input tables, the cost function $cost_{Join}(\mathbf{N} = \{n_1, n_2\})$ equals to the sum of n_1 number of records reading from the first secure array of size n_1, $(n_1 * n_2)$ number of record reading from the second array of size n_2, and $(n_1 * n_2)$ number of record writes, i.e. $n_1 * c_{read}(n_1) + n_1 * n_2 * c_{read}(n_2) + n_1 * n_2 * c_{write}(n_1 * n_2)$. The other operators are modeled similarly in Table 2. The cost function for copying from an array of size n to an array of size n' can be modeled as $cost_c(n, n') = n' * c_{read}(n) + n' * c_{write}(n')$.

In this work, we use ObliVM [32] to implement our RAM based relational operators, under a SMCQL [4] private data federation design. We chose ObliVM due to its compiler, which allows SMCQL to easily convert operators written as C-style code into secure computation programs. However, Shrinkwrap can support more recent ORAM implementations such as DORAM [12] by summarizing their cost behavior as in Table 2 and using those operator costs in our cost model. In SMCQL, private intermediate results are held in secure arrays, i.e., oblivious RAM (ORAM), and operators are compiled as circuits that receive an ORAM as input, evaluate the operator, and output the result ORAM to the next operator. With Shrinkwrap, we resize the intermediate ORAM arrays by obliviously eliminating excess dummy tuples, reducing the intermediate cardinalities of our results and improving performance. Any ORAM implementation is compatible with Shrinkwrap, as long as its I/O characteristics can be similarly defined as in Table 2.

Table 2: Operator I/O cost

Operator	$cost_o(\mathbf{N} = \{n_1, n_2, \ldots\})$
Filter	$n_1 * c_{read}(n_1) + n_1 * c_{write}(n_1)$
Join	$n_1 * c_{read}(n_1) + n_1 * n_2 * c_{read}(n_2)$ $+n_1 * n_2 * c_{write}(n_1 * n_2)$
Aggregate	$n_1 * c_{read}(n_1) + c_{write}(n_1)$
Aggregate, Group By	$n_1 * c_{read}(n_1) + n_1 * c_{write}(n_1)$
Window Aggregate	$n_1 * c_{read}(n_1) + n_1 * c_{write}(n_1)$
Distinct	$n_1 * c_{read}(n_1) + n_1 * c_{write}(n_1)$
Sort	$n_1 * \log^2(n_1) * (c_{read}(n_1) + c_{write}(n_1))$

6.2 Circuit Model

Alternatively, if a private data federation uses a circuit based protocol to express database operators, we can estimate the cost of an operator o, $cost_o(\mathbf{N})$, with n_{gates} gates in its circuit, a maximum circuit depth of $d_{circuit}$, n_{in} elements of input, and an output size of n_{out} as:

$$cost_{op}(\mathbf{N}) = c_{in} * n_{in} + c_g * n_{gates} + c_d * d_{circuit} + c_{out} * n_{out},$$

where c_{in} and c_{out} account for the encoding and decoding costs needed for a given protocol, c_g and c_d account for the costs due to the gate count and circuit depth respectively, the number of input element is $n_{in} = \sum_{n_j \in \mathbf{N}} n_j$ and the number of output element is $n_{out} = \prod_{n_j \in \mathbf{N}} n_j$.

The generic nature of this model allows us to capture the performance features of all circuit-based secure computation protocols. For example, in this work we utilize EMP toolkit [51], a state of the art circuit protocol, in conjunction with Shrinkwrap. As EMP has not been used in private data federations, we implemented EMP with SMCQL. Instead of compiling operators into individual circuits, EMP compiles a SQL query into a single circuit, though it still pads intermediate results within the circuit. In this setting, we can still apply Shrinkwrap as we did in the RAM model, where we minimize the padding according to our differential privacy guarantees. Instead of modifying the ORAM, we directly modify the circuit to eliminate dummy tuples.

For both our RAM model and our circuit model implementations, Shrinkwrap uses a cost model to estimate the execution cost of intermediate result padding. With the model, we can carry out our privacy budget optimization and allocation as seen in Section 5. Shrinkwrap's protocol agnostic design allows it to optimize the performance of secure computation, regardless of the implementation.

