PRISM: Privacy-Preserving and Verifiable Set Computation Over Multi-Owner Secret Shared Outsourced Databases

Shantanu Sharma, Yin Li, Sharad Mehrotra, *Fellow, IEEE*, Nisha Panwar, Peeyush Gupta, and Dhrubajyoti Ghosh

***Abstract*—Private set computation over multi-owner databases is an important problem with many applications — the most well studied of which is private set intersection (PSI). This article proposes PRISM, a secret-sharing based approach to compute private set operations (i.e., intersection and union, as well as aggregates such as count, sum, average, maximum, minimum, and median) over outsourced databases belonging to multiple owners. PRISM enables data owners to pre-load the data onto non-colluding servers and exploits the additive and multiplicative properties of secret-shares to compute the above-listed operations. PRISM takes (at most) two rounds of communication between non-colluding servers (storing the secret-shares) and the querier for executing the above-mentioned operations, resulting in a very efficient implementation. PRISM also supports result verification techniques for each operation to detect malicious adversaries. Experimental results show that PRISM scales both in terms of the number of data owners and database sizes, to which prior approaches do not scale.**

***Index Terms*—Additive sharing, aggregation operation, computation and data privacy, data and computation outsourcing, multi-party computation, private set intersection, private set union, result verification, set cardinality, shamir's secret-sharing.**

I. Introduction

DATABASE-AS-A-SERVICE (DaS) [34] has become increasingly popular with the emergence of cloud computing. Initially, DaS focused on a single database (DB) owner who submitted encrypted data to the cloud, allowing either the owner or their clients to query it. However, a more common scenario involves multiple datasets, each belonging to a different owner. These data owners do not trust one another but need to execute queries on common attributes of the datasets. The query execution must not reveal the content of the database belonging to one DB owner to others, except for the leakage that may occur from the answer to the query. The most common form of such queries is the *private set intersection* (PSI) [27], [37], [42], [46], [59], [60], [62]. An example use-case of PSI include *syndromic surveillance*, wherein organizations, such as pharmacies and/or hospitals, share information (*e.g.*, a sudden increase in sales of specific drugs such as analgesics or anti-allergy medicine, tele-health calls, and school absenteeism requests) to enable early detection of community-wide outbreaks of diseases. PSI is also a building block for performing joins across private databases — it essentially corresponds to performing a semi-join operation on the join attribute [43].

Private set computations over datasets owned by different DB owners/organizations can, in general, be implemented using secure multiparty computation (SMC) [31], [52], [71], a well-known cryptographic technique that has been prevalent since more than three decades. SMC allows DB owners to securely execute any function over their datasets without revealing their data to other DB owners. However, SMC can be very slow, often by order of magnitude [48]. As a result, techniques that can be used to more efficiently compute private set operations have been developed; particularly, in the context of PSI [27], [37], [42], [59], [62] and *private set union* (PSU) [21], [45]. PSU refers to privately computing the union of all databases. Several approaches using homomorphic encryption [14], polynomial evaluation [27], garbled-circuit techniques [37], hashing [26], [59], [67], hashing and oblivious pseudorandom functions (OPRF) [47], Bloom-filter [55], and oblivious transfer [58], [61] have been proposed to implement private set operations.

Recent work on private set operations has also explored performing aggregation on the result of PSI operations. For instance, [39] studied the problem of private set intersection sum (PSI Sum), motivated by the internet advertising use-case, where one party has information about customers who clicked on specific advertisements during their web session, and another party has a list of transactions about items listed in the advertisements that resulted in a purchase by the customers. Both parties may wish to securely learn the total sales attributed to customers

Manuscript received 20 October 2021; revised 19 April 2023; accepted 9 May 2023. Date of publication 24 May 2023; date of current version 16 May 2024. The work of Shantanu Sharma was supported by NSF under Grant 2245374. The work of Yin Li was supported in part by National Natural Science Foundation of China under Grant 62372107 and in part by National Key Research and Development Program of China under Grant 2021YFB3101303. The work of Nisha Panwar was supported in part by NSF under Grant 2131538. The work of Sharad Mehrotra was supported by NSF under Grant 2420846, Grant 2245372, Grant 2133391, Grant 2008993, and Grant 1952247. *(Corresponding author: Shantanu Sharma.)*

Shantanu Sharma is with the New Jersey Institute of Technology, Newark, NJ 07102 USA (e-mail: shantanu.sharma@njit.edu).

Yin Li is with the Dongguan University of Technology, Dongguan 523820, China (e-mail: yunfeiyangli@gmail.com).

Nisha Panwar is with Augusta University, Augusta, GA 30912 USA (e-mail: npanwar@augusta.edu).

Sharad Mehrotra, Peeyush Gupta, and Dhrubajyoti Ghosh are with the University of California, Irvine, CA 92697 USA (e-mail: sharad@ics.uci.edu; peeyushg@uci.edu; dhrubajg@uci.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TDSC.2023.3279356, provided by the authors.

Digital Object Identifier 10.1109/TDSC.2023.3279356

clicking on the advertisements, without revealing their databases to each other due to fair/competitive business strategies.

Existing approaches on private set computation (including recent work on aggregation) are limited in several ways:

- Work on PSI or PSU has largely focused on the case of two DB owners, with some exceptions that address more than two DB owners scenarios, e.g., [15], [27], [36], [38], [45], [48], [72]. There are several interesting use-cases, where one may wish to compute PSI over multiple datasets. For instance, in the syndromic surveillance example listed above, one may wish to compute intersection amongst several independently owned databases. Generalizing existing two-party PSI or PSU approaches to the case of multiple DB owners results in significant overhead [48]. For instance, [2], which is designed for two DB owners, incurs $(nm)^2$ communication cost, when extended to $m > 2$ DB owners, where n is the dataset size. Even recent work supporting multiple DB owners incurred significant computational overhead; e.g., [38] took ≈12 seconds for PSI over 24 DB owners having 1024 values.
- Techniques to privately compute aggregation over set operations have not been studied systematically. In the database literature, aggregation functions [54] are typically classified as: *summary* aggregations (such as count, sum, and average) or *exemplary* aggregations (such as minimum, maximum, and median). Existing literature has only considered the problem of PSI Sum [39] and cardinality determination, i.e., the size of the intersection or union [21], [24]. Techniques for exemplary aggregations (and even for summary aggregations) that may compute over multiple attributes have not been explored.
- Many of the existing solutions do not deal with a large amount of data, due to either inefficient cryptographic techniques or multiple communication rounds amongst DB owners. For instance, recent work [48], [49], [72] dealt with data that is limited to sets of size less than or equal to ≈1 M in size.

This paper introduces PRISM — a novel approach for computing collaboratively over multiple databases. PRISM is designed for both PSI and PSU, and it supports both summary, as well as, exemplar aggregations. Unlike existing SMC techniques (wherein DB owners compute operations privately through a sequence of communication rounds), in PRISM, DB owners outsource their data in secret-shared form to multiple ***non-colluding public servers***. As will become clear, PRISM exploits the homomorphic nature of secret-shares (both additive and multiplicative) to enable servers to compute private set operations independently (to a large degree). These results are then returned to DB owners to compute the final results. In PRISM, any operator requires at most two communication rounds between DB owners and servers, where the first round finds tuples that are in the intersection or union of the set, and the second round computes the aggregation function over the objects in the intersection/union.

By using public servers for computation over secret-shared data, PRISM achieves the identical security guarantees as existing SMC systems (e.g., Sharemind [7], Jana [4], and Conclave [68]). The key advantage of PRISM is that by outsourcing data in secret shared form and exploiting homomorphic properties, PRISM does not require communication among server before/during/after the computation, which allows PRISM to perform efficiently even for large data sizes and for a large number of DB owners (as we will show in experiment section). Since PRISM uses the public servers, which may act maliciously, PRISM supports oblivious result verification methods.

In summary, PRISM offers the following benefits: (*i*) *Information-theoretical security:* It achieves information-theoretical security at the servers and prevents them to learn anything from input/output/access-patterns/output-size. (*ii*) *No communication among servers:* It does not require any communication among servers, unlike SMC based solutions. (*iii*) *No trusted entity:* It does not require any trusted entity that performs the computation on the cleartext data, unlike the recent SMC system Conclave [68]. (*iv*) *Several DB owners and large-sized dataset:* It deals with several DB owners having a large-size dataset.

Outline: Different sections of this papers are organized as follows:

1) Section II provides the definitions and examples of PSI, PSU, and aggregation operations over PSI/PSU.
2) Section III-A provides an overview of existing techniques (such Shamir's secret-sharing, additive sharing, cyclic group, permutation function, and pseudorandom number generator) that we will use in developing our algorithms. Section III-B provides details about entities involved in PRISM. Section III-C provides an overview of PRISM system. Section III-D provides security properties.
3) Section IV provides details of assumptions and parameters related to different entities.
4) Section V provides PSI algorithm and PSI output verification algorithm, respectively.
5) Sections VI-A and VI-A2 provide PSI sum algorithm and PSI sum output verification algorithm, respectively. Section VI-B extends PSI sum algorithm for computing PSI average algorithm.
6) Section VI-C provides PSI maximum algorithms. Section VI-D extends PSI maximum algorithm for computing PSI median algorithm.
7) Section VI-E provides PSI count and verification algorithms.
8) Section VI-F extends PSI algorithm of Section V for computing PSI over multiple columns and proposes *bucketization-based PSI*.
9) Section VII provides PSU algorithm. *Note:* Aggregations queries over PSU can be executed in a similar way and by following the methods of aggregation over PSI (Section VI). Also, using bucketization-based method, PSU method can be executing over multiple columns. Thus, we do not explicitly provide aggregation algorithm over PSU and bucketization-based PSU.
10) Section VIII-A provides experimental evaluation of PRISM. Section VIII-B compares PRISM against existing PSI/PSU techniques.

II. PRIVATE SET OPERATIONS

We, first, define the set of operations supported by PRISM. Let $DB_1, \ldots, DB_m$ be $m > 2$ independent DBs owned by m DB owners $\mathcal{DB}_1, \ldots, \mathcal{DB}_m$. We assume that each DB owner is

TABLE I
HOSPITAL 1

	Name	Age	Disease	Cost
τ_1	John	4	Cancer	100
τ_2	Adam	6	Cancer	200
τ_3	Mike	2	Heart	300

TABLE II
HOSPITAL 2

	Name	Age	Disease	Cost
ν_1	John	8	Cancer	100
ν_2	Adam	5	Fever	70
ν_3	Bob	4	Fever	50

Note: τ_i, ν_i, and ρ_i denote the i^{th} tuples of tables.

TABLE III
HOSPITAL 3

	Name	Age	Disease	Cost
ρ_1	Carl	8	Cancer	300
ρ_2	John	4	Cancer	700
ρ_3	Lisa	5	Heart	500

(partially) aware of the schema of data stored at other DB owners. Particularly, DB owners have knowledge of the attribute(s) of the data stored at other DB owners on which the set-based operations (i.e., intersection or union) can be performed. Also, DB owners know about the attributes on which aggregation functions (e.g., sum, min, max) be supported. Such an assumption is needed to ensure that PSI/PSU and aggregation queries are well defined. Other than the above requirement, the schema of data at different databases may be different.

Now, we define the private set operations supported by PRISM formally and their corresponding privacy requirements (corresponding SQL statements are shown in Table IV). In defining the operators (and in the rest of the paper), we will use the example tables shown in Tables I, II, and III that are owned by three different DB owners (in our case, hospitals).