6.3 M-Party Support

In this work, we implement Shrinkwrap using two party secure computation protocols, meaning that we run our experiments over two data owners. However, Shrinkwrap supports additional data owners. By leveraging the associativity of our operators, we can convert all m party computations into a series of binary computations. For example, assume we want to join to tables R and S, which are horizontally partitioned across 3 data owners as $R1, R2, R3$, and $S1, S2, S3$. We can join across the 3 data owners as: $(R1 \bowtie S1) \cup (R1 \bowtie S2) \cup (R3 \bowtie S3) \cup (R2 \bowtie S1) \cup (R2 \bowtie S2) \cup (R2 \bowtie S3) \cup (R3 \bowtie S1) \cup (R3 \bowtie S2) \cup (R3 \bowtie S3)$. We

Table 3: HealthLNK query workload.

Name	Query
Dosage Study	SELECT DISTINCT d.pid FROM diagnoses d, medications m WHERE d.pid = m.pid AND medication = 'aspirin' AND icd9 = 'circulatory disorder' AND dosage = '325mg'
Comorbidity	SELECT diag, COUNT(*) cnt FROM diagnoses WHERE pid ∈ cdiff_cohort ∧ diag <> 'cdiff' ORDER BY cnt DESC LIMIT 10;
Aspirin Count	SELECT COUNT(DISTINCT pid) FROM diagnosis d JOIN medication m on d.pid = m.pid JOIN demographics demo on d.pid = demo.pid WHERE d.diag = 'heart disease' AND m.med = 'aspirin' AND d.time <= m.time;
3-Join	SELECT COUNT(DISTINCT pid) FROM diagnosis d JOIN medication m on d.pid = m.pid JOIN demographics demo on d.pid = demo.pid JOIN demographics demo2 ON d.pid = demo2.pid WHERE d.diag = 'heart disease' AND m.med = 'aspirin' AND d.time <= m.time;

can execute this series of binary joins using two party secure computation then union the results. With this construction, we can scale Shrinkwrap out to any number of parties. However, this algorithm is not efficient due to naively carrying out m^2 secure computation operations for every m parties we compute over. We can use m party secure computation protocols [3, 5, 20], which have fewer high-level operators and made large performance strides, but these protocols are still expensive. We leave performance improvements in this setting for future work.

7. RESULTS

We now look at Shrinkwrap performance using both real world and synthetic datasets and workloads. First, we cover our experimental design and setup. Next, we evaluate the end to end performance of our system and show the performance vs privacy trade-off under both true and differentially private answer settings. Then, we look at how Shrinkwrap performs at the operator level and the accuracy of our cost model, followed by a discussion of our budget splitting strategies. Finally, we examine how Shrinkwrap scales to larger datasets and more complex queries.

7.1 Experimental Setup

For this work, we implemented Shrinkwrap on top of an existing healthcare database federation that uses SMCQL [4]. This private data federation serves a group of hospitals that wish to analyze the union of their electronic health record systems for research while keeping individual tuples private.

A clinical data research network (CDRN) is a consortium of healthcare sites that agree to share their data for research. CDRN data providers wish to keep their data private. We examine this work in the context of HealthLNK [44], a CDRN for Chicago-area healthcare sites. This repository contains records from seven 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.

Medical researchers working on HealthLNK develop SQL queries and run them on data from a set of healthcare sites. These researchers are the clients of the private data federation and the healthcare sites are the private data owners.

Dataset: We evaluate Shrinkwrap on medical data from two Chicago area hospitals in the HealthLNK data repository [44] over one year of data. This dataset has 500,000 patient records, or 15 GB of data. For additional evaluation, we generate synthetic data up to 750 GB based on this source medical data. To simplify our experiments, we use a public patient registry for common diseases that maintains a list of anonymized patient identifiers associated with these conditions. We filter our query inputs using this registry.

Query Workload: For our experiments, we chose three representative queries based on clinical data research protocols [23, 43] and evaluate Shrinkwrap on this workload using de-identified medical records from the HealthLNK data repository. We also generate synthetic versions of *Aspirin Count* that contain additional join operators. We refer to these queries by the number of join operators, e.g., 3-Join for 3 join operators. The queries are shown in Table 3.

Configuration: Shrinkwrap query processing runs on top of PostgreSQL 9.5 running on Ubuntu Linux. We evaluated our two-party prototype on 6 servers running in pairs. The servers each have 64 GB of memory, 7200 RPM NL-SAS hard drives, and are on a dedicated 10Gb/s network. Our results report the average runtime of three runs per experiment. We implement Shrinkwrap under both the RAM model (using ObliVM [33]) and circuit model (using EMP[51]). For most experiments, we show results using the RAM model, though corresponding circuit model results are similar. Unless otherwise specified, the results show the end-to-end runtime of a query with output policy 1, i.e., true results.

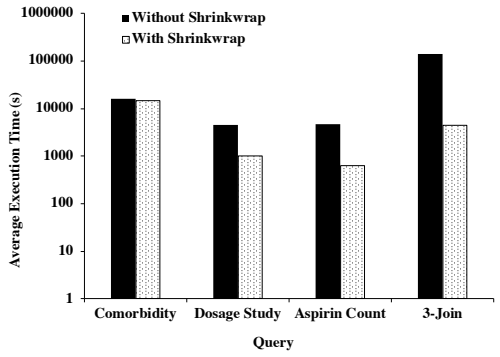
7.2 End-to-end Performance

In our first experiment, we look at the the end to end performance of Shrinkwrap compared to baseline private data federation execution. For execution, we set ϵ to 0.5, δ to 0.00005, and use the optimal budget splitting approach.

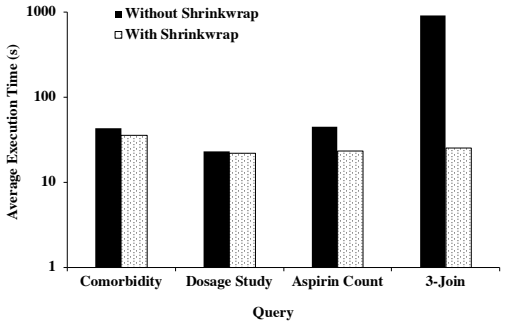
In Figure 5, we look at the overall performance of four queries under Shrinkwrap for both RAM model and circuit model secure computation. For *Comorbidity*, the execution does not contain any join operators, meaning that there is no explosion in intermediate result cardinality, so Shrinkwrap provides fewer benefits. *Dosage Study* contains a join operator with a parent distinct operator. Applying differential privacy to the output of the join improves performance by close to 5x under the RAM model, but sees fewer benefits under the circuit model. *Aspirin Count* contains two joins and sees an order of magnitude improvement under the RAM model and a 2x improvement under the circuit model. Finally, *3-Join* has an enormous cardinality blow-up, allowing Shrinkwrap to improve its performance by 33x under the RAM model and 35x under the circuit model. In Section 7.6, we examine this effect of join operators on performance. Although implementation in the circuit model outperforms the RAM model by an order of magnitude, Shrinkwrap gives similar efficiency improvements in both systems.

7.3 Privacy, Performance, and Accuracy

We consider the performance vs privacy trade-off provided by Shrinkwrap and evaluate the effect of the execution privacy budget on query performance. For this experiment, we use the *3-Join* query with the optimal budget splitting approach, fix the total privacy budget $\epsilon = 1.5$ and $\delta = 0.00005$,

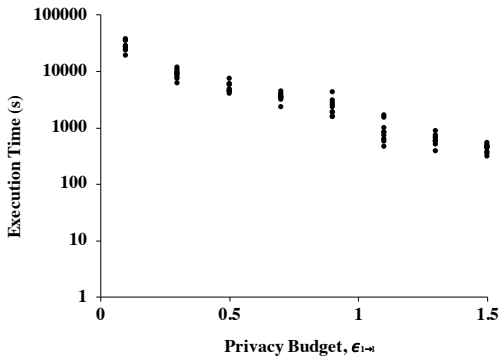


(a) RAM Model (ObliVM),

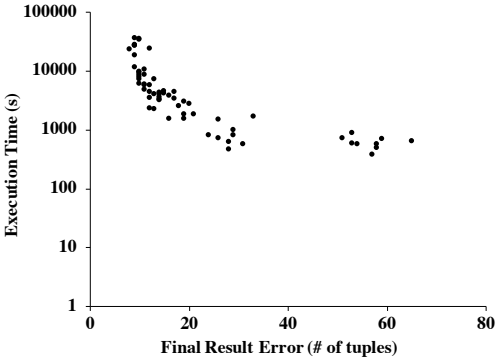


(b) Circuit Model (EMP)