1) *Private Set Intersection (PSI) (Section V):* PSI finds the common values among m DB owners for a specific attribute A_c, i.e., $DB_1.A_c \cap \ldots \cap DB_m.A_c$. For example, PSI over disease column of Tables I, II, and III will return {Cancer} as a common disease treated by all hospitals. Note that *a hospital computing PSI on disease should not gain any information about other possible disease values (except for the result of the PSI) associated with other hospitals*.
2) *Private Set Union (PSU) (Section VII):* PSU finds the union of values among m DB owners for a specific attribute A_c, i.e., $DB_1.A_c \cup \ldots \cup DB_m.A_c$. For example, PSU over disease column returns {Cancer, Fever, Heart} as diseases treated by all hospitals. Again, *a hospital computing PSU over other hospitals must not gain information about the specific diseases treated by other hospitals, or how many hospitals treat which disease*.
3) *Aggregation over private set operators (Section VI):* Aggregation ${}_{A_c}\mathcal{G}_\theta(A_x)$ computes the aggregation function θ on the attribute A_x ($A_c \neq A_x$) for the groups corresponding to the output of set-based operations (PSI or PSU) on attribute A_c. For example, the aggregation function sum on cost attribute corresponding to PSI over disease attribute (i.e., ${}_{\text{disease}}\mathcal{G}_{sum}(\text{cost})$) returns a tuple $\{\text{Cancer,1400}\}$. The same aggregation function over PSU will return $\{\langle\text{Cancer,1400}\rangle, \langle\text{Fever,120}\rangle, \langle\text{Heart,800}\rangle\}$. Likewise, the output of aggregation ${}_{\text{disease}}\mathcal{G}_{max}(\text{age})$ over PSI would return $\{\text{Cancer,8}\}$, while the same over PSU would return $\{\langle\text{Cancer,8}\rangle, \langle\text{Fever,5}\rangle, \langle\text{Heart,5}\rangle\}$. Note that the count operation does not require specifying an aggregation attribute A_x and can be computed over the attribute(s) associated with PSI or PSU. For example, count over PSI (PSU) on disease column will return 1 (3) respectively. From the perspective of privacy requirement, in case of PSI on disease column, a hospital executing an aggregation query (maximum of age or sum of cost) should only gain information about the answer, i.e., *elements in the PSI and the corresponding aggregate value*. It should not gain information about other diseases that are not in the intersection. Likewise, for PSU, the hospital will gain information about *all elements in the union and their corresponding aggregate values, but will not gain any specific information about which database contains which disease values, or the number of databases with a specific disease*.

III. PRELIMINARY

This section describes the cryptographic concepts that serve as building blocks for PRISM, provides an overview of PRISM approach, and discusses its security properties.

A. Building Blocks

PRISM is based on additive secret-sharing (SS), Shamir's secret-sharing (SSS), cyclic group, and pseudorandom number generator. We provide an overview of these techniques, below.

Additive Secret-Sharing (SS): Additive SS is the simplest type of the SS. Let δ be a prime number. Let $\mathbb{G}_\delta$ be an Abelian group under modulo addition δ operation. All additive shares are defined over $\mathbb{G}_\delta$. In particular, the DB owner creates c shares $A(s)^1, A(s)^2, \ldots, A(s)^c$ over $\mathbb{G}_\delta$ of a secret, say s, such that $s = A(s)^1 + A(s)^2 + \ldots + A(s)^c$. The DB owner sends share $A(s)^i$ to the i^{th} server (belonging to a set of c non-communicating servers). These servers cannot know the secret s until they collect all c shares. To reconstruct s, the DB owner collects all the shares and adds them. Additive SS allows *additive homomorphism*. Thus, servers holding shares of different secrets can locally compute the sum of those shares. Let $A(x)^i$ and $A(y)^i$ be additive shares of two secrets x and y, respectively, at a server i, then the server i can compute $A(x)^i + A(y)^i$ that enable DB owner to know the result of $x + y$. The precondition of *additive homomorphism is that the sum of shares should be less than δ*.

Example: Let $\mathbb{G}_5 = \{0, 1, 2, 3, 4\}$ be an Abelian group under the addition modulo 5. Let 4 be a secret. The DB owner may create two shares, such as 3 and 1 (since $4 = (3 + 1) \bmod 5$), and sends them to two servers.

TABLE IV
SQL SYNTAX OF OPERATIONS SUPPORTED BY PRISM

PSI	SELECT A_c FROM db_1 INTERSECT ... INTERSECT SELECT A_c FROM db_m
PSU	SELECT A_c FROM db_1 UNION ... UNION SELECT A_c FROM db_m
PSI count	SELECT COUNT(A_c) FROM db_1 INTERSECT ... INTERSECT SELECT A_c FROM db_m

PSI θ
 $\theta \in$ (AVG, SUM, MAX, MIN, Median)
	CREATE VIEW $CommonA_c$ as SELECT A_c FROM db_1 INTERSECT ... INTERSECT SELECT A_c FROM db_m
	 SELECT A_c, θ(A_x) FROM (SELECT A_x, A_c FROM db_1, $CommonA_c$ WHERE $db_1.A_c$ = $CommonA_c.A_c$ UNION ALL ...
	 UNION ALL SELECT A_x, A_c FROM db_m, $CommonA_c$ WHERE $db_m.A_c$ = $CommonA_c.A_c$) as inner_relation Group By A_c

Shamir's Secret-Sharing (SSS) [65]: The DB owner randomly selects a polynomial of degree c' with c' random coefficients, i.e., $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{c'}x^{c'}$, where $f(x) \in \mathbb{F}_p[x]$, p is a prime number, $\mathbb{F}_p$ is a finite field of order p, $a_0 = s$, and $a_i \in \mathbb{N}$ $(1 \leq i \leq c')$. The DB owner distributes the secret s into c shares by computing $f(x)$ $(x = 1, 2, \ldots, c)$ and sends an i^{th} share to the i^{th} server (belonging to a set of c non-communicating servers). The secret can be reconstructed using any $c' + 1$ shares using Lagrange interpolation [18].

SSS, also, allows *additive homomorphism*, i.e., if $S(x)^i$ and $S(y)^i$ are SSS of two secrets x and y, respectively, at a server i, then the server i can compute $S(x)^i + S(y)^i$, which will result in $x + y$ at DB owner.

Cyclic Group Under Modulo Multiplication: Let η be a prime number. A group $\mathbb{G}$ is called a cyclic group, if there exists an element $g \in \mathbb{G}$, such that all $x \in \mathbb{G}$ can be derived as $x = (g^i)$ (where i in an integer number $\mathbb{Z}$) under modulo multiplicative η operation. The element g is called a generator of the cyclic group, and the number of elements in $\mathbb{G}$ is called the *order* of $\mathbb{G}$. Based on each element x of a cyclic group, we can form a cyclic subgroup by executing $x^i \bmod \eta$.

Example: $g = 2$ is a generator of a cyclic group under multiplication modulo $\eta = 11$ for the following group: $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Note that the group elements are derived by $2^i \bmod 11$. By taking the element 5 of this cyclic group, we form the following cyclic subgroup $\{1, 3, 4, 5, 9\}$, under multiplication modulo $\eta = 11$, by $5^i \bmod 11$.

Permutation Function $\mathcal{PF}$: Let A be a set. A permutation function $\mathcal{PF}$ is a bijective function that maps a permutation of A to another permutation of A, i.e., $\mathcal{PF} : A \rightarrow A$.

Pseudorandom Number Generator $\mathcal{PRG}$: A pseudorandom number generator is a deterministic and efficient algorithm that generates a pseudorandom number sequence based on an input seed [6], [30].

B. Entities and Trust Assumption

PRISM assumes the following four entities:

1) The m **database (DB) owners** (or users), who wish to execute computation on their joint datasets. We assume that each DB owner is trusted and does not act maliciously.
2) A set of $c \geq 2$ ***servers*** that store the secret-shared data outsourced by DB owners and execute the requested computation from authenticated DB owners. The data transmission between a DB owner and a server takes place in encrypted form or by using anonymous routing [32] to prevent the locations of servers and the shares from an adversary, eavesdropping on the communication channel between DB owners and servers.

We assume that servers do not maliciously communicate (i.e., non-colluding servers) with each other in violation of PRISM protocols. Unlike other MPC mechanisms [7], (as will be clear soon), PRISM protocols do not require the servers to communicate before/during/after the execution of the query. The security of secret-sharing techniques requires that out of the c servers, no more than $c' < c$ communicate maliciously or collude with each other, where c' is a minority of servers (i.e., less than half of c). Thus, we assume that a majority of servers do not collude and communicate with each other, and hence, a legal secret value cannot be generated/inserted/updated/deleted at the majority of the servers.

It should be noted that the assumption of the collusion of servers in violation of the protocol is a common requirement for secret-sharing based protocols, and has been made in prior work [7], [16], [65], [70]. This assumption is based on factors such as economic incentives (as violating the protocol goes against their financial interests), legal restrictions (as collusion may be illegal), and jurisdictional boundaries. To further increase the realism of the assumption, servers can be selected from different cloud providers. For the purpose of simplicity, we assume that none of the servers collude with each other – that is they not communicate directly. Thus, to reconstruct the original secret value from the shares, *two additive shares* suffice. In the case of PSI sum (as will be clear in Section VI-A), we need to multiply two shares (each of degree one) and that increases the degree of the polynomial to two. To reconstruct the secret value of degree two, we need at least three multiplicative (Shamir's secret) shares.

While we assume that servers do not collude, we will consider two types of adversarial models for the servers in the context of the computation that they perform: (*i*) ***Honest-but-curious*** (HBC) servers that correctly compute the assigned task without tampering with data or hiding answers. It may, however, exploit side information (e.g., the internal state of the server, query execution, background knowledge, and output size) to gain as much information as possible about the stored data/computation/results. The HBC adversarial model is considered widely in many cryptographic algorithms and in DaS model [12], [34], [69]. (*ii*) ***Malicious*** adversarial servers that can delete/insert tuples from the relation, and hence, is a stronger adversarial model than HBC.

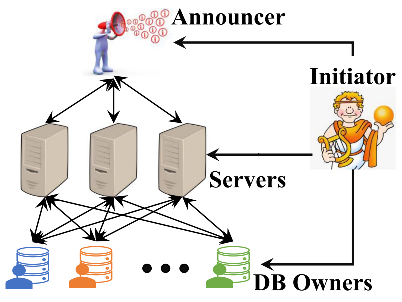


Fig. 1. PRISM model.

3) A ***initiator*or *oracle*** is a trusted entity who is aware of m DB owners and servers. Prior to outsourcing data by DB owners, the initiator shares the identities of servers with DB owners and vice versa. Additionally, the initiator conveys the desired parameters (such as a hash function, Abelian and cyclic group parameters, $\mathcal{PF}$, and $\mathcal{RRG}$) to both servers and DB owners. The initiator's role is similar to that of a trusted certificate authority in a public-key infrastructure, and is trusted by all other entities. The initiator does not have access to any data or results, as it does not store any data, and data/results are not provided to servers via the initiator. The role of the initiator has also been studied in existing PSI work [63], [73].
4) An **announcer** $\mathcal{S}_a$ that participates only in maximum, minimum, and median queries to announce the results. $\mathcal{S}_a$ communicates (not maliciously) with servers and the initiator (and not with DB owners).