Figure 5: End-to-End Shrinkwrap Performance for All Queries at $\epsilon = 0.5$, $\delta = .00005$.



(a) Performance v.s. $\epsilon_{1\to l}$.



(b) Performance v.s. Error.

Figure 6: Privacy, Performance, and Accuracy Tradeoffs. Computed using different levels of privacy budget for performance $\epsilon_{1\to l} \in \{0.1, ..., 1.5\}$. Executed using RAM Model, *3-Join*, $\epsilon = 1.5, \delta = 0.00005$.

and use output setting 2, i.e. differentially private final answers. We vary the privacy budget for performance δ_{1-l} from 0.1 to 1.5 and fix $\delta_{1\to l} = \delta$. The remaining privacy budget $\epsilon - \epsilon_{1\to l}$ is spent on the query output using a Laplace mechanism.

In Figure 6a, we see the execution time as a function of the privacy budget for performance ($\epsilon_{1\to l}$). As the privacy budget for performance $\epsilon_{1\to l}$ increases, the execution time decreases. A lower budget means that more noise must be present in the intermediate result cardinalities, which translates to larger cardinalities and higher execution time variance. We know that I/O cost dominates the Shrinkwrap cost model, so larger cardinalities mean more I/O accesses, which causes lower performance.

In Figure 6b, we show the accuracy versus privacy tradeoff by varying the privacy budget for performance $\epsilon_{1\to l}$. We report the execution time and the error introduced to the final query answer at various privacy budget for performance $\epsilon_{1\to l} < 1.5$. Out of a total possible output size of 5500 tuples, the query output is $10/5500 = 0.18\%$ of the total possible output size. If we allow additive noise of 0.36% (20/5500), we get a 100x performance improvement. Since the total privacy ϵ is fixed, the more we spend for performance (larger $\epsilon_{1\to l}$), the less privacy budget we have at the output, i.e., as our execution time improves, our output noise increases.

7.4 Evaluating Budget Splitting Strategies

We first examine the relative performance of the three budget splitting strategies introduced in Section 6: uniform, eager, and optimal. Recall that uniform splits the budget evenly across all the operators, eager inserts the entire budget at the first operator, and uses the Shrinkwrap cost model, along with cardinality estimates, to identify an optimal budget split. We also include an *oracle* approach that shows the performance of Shrinkwrap if true cardinalities are used in the cost model to split the budget instead of cardinality estimates. Note that oracle does not satisfy differential privacy, but gives an upper bound on the best performance achievable through privacy budgeting. For this experiment, we use the *Aspirin Count* and *3-Join* queries since they contain multiple operators where Shrinkwrap generates differentially-private cardinalities and set the privacy parameters usable during execution to $\epsilon = 0.5$ and $\delta = .00005$.

Figure 7 displays the relative speedup of all three approaches and the oracle for both queries over the baseline, fully padded, private data federation execution. All three Shrinkwrap approaches provide significant performance improvements, ranging from 3x to 33x. As expected, optimal performs best for both queries. We also see that eager performs better for *Aspirin Count*, while uniform performs better for *3-Join*.