C. PRISM Overview

Let us first understand the working of PRISM at the high-level. PRISM contains four phases (see Fig. 1), as follows:

PHASE 0: Initialization: The initiator sends desired parameters (see details in Section IV) related to additive SS, SSS, cyclic group, $\mathcal{PF}$, and $\mathcal{PRG}$ to all entities and informs them about the identity of others from/to whom they will receive/send the data.

PHASE 1: Data Outsourcing by DB owners: DB owners create additive SS or SSS of their data, by following the methods given in Section V for PSI and PSU, Section VI-A for PSI/PSU-sum, and Section VI-C for PSI/PSU-maximum/minimum. Then, they outsource their secret-shared data to non-colluding servers. Note that for the purpose of explanations, we will write the data outsourcing method with query execution. However, it is not required to outsource the data at the time of query.

PHASE 2: Query Generation by the DB owner: A DB owner who wishes to execute SMC over datasets of different DB owners, sends the query to the servers. For generating secret-shared queries for PSI, PSU, count, sum, maximum, and for their verification, the DB owner follows the method given in Sections V and VI.

PHASE 3: Query Processing: The servers process an input query and respective verification method in an oblivious manner, such that neither the query nor the results satisfying the query/verification are revealed to the adversary. Finally, servers transfer their outputs to DB owners.

PHASE 4: Final processing at the DB owners: The DB owner either adds the additive shares or performs Lagrange interpolation on SSS to obtain the answer to the query.

The complexity of algorithms is presented in Table V.

D. Security Property

As mentioned in the adversarial setting in Section III-B, an adversarial server wishes to learn the (entire/partial) input and output data, while a DB owner may wish to know the data of other DB owners. Hence, a secure algorithm must prevent an adversary to learn the data (*i*) from the ciphertext representation of the data, (*ii*) from query execution due to access-patterns (i.e., the adversary can learn the physical locations of tuples that are accessed to answer the query), and (*iii*) from the size of the output (i.e., the adversary can learn the number of tuples satisfy the query). The attacks on a dataset based on access-patterns and output-size are discussed in [13], [40]. In order to prevent these attacks, our security properties are identical to the standard security definition as in [11], [12], [28]. An algorithm is *privacy-preserving* if it maintains DB owners' privacy, data/computation privacy from the servers, and performs identical operations regardless of the inputs.

Privacy from servers requires that datasets of DB owners must be hidden from the server, before/during/after any computation. To prevent frequency analysis, the protocol must ensure that two or more occurrences of the same value in the dataset are represented differently at the server. In the case of PSI/PSU, the servers must not know if a value is common or not, or the number of DB owners who have a particular value in the result set. In the case of aggregation operations, the output of aggregation over an attribute A_x corresponding to the attributes A_c involved in PSI or PSU should not be revealed to servers. In the case of maximum/median/minimum query, the servers must not learn the maximum/minimum value or the identity of the DB owner who possesses such values. The protocol must ensure that the server's behavior is identical for a particular type of query (e.g., PSI or PSU), thereby hiding access patterns and output sizes to prevent the servers from learning anything from the query execution.

DB owner privacy requires that the DB owners must not gain any knowledge about other datasets except their own and the final result of the computation. In the case of PSI/PSU, the DB owners should only learn about the intersection/union set, and they must not know how many DB owners do not have a particular value in their datasets. In the case of aggregation operations, the DB owners should only learn the output of the aggregation operation, and not the individual values on which the aggregation was performed.

Properties of verification: A verification method must be oblivious and find any misbehavior of servers when computing a query. We follow the verification properties from [41] that the verification method cannot be refuted by the majority of the malicious servers and should not leak any additional information.

TABLE V
COMPLEXITIES OF THE ALGORITHMS

Algorithms	Scan at a server		Communication		Interpolation at a DB owner	
	Rounds	Cost	Rounds	Cost from server to a DB owner	Round	Cost
PSI §5.1, PSU §7, PSI/PSU count §6.5	1	b	1	b	1	b
PSI verification §5.2, PSI/PSU count verification §6.5	1	$2b$	1	$2b$	1	$2b$
PSI sum §6.1 †	2	$b(k+1)$	1	bk	1	bk
PSI sum verification §6.1.1 †	2	$2b(k+1)$	1	$2b(k+1)$	1	$2b(k+1)$
PSI average §6.2 †	2	$b(2k+1)$	1	$2bk$	1	$2bk$
PSI average verification †	2	$2b(2k+1)$	1	$2b(2k+1)$	1	$2b(2k+1)$
PSI maximum §6.3	2	$b(k+1)$	2	$b+1$	2	$b+2$
PSI maximum identity revealing §6.3	2	$b(k+1)+m$	3	$b+m+2$	2	$b+m+2$

Notations. $b = \text{Dom}(A_c)$. k = # columns involved in aggregation operations. m = # DB owners. †: Not including the communication cost for computing PSI, since it requires sending the output to only one of the DB owners. This table shows data scanning rounds at the servers for executing the algorithms, data scanning rounds at the DB owner for interpolation of the results, and communication rounds between a server and a DB owner. Also, this table shows the cost in terms of the number of values a server needs to read for algorithms' execution, the number of values a DB owner needs to reads for interpolation, and the number of values a server sends to a DB owner.

TABLE VI
FREQUENTLY USED NOTATIONS IN THE PAPER

Notation	Meaning
A_c	An attribute on which PSI/PSU executes
$\text{Dom}(A_c)$	domain of attribute A_c
A_x	An attribute used in aggregation
m	Number of DB owners
$\mathcal{DB}_i$	The i^{th} DB owner
$A(m)^{\phi}$	ϕ^{th} additive share of the number m
δ	A prime number defining modulo addition δ operation
$\mathbb{G}_{\delta}$	Abelian group under modulo addition δ operation
g	Generator of a cyclic group
η	A prime number defining modulo multiplicative η operation
η'	$\eta \times \alpha$, where $\alpha > 1$
$\mathcal{PRG}$	Pseudo-random number generator
$\mathcal{S}_a$	The announcer
$\mathcal{S}_{\phi}$	A server ϕ, where $\phi \in \{1, 2, 3\}$
$\mathcal{PF}_{s1}, \mathcal{PF}_{s2}$	The permutation functions known to servers
$\mathcal{PF}_{db1}, \mathcal{PF}_{db2}$	The permutation functions known to all DB owners.
$\mathcal{PF}_i$	A permutation function known to the initiator
$\mathcal{PF}$	A permutation function known to both servers and DB owners
$\chi = \{x_1, x_2, \ldots, x_b\}$	A hash table at the j^{th} DB owner of length $b = \lvert\text{Dom}(A_c)\rvert$
$A(x_i)_j^{\phi}$	The ϕ^{th} additive share of an i^{th} element of χ_j of $\mathcal{DB}_j$
$\overline{x_j}$	The complement value of x_j
$\overline{\chi_j}$	A table having the complement values of χ_j
$A(\chi)_j^{\phi}$	The ϕ^{th} additive share of the table χ_j
$A(\overline{\chi})_j^{\phi}$	The ϕ^{th} additive share of the table $\overline{\chi_j}$
$output_i^{\mathcal{S}_{\phi}}$	The output computed for the i-th value at server $\mathcal{S}_{\phi}$
$\ominus$	The modular subtraction operation
$\oplus$	The modular addition operation

IV. ASSUMPTIONS & PARAMETERS

Different entities in PRISM protocols are aware of the following parameters to execute the desired task, and commonly used parameters in this paper are presented in Table VI:

Parameters known to the initiator: The initiator knows all parameters used in PRISM and distributes them to different entities (only once) as they join in PRISM protocols. Note that the initiator can select these parameters (such as η, δ) to be large to support increasing DB owners over time without updating parameters. Thus, when new DB owners join, the initiator simply needs to inform DB owners/servers about the increase in the number of DB owners in the system, but does not need to change *all* parameters.

Additionally, the initiator does the following: (*i*) Selects a polynomial ($\mathcal{F}(x) = a_{m+1}x^{m+1} + a_m x^m + \ldots + a_1 x + a_0$, where $a_i > 0$) of degree more than m, where m is the number of DB owners, and sends the polynomial to all DB owners. This polynomial will be used during the maximum computation. Importantly, this polynomial $\mathcal{F}(x)$ generates values at different DB owners in an order-preserving manner, as will be clear in Section VI-C, and the degree of the polynomial must be more than m to prevent an entity, who has m different values generated using this polynomial, to reconstruct the secret value (a condition similar to SSS); and beyond $m+1$, the degree of the polynomial does not impact the security, in this case. (*ii*) Generates a permutation function $\mathcal{PF}_i$, and produces four different permutation functions that satisfy Eq. (1):

$$\mathcal{PF}_{s1} \odot \mathcal{PF}_{db1} = \mathcal{PF}_{s2} \odot \mathcal{PF}_{db2} = \mathcal{PF}_i \qquad (1)$$

Here, the symbol $\odot$ represents composition of the permutations, and these functions can be selected over a permutation group. The initiator provides $\mathcal{PF}_{s1}$ and $\mathcal{PF}_{s2}$ to all servers and $\mathcal{PF}_{db1}$ and $\mathcal{PF}_{db2}$ to all DB owners.

Parameters known to the announcer: The announcer $\mathcal{S}_a$ knows δ, a prime number used to define modulo addition for an Abelian group (Section III-A). The announcer helps in maximum and median algorithms.

Parameters known to DB owners: All DB owners know the following parameters: (*i*) m, i.e., the number of DB owners. (*ii*) $\delta > m$, (*iii*) η, where η is a prime number used to define modular multiplication for a cyclic group (Section III-A). Note that DB owners do not know the generator g of the cyclic group. (*iv*) A common hash function. (*v*) The domain of the attribute A_c on which they want to execute PSI/PSU. Note that knowing the domain of the attribute A_c does not reveal that which of the DB owner has a value of the domain or not. (Such an assumption is also considered in prior work [37].) (*vi*) Two permutation functions $\mathcal{PF}_{db1}$ and $\mathcal{PF}_{db2}$. (*vii*) The polynomial $\mathcal{F}(x)$ given by the initiator. (*viii*) A permutation function $\mathcal{PF}$, and the same permutation function will also known to servers.

PSI, PSU, sum, average, count algorithms are based on the assumptions 1-5. PSI verification, sum verification, count, and count verification algorithms are based on the Assumptions 1-6. Maximum, maximum verification, and median algorithms are based on the Assumptions 1-8.

Further, we assume that any DB owner or the initiator provides additive shares of m to servers for executing PSI, and the DB owners have only positive integers to compute the maximum. Since the current PSI maximum method uses modular operations (as will be clear in Section VI-C), we cannot handle floating point values directly. Nevertheless, we can find the maximum for a large class of practical situations, where the precision of decimal is limited, say $k > 0$ digits by simply multiplying each number by 10^k and using the current PSI maximum algorithm. For example, we can find the maximum over $\{0.5, 8.2, 8.02\}$ by computing the maximum over $\{50, 820, 802\}$. Designing a more general solution that does not require limited precision is non-trivial.

Parameters known to servers: Servers know the following parameters: (*i*) m, $\delta > m$, the generator g of the cyclic (sub)group of order δ and $\eta' = \alpha \times \eta$ and $\alpha > 1$. Also, based on the group theory, $\eta - 1$ should be divisible by δ. Note that servers do not know η. (*ii*) A permutation function $\mathcal{PF}$, and recall that the same permutation function is also known to DB owners. (*iii*) Two permutation functions $\mathcal{PF}_{s1}$ and $\mathcal{PF}_{s2}$. (*iv*) A common pseudo-random number generator $\mathcal{PRG}$ that generates random numbers between 1 and $\delta - 1$. Note that $\mathcal{PRG}$ is unknown to DB owners. PSI, sum, and average algorithms are based on the Assumptions 1. Maximum, maximum verification, and median algorithms are based on the assumptions 1,2. Count and its verification are based on the Assumptions 1,3. PSU and its verification are based on the Assumptions 1,4.

V. Private Set Intersection (PSI) Query

This section, first, develops a method for finding PSI among $m > 2$ different DB owners on an attribute A_c (which is assumed to exist at all DB owners. Later in Section VI-F, we develop a method to execute PSI over multiple attributes and a method to reduce the communication cost of PSI.

Before going into details of PSI method, we first describe our proposed idea to compute PSI.

High-level idea: Each of $m > 2$ DB owners uses a publicly known hash function to map distinct values of A_c attribute in a table of cells at most $|\text{Dom}(A_c)|$, where $|\text{Dom}(A_c)|$ refers to the size of the domain of A_c. This ensures that if a value $a_j \in A_c$ exists at any DB owner, it is mapped to the same cell in the table by all DB owners. Then, all values of the table are outsourced in the form of additive shares to *two non-colluding servers* $\mathcal{S}_\phi, \phi \in \{1, 2\}$. These servers use an oblivious algorithm to determine the common items/intersection and return a shared output vector of the same length as the received shares from DB owners. Finally, each DB owner adds the results to know the final answer.

Construction: We create the following construction over the elements of a group under addition and the elements of a cyclic group under multiplication. Note that we can select any cyclic group such that $\eta > m$.

$$(x + y) \bmod \delta = 0, (g^x \times g^y) \bmod \eta = 1 \quad (2)$$

Based on this construction, below, we explain PSI finding algorithm:

Step 1. DB owners: Each DB owner finds distinct values in an attribute (A_c, which exists at all DB owners, as per our assumption given in Section IV) and executes the hash function on each value a_i to create a table $\chi = \{x_1, x_2, \ldots, x_b\}$ of length $b = |\text{Dom}(A_c)|$. The hash function maps the value $a_i \in A_c$ to one of the cells of χ, such that the cell of χ corresponding to the value a_i holds 1; otherwise 0. It is important that each cell must contain only a single one corresponding to the unique value of the attribute A_c, and note that if a value $a_i \in A_c$ exists at any

TABLE VII
$\mathcal{DB}_1$

Value	Share 1	Share 2
1	4	-3
0	2	-2
1	3	-2

TABLE VIII
$\mathcal{DB}_2$

Value	Share 1	Share 2
1	3	-2
1	4	-3
0	3	-3

TABLE IX
$\mathcal{DB}_3$

Value	Share 1	Share 2
1	2	-1
0	3	-3
1	4	-3

DB owner, then one corresponding to a_i is placed at an identical cell of χ at the DB owner. The table at $\mathcal{DB}_j$ is denoted by χ_j. Finally, $\mathcal{DB}_j$ creates additive secret-shares of each value of χ_j (i.e., additive secret-shares of either one or zero) and outsources the ϕ^{th}, $\phi \in \{1, 2\}$, share to the server $\mathcal{S}_\phi$. We use the notation $A(x_i)_j^\phi$ to refer to ϕ^{th} additive share of an i^{th} element of χ_j of $\mathcal{DB}_j$. Recall that before the computation starts, the initiator informs the locations of servers to DB owners and vice versa (Section III-B).

Step 2. Servers: Each server $\mathcal{S}_\phi$ ($\phi \in \{1, 2\}$) holds the ϕ^{th} additive share of the table χ (denoted by $A(\chi)_j^\phi$) of j^{th} ($1 \le j \le m$) DB owners and executes Eq. (3):

$$output_i^{\mathcal{S}_\phi} \leftarrow g^{((\oplus_{j=1}^{j=m} A(x_i)_j^\phi) \ominus A(m)^\phi)} \bmod \eta', (1 \le i \le b) \quad (3)$$

where $\oplus$ and $\ominus$ show the modular addition and modular subtraction operations, respectively. We used the symbols $\oplus$ and $\ominus$ to distinguish them from the normal addition and subtraction. Particularly, each server $\mathcal{S}_\phi$ performs the following operations: (*i*) modular addition (under δ) of the i^{th} additive secret-shares from all m DB owners, (*ii*) modular subtraction (under δ) of the result of the previous step from the additive share of m (i.e., $A(m)^\phi$), (*iii*) exponentiation by g to the power the result of the previous step and modulo by η', and (*iv*) sends all the computed b results to the m DB owners.

Step 3. DB owners: From two servers, DB owners receive two vectors, each of length b, and perform modular multiplication (under η) of outputs $output_i^{\mathcal{S}_1}$ and $output_i^{\mathcal{S}_2}$, where $1 \le i \le b$, i.e.,

$$fop_i \leftarrow (output_i^{\mathcal{S}_1} \times output_i^{\mathcal{S}_2}) \bmod \eta \quad (4)$$

This step results in an output array of b elements, which may contain any value. However, if an i^{th} item of χ_j exists at all DB owners, then fop_i must be one, since $\mathcal{S}_\phi$ have added additive shares of m ones at the i^{th} element and subtracted from additive share of m that results in $(g^0 \bmod \eta') \bmod \eta = 1$ at DB owner. Please see the correctness argument below after the example.

Example V.A: Assume three DB owners: $\mathcal{DB}_1$, $\mathcal{DB}_2$, and $\mathcal{DB}_3$; see Tables I, II, and III. For answering a query to find the common disease that is treated by each hospital, DB owners create their tables χ as shown in the first column of Tables VII, VIII, and IX. For example, in Table VIII, $\langle 1, 1, 0\rangle$ corresponds to cancer, fever, and heart diseases, where 1 means that the disease is treated by the hospital. We select $\delta = 5$, $\eta = 11$, and $\eta' = 143$. Hence, the Abelian group under modulo addition contains $\{0, 1, 2, 3, 4\}$, and the cyclic (sub)group (with $g = 3$) under modulo multiplication contains $\{1, 3, 4, 5, 9\}$. Assume additive shares of $m = 3 = (1 + 2) \bmod 5$.

Step *1: DB Owners.* DB owners generate additive shares as shown in the second and third columns of Tables VII, VIII, and IX, and outsource all values of the second and third columns to $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively.

Step *2: Servers.* Server $\mathcal{S}_1$ will return the three values 27, 27, 81, by executing the following computation, to all three DB owners:

$$3^{((((4+3+2) \bmod 5)-1) \bmod 5)} \bmod 143 = 27$$

$$3^{((((2+4+3) \bmod 5)-1) \bmod 5)} \bmod 143 = 27$$

$$3^{((((3+3+4) \bmod 5)-1) \bmod 5)} \bmod 143 = 81$$

Server $\mathcal{S}_2$ will return values 9, 1, and 1 to all three DB owners:

$$3^{((((-3-2-1) \bmod 5)-2) \bmod 5)} \bmod 143 = 9$$

$$3^{((((-2-3-3) \bmod 5)-2) \bmod 5)} \bmod 143 = 1$$

$$3^{((((-2-3-3) \bmod 5)-2) \bmod 5)} \bmod 143 = 1$$

Step *3: DB owners.* The DB owner obtains a vector $\langle 1, 5, 4\rangle$, by executing the following computation (see below). From the vector $\langle 1, 5, 4\rangle$, DB owners learn that cancer is a common disease treated by all three hospitals. However, the DB owner does not learn anything more than this; note that in the output vector, the values 5 and 4 correspond to zero. For instance, $\mathcal{DB}_1$, i.e., hospital 1, cannot learn whether fever and heart diseases are treated by hospital 2, 3, or not.

$$(27 \times 9) \bmod 11 = 1$$

$$(27 \times 1) \bmod 11 = 5$$

$$(81 \times 1) \bmod 11 = 4 \blacksquare$$

Correctness: When we plug Eq. (3) into Eq. (4), we obtain:

$$fop_i = (g^{(\oplus_{j=1}^{j=m} A(x_i)_j^1) \ominus A(m)^1} \times g^{(\oplus_{j=1}^{j=m} A(x_i)_j^2) \ominus A(m)^2} \bmod \eta') \bmod \eta$$
$$= (g^{(\oplus_{j=1}^{j=m} (x_i)_j - m)} \bmod \eta') \bmod \eta$$

We utilize the modular identity, i.e., $(x \bmod \alpha\eta) \bmod \eta = x \bmod \eta$; thus, $fop_i = g^{(\sum_{j=1}^{j=m} (x_i)_j - m)} \bmod \eta$. Only when $\sum_{j=1}^{j=m} (x_i)_j = m$, the result of above expression is one. Otherwise, it is a nonzero number.

Information leakage discussion: is provided in Appendix A, available online.

PSI Result Verification: is provided in Appendix F.1, available online.

VI. Aggregation Operation Over PSI

Prism supports both summary and exemplar aggregations. Below, we describe how Prism implements sum Section VI-A, average Section VI-B, maximum Section VI-C, median Section VI-D and count operations Section VI-E. Also, in our discussion below, we will consider set-based operation PSI on a single attribute A_c. Section VI-F will extend the discussions to support PSI over on multiple attributes and over a large-size domain.

A. PSI Sum Query

A PSI sum query computes the sum of values over an attribute corresponding to common items in another attribute; see example given in Section II. This section develops a method for computing PSI-sum query using both additive and multiplicative shares. Specifically, we use additive shares to identify common items over an attribute A_c, and then employ multiplicative shares (SSS) to compute the sum of shares of an attribute A_x corresponding to the common items in A_c.

Step 1. DB owners: $\mathcal{DB}_j$ creates their χ_j table over the distinct values of A_c attribute by following Step 1 of PSI; see Section V. Here, $\chi_j = \{\langle x_{i1}, x_{i2}\rangle\}$, where $1 \le i \le b$ and $b = |\text{Dom}(A_c)|$, i.e., the i^{th} cell of χ_j contains a pair of values, $\langle x_{i1}, x_{i2}\rangle$, where *(i)* $x_{i1} = 1$, if a value $a_i \in A_c$ is mapped to the i^{th} cell, otherwise, 0; and *(ii)* x_{i2} contains the sum of values of A_x attribute corresponding to a_i; otherwise, 0. $\mathcal{DB}_j$ creates additive shares of x_{i1} (denoted by $A(x_{i1})_j^{\phi}$, $\phi = \{1, 2\}$) and sends to two servers $\mathcal{S}_1$ and $\mathcal{S}_2$. Also, $\mathcal{DB}_j$ creates SSS of x_{i2} (denoted by $S(x_{i2})_j^{\phi}$, $\phi = \{1, 2, 3\}$) and sends to three servers $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$.

Step 2. Servers: Servers $\mathcal{S}_1$ and $\mathcal{S}_2$ find common items using additive shares by implementing Eq. (3) and send all computed b results to all DB owners. Since the result is in additive shared form, it cannot be multiplied to SSS. Thus, servers send the output of PSI to *one of the DB owners selected randomly* and wait to receive multiplicative shares corresponding to common items. The reason of randomly selecting only one DB owner is just to reduce the communication overhead of sending/receiving additive/multiplicative shares, and it does not impact the security. Note that all DB owners can receive the PSI outputs and generate multiplicative shares.