The benefits of minimized intermediate result cardinalities cascades as those results flow through the operator tree. By using the entire budget on the first join operator in *Aspirin Count*, the eager approach maximizes the effect of the intermediate result cardinality reduction, and as a result, outperforms the uniform approach. However, for the *3-Join*

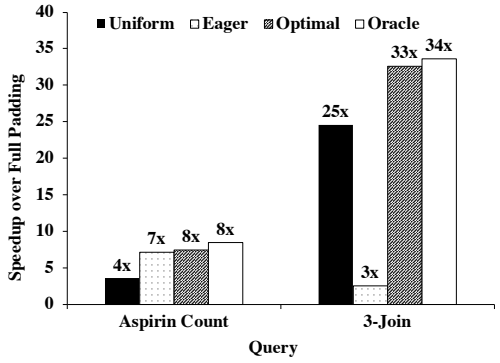


Figure 7: Shrinkwrap execution speedup over baseline using different budget strategies. Executed using RAM model, $\epsilon = 0.5$, $\delta = .00005$

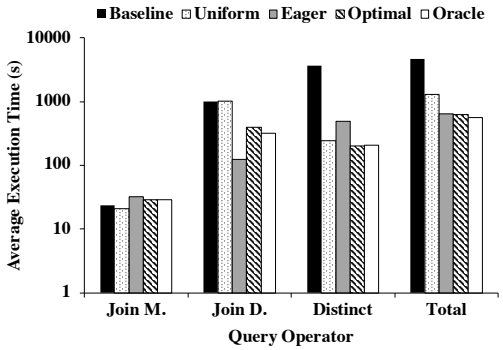


Figure 8: Per operator performance using different budget strategies. Executed using *Aspirin Count*, RAM model, $\epsilon = 0.5$, $\delta = .00005$

query, the presence of the additional join operator overrides the cascading cardinality benefit of the first join operator. The uniform approach outperforms the eager approach by ensuring that all three of the join operators receive differentially private cardinalities. The optimal approach outperforms both eager and uniform by combining the best of both worlds. Optimal applies differential privacy to all operators, like uniform, but it uses a larger fraction of the budget on earlier operators, like eager.

Looking at the oracle approach, we can evaluate the accuracy of our cost model. For privacy reasons, our cost model does not use the true cardinalities. Here, we see that the optimal approach, which uses estimated cardinalities, and the oracle approach, which uses true cardinalities, provide similar performance. In experiments using *Aspirin Count*, we calculate the correlation coefficient between the true execution time and the estimated execution time based on our cost model as .998 for the circuit model and .931 for the RAM model, given a $\epsilon = 0.5$, $\delta = .00005$. We see that the Shrinkwrap cost model provides a reasonably accurate prediction of the true cost.

7.5 Operator Breakdown

Now, we look at the execution time by operator to see where Shrinkwrap provides the largest impact. We include only private operators in the figure. For this experiment we use the *Aspirin Count* query with $\epsilon = 0.5$ and $\delta = 0.00005$. We show the execution times for the baseline, fully padded approach for intermediate results, as well as the uniform, eager, and optimal budget splitting approaches.

Figure 8 shows the execution times of each operator in the *Aspirin Count* query. Note that Shrinkwrap does not apply any differential privacy until the output of Join M. (Join on Medications), so we see no speed-up in the actual Join M. operator, only in the later query operators.

The first operator to see a benefit from Shrinkwrap is Join D. (Join on Demographics), the second join in the query. Shrinkwrap generates a differentially private cardinality for the output of the previous join, Join M., according to the selected budget splitting approach. For uniform, the allocated budget is not large enough to generate a differentially-private cardinality that is smaller than the fully padded cardinality. In fact, the overhead of calculating the differentially-private cardinality actually causes the operator runtime to go above the baseline runtime. For eager, Shrinkwrap spends the entire budget on the output of the first join and reduces the intermediate cardinality by an order of magnitude. The optimal approach uses enough budget to reduce the intermediate cardinality, but not as much as eager.

The next operator, DISTINCT, also sees a large performance benefit. All three budget splitting approaches reduce the runtime for DISTINCT by about an order of magnitude. Since the baseline intermediate cardinality for DISTINCT is much larger, the uniform approach generates a differentially-private cardinality that reduces the intermediate result size and improves performance. The eager approach does not have any additional budget to use, but the effect of the previous intermediate cardinality cascades through the query tree and reduces the execution time for DISTINCT as well. Finally, the optimal approach applies its remaining budget to see a significant performance improvement.