Step 3. DB owners: On receiving b values, the DB owner finds the common items by executing Eq. (4) and generates a vector of length b having 1 or 0 only, where 0 is obtained by replacing random values of fop. Finally, the DB owner creates three SSS of each of the b value, denoted by $S(z_i)^{\phi}$, $\phi = \{1, 2, 3\}$, and sends to three servers.

Step 4. Servers: Servers $\mathcal{S}_{\phi}$,$_{\phi=\{1,2,3\}}$, execute the following:

$$sum_i^{\mathcal{S}_\phi} \leftarrow \textstyle\sum_{j=1}^{j=m} S(x_{i2})_j^{\phi} \times S(z_i)^{\phi}, 1 \le i \le b \qquad (5)$$

Each server multiplies $S(z_i)^{\phi}$ by $S(x_{i2})_j^{\phi}$ of each DB owner, adds the results, and sends them to all DB owners.

Step 5. DB owners: From three servers, all DB owners receive three vectors, each of length b, and perform Lagrange interpolation on each i^{th} value of the three vectors. Thus, the DB owner knows two things: (i) the common item in A_c attribute, and (ii) the sum of the value in A_x corresponding to the common items

TABLE X
$\mathcal{DB}_1$ SHARES FOR PSI SUM

Value	Share 1	Share 2	Cost	Polynomial	CShare 1	CShare 2	CShare 3
1	4	-3	300	$x+300$	301	302	303
0	2	-2	0	$300x+0$	300	600	900
1	3	-2	300	$2x+300$	302	304	306

TABLE XI
$\mathcal{DB}_2$ SHARES FOR PSI SUM

Value	Share 1	Share 2	Cost	Polynomial	CShare 1	CShare 2	CShare 3
1	3	-2	100	$2x+100$	102	104	106
1	4	-3	120	$x+120$	121	122	123
0	3	-3	0	$50x+0$	50	100	150

TABLE XII
$\mathcal{DB}_3$ SHARES FOR PSI SUM

Value	Share 1	Share 2	Cost	Polynomial	CShare 1	CShare 2	CShare 3
1	2	-1	1000	$x+1000$	1001	1002	1003
0	3	-3	10	$10+x$	10	20	30
1	4	-3	500	$x+500$	501	502	503

TABLE XIII
$\mathcal{DB}_1$ CREATING MULTIPLICATIVE SHARES OF PSI OUTPUTS

PSI output		Polynomial	PSIShare 1	PSIShare 2	PSIShare 3
1	1	$x+1$	2	3	4
9	0	$2x+0$	2	4	6
3	0	$3x+0$	3	6	9

in A_c. Section VI-A1 will extend this method in which we will reveal to DB owners only the sum of the values corresponding to the common items, but not the common items.

Example VI-A: Consider the three DB owners: $\mathcal{DB}_1, \mathcal{DB}_2$, and $\mathcal{DB}_3$; see Tables I, II, and III, and consider a query to find the sum of the cost corresponding to the common disease that is treated by each hospital. In this example, we select $\delta = 5$, $\eta = 11$, and $\eta' = 143$. Hence, the Abelian group under modulo addition contains $\{0, 1, 2, 3, 4\}$, and the cyclic (sub)group (with $g = 3$) under modulo multiplication contains $\{1, 3, 4, 5, 9\}$. Assume additive shares of $m = 3 = (1 + 2) \bmod 5$.

STEP 1. DB Owners: DB owners generate additive shares corresponding to the diseases, as we did in Example V.1, and shown in the second and third columns of Tables X, XI, and XII. Also, DB owners find the cost group by diseases, as shown in the fourth column of Tables X, XI, and XII. Then, DB owners select polynomials of the same degree and create three multiplicative shares of the cost, as shown in sixth, seventh, and eighth columns of Tables X, XI, and XII. Additive shares and multiplicative shares are outsourced to the servers.

STEP 2. Servers: Servers $\mathcal{S}_1$ and $\mathcal{S}_2$ compute PSI as we did in STEP 2 of Example V.1 and send the output to one of the DB owners, say $\mathcal{DB}_1$.

STEP *3: DB owner.* $\mathcal{DB}_1$ finds PSI as we did in STEP 3 of Example V.1, replaces random values (the output of STEP 3 computation in Example V.1 by zero), and creates multiplicative of one and zero, as shown in Table XIII:

STEP 4. Servers: On receiving multiplicative shares of PSI outputs from $\mathcal{DB}_1$, servers execute the following computations. $\mathcal{S}_1$ executes the following computation and returns 2808, 862, 2559 to all DB owners.

$$2 \times (301 + 102 + 1001) = 2808$$
$$2 \times (300 + 121 + 10) = 862$$
$$3 \times (302 + 50 + 501) = 2559$$

$\mathcal{S}_2$ executes the following computation and returns 4224, 2968, 5436 to all DB owners.

$$3 \times (302 + 104 + 1002) = 4224$$
$$4 \times (600 + 122 + 20) = 2968$$
$$6 \times (304 + 100 + 502) = 5436$$

$\mathcal{S}_3$ executes the following computation and returns 4224, 2968, 5436 to all DB owners.

$$4 \times (303 + 106 + 1003) = 5648$$
$$6 \times (900 + 123 + 30) = 6318$$
$$9 \times (306 + 150 + 503) = 8631$$

STEP 5. DB owner: DB owners perform Lagrange interpolation over $\langle 2808, 4224, 5648\rangle$, $\langle 862, 2968, 6318\rangle$, $\langle 2559, 5436, 8631\rangle$, and it results in values 1400, 0, 0, i.e., the sum of the cost corresponding to the common disease, i.e., cancer, is 1400.

Note. Sum of values corresponding to common items: In situations where there are multiple common values (e.g., diseases such as cancer and flu are common diseases in our example), the DB owner may want to know the sum of cost corresponding to all common items/disease. It means that the DB owner wish to learn the sum of the cost corresponding to cancer and flu. However, to do so, the DB owner need not know the cost of individual diseases separately and add them up. Doing so will reveal the cost of individual common items. In our approach, the servers can directly add up the values corresponding to all common items after executing the STEP 4 mentioned above and send the final output to the DB owners. The DB owners will receive the desired answer through interpolation under complete security requirements. For instance, the DB owner will learn the sum of the cost corresponding to cancer and flu. It is possible because the cost is stored using multiplicative sharing techniques that are additive homomorphic.

1) A Variant of PSI Sum: One variation of the PSI sum problem involves revealing only the sum of values in attribute A_x that correspond to common items, without disclosing the common items themselves. For instance, if we want to determine the cost of treatment for the common disease treated by all hospitals in Tables I,II, and III, the output should be 1400, without the DB owner learning that this value corresponds to cancer. Such a problem has been motivated and considered in [39].

Our PSI sum method can be easily extended to support such a query, as follows: In STEP 2 of Section VI-A, severs $\mathcal{S}_1$ and $\mathcal{S}_2$ perform a permutation function, say $\mathcal{PF}_{s1}$ on the output of STEP 2 before sending them to one of the DB owners. In STEP 4, on receiving the multiplicative shares from the DB owners, servers first permute back the multiplicative shares and then execute the computation given in STEP 4 of Section VI-A. Finally, before sending the output of STEP 4, servers perform a permutation function, say $\mathcal{PF}_{s2}$. Thus, in STEP 5 of Section VI-A, after executing Lagrange interpolation, DB owners only learn the sum of values corresponding to the common items in A_c.

2) PSI Sum Information Leakage and Verification: Correctness and information leakage of PSI Sum are provided in Appendix B, available online, and the PSI Sum correctness is given in Appendix F.2, available online.

B. PSI Average Query

A PSI average query on cost column corresponding to the common disease in Tables I, II, and III returns {Cancer, 280}. PSI average query works in a similar way as PSI sum query. In short, $\mathcal{DB}_j$ creates $\chi_j = \{\langle x_{i1}, x_{i2}, x_{i3}\rangle\}$, where $1 \leq i \leq b, b = |\text{Dom}(A_c)|$, and x_{i1}, x_{i2} are identical to the values we created in STEP 1 of PSI sum (Section VI-A). The new value x_{i3} contains the number of tuples at $\mathcal{DB}_j$ corresponding to x_{i1}. For example, in case of Table I, one of the values of χ_1 will be {⟨Cancer, 300, 2⟩}, i.e., Table I has two tuples corresponding to Cancer and cost 300. All values x_{i3} are outsourced in multiplicative share form. Then, we follow STEPS 2 and 3 of PSI sum. In STEP 4, the servers also multiply the received i^{th} SSS values corresponding to the common value to x_{i2}, x_{i3} and add the values. Finally, in STEP 5, DB owners interpolate vectors corresponding to all b values of x_{i2}, x_{i3} and find the average by dividing the values appropriately.

Correctness: can be argued similar to PSI sum.

Information leakage: can be argued similar to PSI sum. Note that here we reveal the total number of tuples and the sum of values corresponding to the common values.

C. PSI Maximum Query

This section develops a method for finding the maximum value in an attribute A_x corresponding to the common values in A_c attribute; refer to Section II for PSI maximum example. Here, our objective is to prevent the adversarial server from learning: (i) the actual maximum values outsourced by each DB owner, (ii) what is the maximum value among DB owners and which DB owners have the maximum value. We allow all the DB owners to know the maximum value and/or the identity of the DB owner(s) having the maximum value. We use pick color to highlight the part that is used to reveal the identity of DB owners having maximum to distinguish which part of the algorithm can be avoided based on the security requirements.

In this method, ***each DB owner uses the polynomial $\mathcal{F}(x)$*** given by the initiator; see Section IV to find how we created $\mathcal{F}(x)$. Note that we use this polynomial to generate values at different DB owners in an order-preserving manner by executing the following STEP 3 and Eq. (6).

The method contains at most three rounds, where the first round finds the common values in an attribute A_c by using STEPS 1-3, the second round finds the maximum value in an attribute A_x corresponding to common items in A_c using STEPs 4-5a, the last round finds DB owners who have the maximum value using STEPs 5b-7. Note that ***the third round is not always required***, if (*i*) we do not want to reveal the identity of the DB owner having the maximum value, or (*ii*) values in A_x column across all DB owners are unique.

STEP 1 at DB owner and STEP 2 at servers: These two steps are identical to STEP 1 and STEP 2 of PSI query execution method (given in Section V).

STEP 3. DB owner: On the received outputs (of STEP 2) from servers, DB owners find the common item in the attribute A_c, as in STEP 3 of PSI query execution method (Section V). Now, to find the maximum value in the attribute A_x corresponding to the common item in A_c, DB owners proceeds as follows:

For the purpose of simplicity, we assume that there is only one common item, say y^{th} item. $\mathcal{DB}_i$ finds the maximum, say $\mathcal{M}_{iy}$, in the attribute A_x of their relation corresponding to the common item y. Note that since we assume only one common element, we refer to the maximum element $\mathcal{M}_{iy}$ by $\mathcal{M}_i$. $\mathcal{DB}_i$ executes Eq. (6) to produce values at DB owners in an order-preserving manner:

$$v_i \leftarrow \mathcal{F}(\mathcal{M}_i) + r_i \tag{6}$$

$\mathcal{DB}_i$ implements the polynomial $\mathcal{F}()$ on $\mathcal{M}_i$ and adds a random number r_i (selected in a range between 0 and $\mathcal{M}_i^m$), and it produces a value v_i. Finally, $\mathcal{DB}_i$ creates additive shares of v_i (denoted by $A(v)_i^{\phi}$) and sends them to servers $\mathcal{S}_\phi$, $\phi = \{1, 2\}$. Note that even if $k \geq 2$ DB owners have the same maximum value $\mathcal{M}_i$, by this step, the value v will be different at those DB owners, with a high probability, $1 - \frac{1}{(\mathcal{M}_i)^{(k-1)m}}$, (depending on the range of r_i). Also, if any two numbers $\mathcal{M}_i < \mathcal{M}_j$, then $\mathcal{F}(\mathcal{M}_i) + r_i < \mathcal{F}(\mathcal{M}_j)$ will hold.

STEP 4. Servers: Each server $\mathcal{S}_\phi$ executes the following:

$$input^{\mathcal{S}_\phi}[i] \leftarrow A(v)_i^{\phi}, 1 \leq i \leq m; output^{\mathcal{S}_\phi}[] \leftarrow \mathcal{PF}(input^{\mathcal{S}_\phi}[])$$

Server $\mathcal{S}_\phi$ collects additive shares from each DB owner and places them in an array (denoted by $input^{\mathcal{S}_\phi}[]$), on which $\mathcal{S}_\phi$ executes the permutation function $\mathcal{PF}$. Then, they send the output the permutation function, i.e., $output^{\mathcal{S}_\phi}[]$, to the announcer $\mathcal{S}_a$ that executes the following:

$$foutput^{\mathcal{S}_a}[i] \leftarrow output^{\mathcal{S}_1}[i] + output^{\mathcal{S}_2}[i], 1 \leq i \leq m \tag{7}$$

$$max, index \leftarrow FindMax(foutput^{\mathcal{S}_a}[]) \tag{8}$$

$\mathcal{S}_a$ adds the i^{th} outputs received from $\mathcal{S}_1$ and $\mathcal{S}_2$, and compares all those numbers to find the maximum number (denoted by max). Also, $\mathcal{S}_a$ produces the index position (denoted by $index$) corresponding to the maximum number in $foutput^{\mathcal{S}_3}[]$. Finally, $\mathcal{S}_a$ creates additive secret-shares of max (denoted by $A(max^{\mathcal{S}_\phi})$, $\phi \in \{1, 2\}$), as well as, of $index$ (denoted by $A(index)^{\mathcal{S}_\phi}$), and sends them to $\mathcal{S}_\phi$ ($\phi \in \{1, 2\}$) that forwards such additive shares to DB owners. Note that if the *protocol does not require to reveal the identity* of the DB owner having the maximum value, $\mathcal{S}_a$ does not send additive shares of $index$.

STEP 5a. DB owner: Now, the DB owners' task is to find the maximum value and/or the identity of the DB owner who has the maximum value. To do so, each DB owner performs:

$$\mathsf{max} \leftarrow A(max)^{\mathcal{S}_1} + A(max)^{\mathcal{S}_2} \tag{9}$$

$$\mathsf{index} \leftarrow A(index)^{\mathcal{S}_1} + A(index)^{\mathcal{S}_2}, pos \leftarrow \mathcal{RPF}(\mathsf{index}) \tag{10}$$

The DB owner finds the identity of the DB owner having the maximum value by adding the additive shares and by implementing reverse permutation function $\mathcal{RPF}$. Note that $\mathcal{RPF}$ works since $\mathcal{PF}$ is known to DB owners and servers (see Assumptions given in Section IV). To find the maximum value, they implement $\mathcal{F}(z)$ and $\mathcal{F}(z+1)$ and evaluate $\mathcal{F}(z) \leq \mathsf{max} < \mathcal{F}(z+1)$, where $z \in \{1, 2, \ldots\}$.[1] If this condition holds to be true, then z is the maximum value, and if $z = \mathcal{M}_i$, then the i^{th} DB owner knows that he/she holds the maximum value. Obviously, if the i^{th} DB owner does not hold the maximum value, then $\mathcal{M}_i < \mathcal{F}(\mathcal{M}_i) + r_i < \mathcal{F}(\mathcal{M}_i + 1) \leq \mathcal{F}(z) \leq \mathsf{max}$.

STEP 5b. DB owner: Note that by the end of STEP 5a, the DB owners know the maximum value and the identity of the DB owner having the same maximum value, due to pos. However, if there are more than one DB owner having the maximum value, the other DB owners cannot learn about it. The reason is: the server $\mathcal{S}_a$ can find only the maximum value, while, recall that, if more than one DB owners have the same maximum value, say $\mathcal{M}$, they produce a different value, due to using different random numbers in STEP 3 Eq. (6). Thus, we need to execute this step 5b to know all DB owners having the maximum value.

After comparing its maximum values against max, $\mathcal{DB}_i$ knows whether it possesses the maximum value or not. Depending on this, $\mathcal{DB}_i$ generates a value $\alpha_i = 0$ or $\alpha_i = 1$, creates additive shares of α_i, and sends to $\mathcal{S}_\phi$, $\phi \in \{1, 2\}$.

STEP 6. Servers: Server $\mathcal{S}_\phi$ allocates the received additive shares to a vector, denoted by $fpos$, and sends the vector $fpos$ to all DB owners, i.e., $fpos^{\mathcal{S}_\phi}[i] \leftarrow A(\alpha)_i^{\mathcal{S}_\phi}, 1 \leq i \leq m$.

STEP 7. DB owner: Each DB owner adds the received additive shares to obtain the vector $\mathsf{fpos}[]$.

$$\mathsf{fpos}[i] \leftarrow fpos^{\mathcal{S}_1}[i] + fpos^{\mathcal{S}_2}[i], 1 \leq i \leq m \qquad (11)$$

By $\mathsf{fpos}[]$, DB owners discover which DB owners have the maximum value, since, recall that in STEP 5, $\mathcal{DB}_i$ that satisfies the condition $(\mathcal{F}(\mathcal{M}_i) \leq \mathsf{max} < \mathcal{F}(\mathcal{M}_i + 1))$ requests $\mathcal{S}_\phi$ to place additive share of 1 at $fpos^{\mathcal{S}_\phi}[i]$.

Example VI-C.1: Assume $\eta = 5003$. Refer to Tables I, II, and III, and consider that all hospitals wish to find the maximum age of a patient corresponding to the common disease and which hospitals treat such patients. We assume that all hospitals know cancer as the common disease.

In STEP 3, all hospitals, i.e., DB owners, find their maximum values in the attribute `Age` corresponding to common disease and implement $\mathcal{F}(x) = x^4 + x^3 + x^2 + x + 1$, sent by the initiator.

$$\mathcal{F}(6) = 1555 + 216 = 1771 = (5000 - 3229) \bmod 5003$$

$$\mathcal{F}(8) = 4681 + 1 = 4682 = (5500 - 818) \bmod 5003$$

$$\mathcal{F}(8) = 4681 + 319 = 5000 = (2500 + 2500) \bmod 5003$$

Further, they add random numbers $(216, 1, 319)$ and create additive shares, which are outsourced to $\mathcal{S}_1$ and $\mathcal{S}_2$. In STEP 4, $\mathcal{S}_1$ holds $\langle 5000, 5500, 2500 \rangle$, permutes them, and sends to $\mathcal{S}_a$. $\mathcal{S}_2$ holds $\langle -3229, -818, 2500 \rangle$, permutes them, and sends to $\mathcal{S}_a$.

[1]To reduce the computation cost, we can select number z similar to binary search method.

$\mathcal{S}_a$ obtains $\langle 4682, 5000, 1771 \rangle$ by adding the received shares from $\mathcal{S}_1, \mathcal{S}_2$, and finds 5000 as the maximum value and 'Hospital 2' to which the value belongs. Finally, $\mathcal{S}_a$ creates additive shares of $5000 = (4000 + 1000) \bmod 5003$, additive shares of the identity of the DB owner as $2 = (200 - 198) \bmod 5003$, and sends to DB owners via $\mathcal{S}_1, \mathcal{S}_2$.

In STEP 5a, all hospitals will know the maximum value as 5000 (with random value added) and identity of the DB owner as 2 on which they implement the reverse permutation function to obtain the correct identity as 'Hospital 3'. Then, 'Hospital 1' knows that they do not hold the maximum, since $\mathcal{F}(6) + 216 < \mathcal{F}(7) < 5000$. 'Hospital 2' knows that they hold the maximum, since $\mathcal{F}(8) < 5000 < \mathcal{F}(9)$. Also, 'Hospital 3' knows that they hold the maximum.

To know which hospitals have the maximum value, in STEP 5b, Hospitals 1, 2, 3' create additive shares of 0, 1, 1, respectively, as: $0 = (200 - 200) \bmod 5003$, $1 = (300 - 299) \bmod 5003$, and $1 = (200 - 199) \bmod 5003$, and send to $\mathcal{S}_1$ and $\mathcal{S}_2$. Finally, in STEP 6, $\mathcal{S}_1$ and $\mathcal{S}_2$ send $\langle 200, 300, 200 \rangle$ and $\langle -200, -299, -199 \rangle$ to all hospitals. In STEP 7, hospitals add received shares, resulting in $\langle 0, 1, 1 \rangle$. It shows that 'Hospitals 2, 3' have the maximum value 8.

Correctness and information leakage discussion: is provided in Appendix C, available online.

PSI Maximum Verification: Appendix F.4, available online, provides a method to verify the maximum value.

D. PSI Median Query

A PSI median query over cost column corresponding to disease column over Tables I, II, and III returns $\{\langle \text{Cancer, } 300 \rangle\}$. Note that here we first add the cost of treatment at each DB owner. However, the approach can be extended to deal with individual tuples. For solving PSI median, we extend the method of finding maximum by executing all steps as specified in Section VI-C with an additional process in STEP 2. Particularly, $\mathcal{S}_a$ in STEP 2 of Section VI-C after adding shares, sorts them, and finds the median value. If the number of DB owners is odd (even), then $\mathcal{S}_a$ finds the middle value (two middle values) in the sorted shares.

E. PSI Count Query

We propose an extension to PSI method (Section V) that only reveals the count of common items among DB owners (i.e., the cardinality of the common set), instead of revealing the common items themselves. The approach involves using a permutation function $\mathcal{PF}s1$ known only to the servers $\mathcal{S}_\phi$, to find the common items over χ and permute the final output at the servers before sending the vector (in additive share form) to DB owners. When DB owners compute on the vector received from the servers to determine the final output, the position of one in the vector does not reveal the common items, while the count of ones reveals the cardinality of the common items. The PSI count method follows all the steps of PSI as described in Section V, with the addition of the same permutation function execution by both servers (in STEP 2 of Section V) before sending the output to the DB owners. DB owners then perform the same computations as given in STEP 3 of Section V, and count the ones to obtain the answer.

Correctness: Correctness of PSI count can be argued identically to the correctness of PSI method given in Section V. In addition, since both servers use the same permutation function, it will produce the correct answer at DB owner.

Information leakage discussion: We can argue information leakage at servers and DB owners like Section V. Moreover, PSI count method hides information about which item is common or not in the attribute A_C, among DB owners. Thus, DB owners will only know the number of common items.