In the *Aspirin Count* query, the three budget splitting approaches allocate the budget between three operators and provide an insight into the trade-off between early and late budget allocation. We see wildly varying execution times for each of the operators, with different operators providing the bulk of the performance cost depending on the budget strategy. Here, the optimal strategy gives the best performance. The substantial variance in execution for these operators demonstrates the value of the added accuracy that our I/O cost model provides.

7.6 Join Scale Up

We now look at how Shrinkwrap scales with more complex workloads. From our experiments, we know that the number of joins in a query has an extremely large impact on execution time. More complex workloads typically contain more nested statements and require more advanced SQL processing, but can be broken down into a series of simpler statements. This work focuses on complexity as a function of the join count in order to target the largest performance bottleneck in a single SQL statement.

In our experiment, we scale the join count and measure the performance. Since Shrinkwrap applies differential privacy at the operator outputs, we measure the performance impact of join operators by looking at the execution time of their immediate parent operator. For consistency, we truncate the number of non-inherited input tuples of each join to equal magnitudes and use $\epsilon = 0.5$ and $\delta = 0.00005$ for all runs.

From Figure 9, we see how execution time scales as a function of the join count. As the number of joins increases,

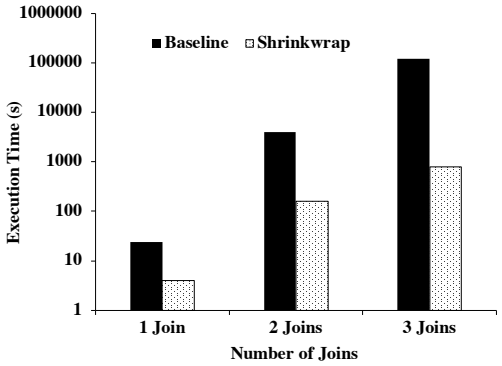


Figure 9: *Aspirin Count* with synthetic join scaling. Executed using RAM model. $\epsilon = 0.5$, $\delta = .00005$

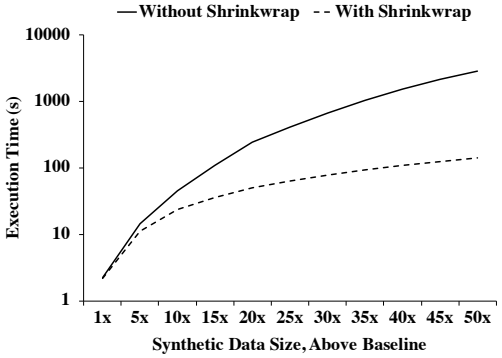


Figure 10: *Aspirin Count* with synthetic data scaling. Executed using Circuit model. $\epsilon = 0.5$, $\delta = .00005$

the execution time also increases. To find the source of the performance slowdown, we can look at the Shrinkwrap cost model. For each join operator, the baseline private data federation fully pads the output. If the join inputs were both of size n, then the output would be n^2. As the number of joins increase, the cardinality rises from n to n^2 to n^3. The exponential increase in output cardinality requires significantly more I/O accesses, which cause the large slow down in performance. Shrinkwrap applies differential privacy to reduce the output size at each join, reducing the magnitude of each join output. While Shrinkwrap still sees a significant growth in execution time as a function of the join count, the overall performance is orders of magnitudes better.

7.7 Data Size Scale Up

We evaluate the performance of Shrinkwrap with increasing data sizes. We leverage the higher efficiency of our circuit model protocol, EMP [51], to examine the effect of Shrinkwrap on larger input data sizes. Without Shrinkwrap, the RAM model protocol, ObliVM [33], needs 1.3 hours to run the *Aspirin Count* query and at 50x, 65 hours to complete. Instead, we use EMP which completes in 15 minutes.

Figure 10 shows the execution time with and without Shrinkwrap using an implementation of EMP [51] on our *Aspirin Count* reference query. We generated synthetic data based on the original tables, giving us an effective maximum data size of 750 GB. In Figure 10, we see that Shrinkwrap provides a significant improvement during execution. As the data size grows, our performance improvement grows as well, reflecting the power and flexibility of Shrinkwrap.