PSI Count Verification: is provided in Appendix F.3, available online.

F. Extending PSI Over Multiple Attributes

In the previous sections, we explained PSI over a single attribute (or a set). We can trivially extend it to multiple attributes (or multisets). Particularly, such a query can be expressed in SQL as follows:

SELECT A_c, A_x FROM db_1 INTERSECT ... INTERSECT SELECT A_c, A_x FROM db_m

Recall that in PSI finding method Section V, $\mathcal{DB}_j$ sends additive shares of a table χ_j of length $b = |\text{Dom}(A_c)|$, where A_c was the attributes on which we executed PSI. Now, we can extend this method by creating a table χ_j of length $b = |\Pi_{i>0}\text{Dom}(A_i)|$, where A_i are attributes on which we want to execute PSI. However, as the domain size and the number of attributes increase, such a method incurs the communication overhead. Thus, to apply the PSI method over a large (and real) domain size, as well as, to reduce the communication overhead, we provide a method, named as bucketization-based PSI. This method is provided in detail in Appendix E, available online.

VII. PRIVATE SET UNION (PSU) QUERY

This section develops a method for finding union (denoted by PSU) among $m > 1$ different DB owners over an attribute A_c (which is assumed to exist at all DB owners.

High-level idea: Likewise PSI method (as presented in Section V), each DB owner uses a publicly known hash function to map distinct values of A_c attribute in a table of cells at most $|\text{Dom}(A_c)|$, where $|\text{Dom}(A_c)|$ refers to the size of the domain of A_c, and outsources each element of the table in additive share form to *two servers* $\mathcal{S}_\phi$, $\phi \in \{1, 2\}$. $\mathcal{S}_\phi$ computes the union obliviously, thereby DB owners will receive a vector of length $|\text{Dom}(A_c)|$ having either 0 or 1 of additive shared form. After adding the share for an i^{th} element, DB owners only know whether the element is in the union or not; nothing else.

STEP 1. DB owner: This step is identical to STEP 1 of PSI (Section V).

STEP 2. Server: Each server $\mathcal{S}_\phi$ ($\phi \in \{1, 2\}$) holds the ϕ^{th} additive share of the table χ of m DB owners and executes the following operation:

$$rand[] \leftarrow \mathcal{PRG}(seed)$$

$$output_i^{\mathcal{S}_\phi} \leftarrow ((\textstyle\sum_{j=1}^{j=m} A(x_i)_j^\phi) \times rand[i]) \bmod \delta \tag{12}$$

TABLE XIV
$\mathcal{DB}_1$

Value	Share 1	Share 2
1	4	-3
0	2	-2
1	3	-2
0	1	-1

TABLE XV
$\mathcal{DB}_2$

Value	Share 1	Share 2
1	3	-2
1	4	-3
0	3	-3
0	2	-2

TABLE XVI
$\mathcal{DB}_3$

Value	Share 1	Share 2
1	2	-1
0	3	-3
1	4	-3
0	3	-3

Each server $\mathcal{S}_\phi$ performs the following operations: (*i*) generates b pseudorandom numbers that are between 1 and $\delta - 1$, (*ii*) performs (arithmetic) addition of the i^{th} additive secret-shares from all DB owners, (*iii*) multiplies the resultant of the previous step with i^{th} pseudorandom number and then takes modulo, and (*iv*) sends b results to all DB owners.

STEP 3. DB owner: On receiving two vectors, each of length b, from two servers, DB owners execute modular addition over i^{th} shares of both vectors to know the final answer Eq. (13). It results in either zero or any random number, where zero shows that the i^{th} element of χ is not present at any DB owner, while a random number shows the i^{th} element of χ is present at one of the DB owners.

$$fop_i \leftarrow (output_i^{\mathcal{S}_1} + output_i^{\mathcal{S}_2}) \bmod \delta \tag{13}$$

Example: Assume three DB owners: $\mathcal{DB}_1$, $\mathcal{DB}_2$, and $\mathcal{DB}_3$; see Tables I, II, and III. Also, assume that the domain of diseases contain cancer, fever, heart, and kidney diseases. Three hospitals wish to know the union of diseases treated by all hospitals, i.e., the name of diseases treated by any hospital. We select $\delta = 5$.

STEP 1. DB Owners: DB owners create their table χ as shown in the first column of Tables XIV, XV, and XVI. For example, $\langle 1, 1, 0, 0\rangle$ in Tables XV corresponds to cancer, fever, heart, and and kidney diseases, where 1 means that the disease is treated by the hospital. After that, DB owners generate additive shares as shown in the second and third columns of Tables XIV, XV, and XVI, and outsource all values of the second and third columns to $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively.

STEP 2. Servers: Assume the servers generate the following four pseudorandom numbers: $\langle 2, 3, 1, 4\rangle$. The server $\mathcal{S}_1$ will return the four values 3, 2, 0, 4 by executing the following computation, to all three DB owners:

$$4 + 3 + 2 = 9 \times 2 = 18 \bmod 5 = 3$$

$$2 + 4 + 3 = 9 \times 3 = 27 \bmod 5 = 2$$

TABLE XVII
TABLE STRUCTURE CREATED BY PRISM

Real data column					For verification					Average
OK	PK	LN	SK	DT	vOK	vPK	vLN	vSK	vDT	aOK

$$3+3+4=10\times 1=10 \bmod 5=0$$
$$1+2+3=6\times 4=24 \bmod 5=4$$

Server $\mathcal{S}_2$ will return values 3, 1, 2, and 1 to all three DB owners:

$$-3-2-1=-6\times 2=-12 \bmod 5=3$$
$$-2-3-3=-8\times 2=-24 \bmod 5=1$$
$$-2-3-3=-8\times 2=-8 \bmod 5=2$$
$$-1-2-3=-6\times 2=-24 \bmod 5=1$$

STEP 3. DB owners: The DB owner obtains a vector $\langle 1,3,2,0\rangle$, by executing the following computation (see below). From the vector $\langle 1,3,2,0\rangle$, DB owners learn that cancer, fever, and heart are the disease treated by at least one of the hospitals. However, the DB owner does not learn anything more than this; i.e., they do not learn whether any disease treated by all hospitals or any disease treated by at least two hospitals.

$$3+3=6 \bmod 5=1$$
$$2+1=3 \bmod 5=3$$
$$0+2=2 \bmod 5=2$$
$$4+1=5 \bmod 5=0$$

Correctness and information leakage discussion: is provided in Appendix D, available online.

VIII. EXPERIMENTAL EVALUATION

This section evaluates the scalability of PRISM on different-sized datasets and a different number of DB owners. Also, we evaluate the verification overhead and compare it against other MPC-based systems. We used a 16 GB RAM machine with 4 cores for each of the DB owners and three AWS servers of 32 GB RAM, 3.5 GHz Intel Xeon CPU with 16 cores to store shares. The communication between DB owners and servers were done using the `scp` protocol, and η, δ were 227, 113, respectively.

A. PRISM Evaluation

Dataset generation: We used five columns (Orderkey (OK), Partkey (PK), Linenumber (LN), Suppkey(SK), and Discount (DT)) of LineItem table of TPC-H benchmark. We experimented with domain sizes (i.e., the number of values) of 5 M and 20 M for the ***OK column*** on which we executed PSI and PSU. Further, we selected at most ***50 DB owners***. To the best of our knowledge, this is the first such experiment of multi-owner large datasets. OK column is used for PSI/PSU, and other columns were used for aggregation operations. To generate secret-shared dataset, each DB owner maintained a LineItem table containing at most 5 M (20 M) OK values. To outsource the database, each DB owner did the following:

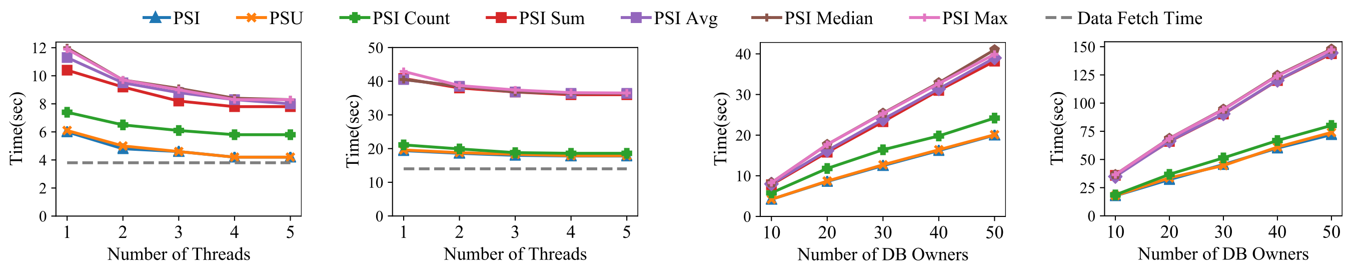
1) Created a table of 11 columns, as shown in Table XVII, in which the first five columns contain the secret-shared data of LineItem table, the next five columns contain the verification data for the first five columns, and the last column (aOK) was used for computing the average. All verification column names are prefixed with the character 'v.'
2) First column of Table XVII was created over OK column of LineItem table (by following STEP 1 of Section V) for executing PSI/PSU over OK. vOK column was created to verify PSI results (by following STEP 1 of Section F.1).
3) Columns PK and vPK were created using the following command: `select OK, sum(PK) from LineItem group by OK`. Other columns $\langle$LN, SK, DT, vLN, vSK, vDT$\rangle$ were created by using similar SQL commands.
4) Columns aOK was created using the following command: `select count(*) from LineItem group by OK`.
5) Finally, permute all values of verification columns and create additive shares of $\langle$OK and vOK$\rangle$, as well as, multiplicative shares of all remaining columns.

Share generation time: The time to generate two additive shares and three multiplicative shares of the respective first five columns of Table XVII in the case of 5 M (or 20 M) OK domain size was 121 s (or 548 s). Furthermore, the time for creating each additional column for verification took 20 s (or 90 s) in the case of 5 M (or 20 M) domain values.

Exp 1. PRISM performance on multi-threaded implementation at AWS: Since identical computations are executed on each row of the table, we exploit multiple CPU cores by writing a parallel implementation of PRISM. The parallel implementation divides rows into multiple blocks with each thread processing a single block. We increased the number of threads from 1 to 5; see Fig. 2, while fixing DB owners to 10. Increasing threads more than 5 did not provide speed-up, since reading/writing of data quickly becomes the bottleneck as the number of threads increase. Observe that the data fetch time from the database remains (almost) identical; see Fig. 2.

PSI and PSU queries: Fig. 2 shows the time taken by PSI/PSU over the OK column. Observe that as the number of values in OK column increases (from 5 M to 20 M), the time increases (almost) linearly from 4 s to 18 s, respectively.

Aggregation queries over PSI: We executed PSI count, average, sum, maximum, and median queries; see Fig. 2. Observe that the processing time of PSI count is almost the same as that of PSI, since it involves only one round of computation in which we permute the output of PSI. In contrast, other aggregation operations (sum, average, maximum, and median) incur almost twice processing cost at servers, since they involve computing PSI over OK column in the first round and, then, computing aggregation in the second round. For this experiment,



(a) 5M OK domain size (1-5M). (b) 20M OK domain size (1-20M). (c) 5M OK domain size (1-5M). (d) 20M OK domain size (1-20M).