8. RELATED WORK

Within the literature, different approaches exist to improve query processing in private data federations, such as databases based on homomorphic encryption, TEEs, differential privacy, and cloud computation [1, 29, 30, 36, 45, 54]. Shrinkwrap, on the other hand, provides general-purpose, hardware-agnostic, *in-situ* SQL evaluation with provable privacy guarantees and exact results.

PINQ [36] introduced the first database with differential privacy, along with privacy budget tracking and sensitivity analysis for operator composition. We extend this work by applying its privacy calculus to private data federations. Follow on work in differential privacy appears in DJoin [38], where the system supports private execution of certain SQL operators over a federated database with strong privacy guarantees and noisy results. Shrinkwrap supports a larger set of database operators for execution and instead of using noisy results to safeguard data, Shrinkwrap uses noisy cardinalities to improve performance.

He et al. [22] applied computational differential privacy in join operators for private record linkage and proposed a three desiderata approach to operator execution: precision, provability, and performance. Shrinkwrap incorporates this style of join execution and approach to execution trade-offs.

Pappas et al. [42] showed that by trading small bits of privacy for performance within provable bounds using bloom filters, systems can provide scalable DBMS queries over arbitrary boolean queries. Shrinkwrap applies this pattern of provable privacy versus performance trade-offs to the larger set of non-boolean arbitrary SQL queries.

Both Opaque [56] and Hermetic [54] use cost models to estimate performance slowdowns due to privacy-preserving computation. In both cases, they use secure enclaves to carry out private computation, which provides constant-cost I/O. As such, their cost models cannot account for the variable-cost I/O present in Shrinkwrap.

9. CONCLUSIONS AND FUTURE WORK

In this work, we introduce Shrinkwrap, a protocol-agnostic system for differentially-private query processing that significantly improves the performance of private data federations. We use a computational differential privacy mechanism to selectively leak statistical information within provable privacy bounds and reduce the size of intermediate results. We introduce a novel cost model and privacy budget optimizer to maximize the privacy, performance, and accuracy trade-off. We integrate Shrinkwrap into existing private data federation architecture and collect results using real-world medical data.

In future research, we hope to further improve the performance of private data federations and the efficiency of Shrinkwrap. We are currently investigating novel secure array algorithms and data structures to improve I/O access time, privacy budget optimizations over multiple queries, and extensions of private data federations using additional secure computation protocols and cryptographic primitives.

Acknowledgments: We thank Abel Kho, Satyender Goel, Katie Jackson, and Jess Joseph Behrens for their guidance and assistance with CAPriCORN and HealthLNK data. We also thank the HealthLNK team for sharing de-identified electronic health record data for this study.

10. REFERENCES

[1] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal Security with Cipherbase. *CIDR*, 2013.

[2] A. Arasu and R. Kaushik. Oblivious Query Processing. *ICDT*, pages 26–37, 2014.

[3] G. Asharov and Y. Lindell. The BGW protocol for perfectly-secure multiparty computation. *Cryptology and Information Security Series*, 10(189):120–167, 2013.

[4] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure Querying for Federated Databases. *PVLDB*, 10(6):673–684, 2017.

[5] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols. *ACM Symposium on Theory of Computing STOC*, pages 503–513, 1990.

[6] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*, pages 478–492. IEEE, 2013.

[7] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov. The tao of inference in privacy-protected databases. *PVLDB*, 11(11):1715–1728, 2018.

[8] D. Bogdanov, L. Kamm, S. Laur, P. Pruulmann-Vengerfeldt, R. Talviste, and J. Willemson. Privacy-preserving statistical data analysis on federated databases. In *Annual Privacy Forum*, pages 30–55. Springer, 2014.

[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. 2018.

[10] S. S. Chow, J.-H. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*, 2009.

[11] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and and M. Zhu. Tools for Privacy Preserving Distributed Data Mining. *ACM SIGKDD Explorations*, 4(2), 2002.

[12] J. Doerner and A. Shelat. Scaling ORAM for Secure Computation. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, pages 523–535, 2017.

[13] C. Dwork. Differential privacy. *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*, pages 1–12, 2006.

[14] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 2014.

[15] K. E. Emam. Heuristics for de-identifying health data. *IEEE Security & Privacy*, 6(4):58–61, 2008.

[16] Ú. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1054–1067. ACM, 2014.

[17] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. *ACM SIGARCH Computer Architecture News*, 45(1):555–568, 2017.

[18] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, New York, NY, USA, 1987. ACM.

[19] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[20] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game. *Stoc '87*, pages 218–229, 1987.

[21] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[22] X. He, A. Machanavajjhala, C. Flynn, and D. Srivastava. Composing Differential Privacy and Secure Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, number 1, pages 1389–1406, New York, New York, USA, 2017.

[23] 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.

[24] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security 2011*, 2011.

[25] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, LNCS, pages 145–161, 2003.

[26] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS*, 2012.

[27] N. Johnson, J. P. Near, and D. Song. Towards Practical Differential Privacy for SQL Queries. *PVLDB*, 11(5):526–539, 2018.

[28] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. 2018.

[29] S. Laur, R. Talviste, and J. Willemson. From oblivious AES to efficient and secure database join in the multiparty setting. *Lecture Notes in Computer Science*, 7954 LNCS:84–101, 2013.

[30] S. Laur, J. Willemson, and B. Zhang. Round-efficient Oblivious Database Manipulation. *ISC'11*, pages 262–277, 2011.

[31] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon : Cache Attacks on Mobile Devices. *USENIX Security*, pages 549–564, 2016.

[32] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating Efficient RAM-Model Secure Computation. *2014 IEEE Symposium on Security and Privacy*, pages 623–638, 2014.

[33] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM : A Programming Framework for Secure Computation. *Oakland*, pages 359–376, 2015.

[34] A. Machanavajjhala, D. Kifer, J. M. Abowd, J. Gehrke, and L. Vilhuber. Privacy: Theory meets practice on the map. In *ICDE*, 2008.

[35] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *PVLDB*, 11(10):1206–1219, 2018.

[36] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *2009 ACM SIGMOD*, pages 19–30. ACM, 2009.

[37] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *CRYPTO*, 2009.

[38] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. *Proceedings of the 10th USENIX Symposium*, page 14, 2012.

[39] 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.

[40] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC : Parallel Secure Computation Made Easy.

[41] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 801–812. ACM, 2013.

[42] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private DBMS. *Proceedings - IEEE Symposium on Security and Privacy*, pages 359–374, 2014.

[43] PCORI. Characterizing the Effects of Recurrent Clostridium Difficile Infection on Patients. *IRB Protocol*, ORA: 14122, 2015.

[44] PCORI. Exchanging de-identified data between hospitals for city-wide health analysis in the Chicago Area HealthLNK data repository (HDR). *IRB Protocol*, 2015.

[45] R. Popa and C. Redfield. CryptDB: protecting confidentiality with encrypted query processing. *SOSP*, pages 85–100, 2011.

[46] J. Saia and M. Zamani. Recent Results in Scalable Multi-Party Computation. 2014.

[47] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[48] S. Vadhan. *The Complexity of Differential Privacy*. Springer International Publishing, 2017.

[49] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, R. Strackx, and K. Leuven. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, 2018.

[50] N. Volgushev, M. Schwarzkopf, B. Getchell, A. Lapets, M. Varia, and A. Bestavros. Conclave Workflow Manager for MPC, 2018.

[51] X. Wang, A. J. Malozemoff, and J. Katz. EMP-Toolkit: Efficient Multiparty Computation Toolkit. https://github.com/emp-toolkit, 2016.

[52] X. Wang, A. J. Malozemoff, and J. Katz. Faster secure two-party computation in the single-execution setting. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10212 LNCS:399–424, 2017.

[53] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious Data Structures. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pages 215–226, 2014.

[54] M. Xu, A. Papadimitriou, A. Feldman, and A. Haeberlen. Using Differential Privacy to Efficiently Mitigate Side Channels in Distributed Analytics. In *Proceedings of the 11th European Workshop on Systems Security - EuroSec'18*, pages 1–6, New York, New York, USA, 2018. ACM Press.

[55] A. C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[56] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 283–298, Berkeley, CA, USA, 2017. USENIX Association.