Fig. 2. Exp 1. (a)–(b) PRISM performance on multi-threaded implementation at AWS. (c)–(d) Exp 2. PRISM dealing with multiple DB owners.

TABLE XVIII
EXP 1. MULTI-COLUMN AGGREGATION QUERY PERFORMANCE (TIME IN SECONDS)

Data size	Sum over different attributes				Max over different attributes			
	1	2	3	4	1	2	3	4
5M	8.2	12.1	15.9	20.4	10	14.6	19	23.5
20M	33.4	48.6	63.5	81.9	36.6	53.3	70	87.4

we computed sum only over DT column and maximum/median over PK column. Table XVIII shows the impact of computing sum and maximum over multiple attributes (from 1 to 4). As we increase the number of attributes, the computation of respective aggregation operation also increases, due to additional addition/multiplication/modulo operations on additional attributes.

Exp 2. Impact of the number of DB owners: Since we developed PRISM to deal with multiple DB owners, we investigated the impact of DB owners by selecting 10, 20, 30, 40, 50 DB owners, for two different domain sizes of OK column. Fig. 2 shows the server processing time for PSI, PSU, and aggregation over PSI. Note that as the number of DB owners increases, the computation time at the server increases linearly, due to linearly increasing number of addition/multiplication/modulo operations; e.g., on 5 M OK values, PSI processing took 4.2 s, 8.6 s, 12.5 s, 16.2 s, and 20 s in the case of 10, 20, 30, 40, 50 DB owners.

Exp 3. Impact of communication cost: PRISM protocols involve at most two rounds, where servers send data of size equal to the domain size in the first and second rounds of query execution. Thus, it is required to measure the impact of communication cost, since it may affect the overall performance. Among the proposed protocols, the maximum amount of data flows for maximum/median queries, due to first receiving the answers of PSI, then additive share transmission from each DB owner to a server, and finally, receiving the answer of the maximum query from a server to DB owners. Here, the overall data was transmitted of size 60 MB (240 MB) in the case of 5 M (20 M) OK values and took 1.2 s (4.8 s), 0.6 s (2.4 s), 0.1 s (0.4 s) on slow (50 MB/s), medium (100 MB/s), and fast (500 MB/s) speed of data transmission. To measure the communication cost, we simulated network cost by finding appropriate delays in the transmission, where delay was determined by dividing data size by the network speed.

Additional Experiments: On the use of bucketization-based PSI and result verification are provided in Appendix G, available online.

B. Comparing With Other Works

We compare PRISM against the state-of-the-art cloud-based industrial MPC-based systems: Galois Inc.'s Jana [4], since it provides the identical security guarantees at servers as PRISM. To evaluate Jana, we inserted two LineItem tables (each of 1 M rows) having $\langle$OK, PK, LN, SK, DT$\rangle$ columns and executed join on OK column. However, the join execution took more than 1 h to complete.

[1], [2], [42], [43], [44], [53], [63] provide cloud-based PSI/PSU/aggregation techniques/systems. ***We could not experimentally compare* PRISM *against such systems***, since none of them is not open source.[2] Thus, in Table XIX, we report experimental results from those papers, just for intuition purposes. With the exception of [42], none of the techniques supports large-sized dataset, has quadratic/exponential complexity or uses expensive cryptographic techniques [63]. While [42] scales better, it does not support aggregation and, moreover, reveals which item is in the intersection set. For a fair comparison, we report PRISM results only for two DB owners in Table XIX, since other papers do not provide experimental results for more than two DB owners. Recall that in our experiments (Fig. 2(c)), PRISM supports 50 DB owners and takes at most ≈41 seconds on 5 M values. Further note that, in the case of 1B values and two DB owners, PRISM takes ≈ 7.3mins, unlike [42] that took ≈10mins.

There are several non-cloud-based PSI approaches. However, ***such approaches cannot be directly compared against*** PRISM, due to a different model used (in which DB owners communicate amongst themselves and do not outsource data to the cloud) and/or different security properties. Just to put some numbers in this context, recent work [48] took 304 s in the case of 14 DB owners each with 1 M values, and [39] took at least ≈400 s for PSI sum on 100 K values.

Comparison between PRISM and OBSCURE: We compare PRISM and secret-sharing-based OBSCURE [33] in Appendix H, available online.

Comparison with other PSI/PSU finding approaches: A survey of PSI protocols may be found in [58]. Existing PSI protocols are based on homomorphic encryption [14], polynomial evaluation [27], a special encryption technique for comparing value [51], garbled-circuit techniques [37], hashing [26], [59], [67], hashing and oblivious pseudorandom

[2]None of these techniques have open sources implementations, except [5]. We installed [5] that works for a very small data and incurs runtime errors. We have reported this issue to the author as well.

TABLE XIX
COMPARISON OF EXISTING *CLOUD-BASED* TECHNIQUES AGAINST PRISM

Papers	[44] & [53]	[63]	[2]	[1]	[42]	[43]	Jana [4]†	SMCQL [5]	Sharemind [7]	Conclave [68]	‡PRISM
Operations supported	PSI	PSI	PSI	PSI	PSI	PSI	PSI, PSU, aggregation	PSI via join & aggregation			**PSI, PSU, aggregation**
Verification Support	×	×	×	✓	✓	×	×	×	×	×	✓
Scalability based on experiments reported (dataset size & time)	N/A	32768 (≈50 **m**)	1 million (≈2 **h**)	32768 (≈16 **m**)	1 billion (≈10 **m**)	1000 (≈9 **m**)	1 million (≈1 **h**)	>23 million (≈23 **h**)	30000 (>2 **h**)	4 million (8 **m**)	**20 million (At most 8 s)**
Communication among servers	N/A	N/A	N/A	N/A	N/A	N/A	**Yes** ∗	**Yes** ∗	**Yes** ∗	**Yes** ∗	**No**
Computational Complexity	$\mathcal{O}(n^m)$	$\mathcal{O}(\alpha mn)$	$\mathcal{O}(n^m)$	$\mathcal{O}(mn^2)$	$\mathcal{O}(mn)$ ‡‡	$\mathcal{O}(n^m)$	$\mathcal{O}(n^m)$	N/A ∗	$\mathcal{O}(n^m)$	N/A ∗	$\boldsymbol{\mathcal{O}(mX)}$

Notes. (*i*) The **scalability numbers are taken from the respective papers.** (*ii*) Results of Sharemind [7] are taken from Conclave [68] experimental comparison. (*ii*) #DB owners were in each paper was reported two; thus, we executed PRISM for two DB owners for this table. (*iv*) Only Jana, SMCQL, Sharemind, and Conclave provide identical security like PRISM. (*v*) **h**: hours. **m**: minutes. **s**: seconds. †: We setup Jana for two DB owners each with 1M values in our experiments. ‡: Conclave [68] uses a trusted party. **Yes**: Requires communication among servers. **No**: No communication among servers. ∗: Based on MPC-based systems. ∗∗: N/A because executing operation in cleartext or at the trusted party. m: #DB owners. n: DB size. X: domain size. ‡‡: A insecure technique that reveals the size of the intersection, and hence fast. α: The cost of Bilinear Map pairing technique.

functions (OPRF) [47], a variant of OPRF known as programmable OPRF [48] Bloom-filter (for PSI and union finding) [55], oblivious Bloom-filter [23], circuit evaluation (e.g., GMW [31] and [3]), and oblivious transfer [58], [61]. Polynomial evaluation-based approach [27] can also be extended the model to deal with multiple DB owners. However, these techniques may suffer from one or more of the following problems: multiple communication rounds, lack of support for multiple DB owners [17], and/or incapable to execute computation like PSI-count or sum queries (except [68] that uses a trusted third party and [39] that is based on homomorphic encryption).

Kamara et al. [42] proposed an encryption-based approach that deals with a large-size dataset, but either reveals the size of the intersection to the adversary or defer the intersection finding to only DB owners. [1], [42] provided watermark-based verification approach for encrypted datasets, where DB owners insert negotiated values (among them) in the real dataset. [21] proposed Diffie Hellman assumption-based PSI cardinality finding for two DB owners under malicious server. [20] also proposed public-key encryption-based PSI for malicious adversary, where one of the two DB owners can behave maliciously during protocol execution. There are other works under a similar model that consider one of the DB owners out of at most three DB owners is malicious [29], [49], [62], [64] and provide algorithms for PSI. Such a model is extended for multiple DB owners using Bloom filter in [72] and using homomorphic encryption in [36], [44], [45]. [50], [56] proposed information-theoretically secure PSI for multiple DB owners, who communicate with each other during the computation. [50], [56] require more than one round to obtain the answer. However, all such techniques either do not support multiple DB owners [17], limited to PSI, or cannot be deployed at the cloud (except [44]).

Only techniques given in [1], [2], [4], [5], [7], [42], [43], [44], [53], [63], [68] are developed for the cloud settings; as we compared in Table XIX.

In contrast, PRISM is applicable to the cloud setting, deals with a malicious adversarial cloud, provides result verification methods, supports different aggregation operations, and requires at most one round between the server and DB owner to answer PSI, PSU, count. PRISM requires at most two rounds in answering PSI sum, maximum, and median queries.

Comparison with maximum/minimum finding approaches: Since the classic Millionaire's problem (for finding the maximum number between two DB owners) has been proposed by Yao [71], many schemes about comparison/maximum finding have been proposed. [22] and [57] proposed bit-wise less than operation that may be used to find the maximum number. Sepia [10] modified the approach of [57] for fining less than operation. [66] extended the approach of [25] and used Yao's approach [71] for finding top-k items. Similarly, [9] proposed top-k items finding approach based on [66]. Several SMC sorting algorithms have been proposed (e.g. [8], [35]), such algorithms may also be used to find the maximum number. [19] proposed a technique for confirming the maximum number, if the maximum number is known; however, [19] cannot compute the maximum/minimum. All such techniques show limitations: many communication rounds, restricted to two DB owners, quadratic computation cost at servers, not dealing with malicious adversaries in the cloud setting, and/or no support for result verification.

IX. CONCLUSION

This paper describes PRISM based on secret-sharing that allows multiple DB owners to outsource data to (a majority of) non-colluding servers that can behave like honest-but-curious servers and malicious servers in terms of the computation that they perform. It exploits the additive and multiplicative homomorphic property of secret-sharing techniques to implement both set operations and aggregation functions efficiently. Experimental results show PRISM scales to both a large number of DB owners and to large datasets, compared to existing systems. Future directions include dealing with: (*i*) multiple attributes more efficiently than bucketization, (*ii*) dealing with malicious DB owners, and (*iii*) a broader set of SQL queries.

REFERENCES

[1] A. Abadi et al., "VD-PSI: Verifiable delegated private set intersection on outsourced private datasets," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2016, pp. 149–168.

[2] A. Abadi, S. Terzis, R. Metere, and C. Dong, “Efficient delegated private set intersection on outsourced private datasets,” *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 4, pp. 608–624, Jul./Aug. 2019.

[3] T. Araki et al., “High-throughput semi-honest secure three-party computation with an honest majority,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 805–817.

[4] D. W. Archer et al., “From keys to databases - real-world applications of secure multi-party computation,” *Comput. J.*, vol. 61, no. 12, pp. 1749–1771, 2018.

[5] J. Bater et al., “SMCQL: Secure query processing for private data networks,” in *Proc. VLDB Endowment*, vol. 10, no. 6, pp. 673–684, 2017.

[6] M. Blum et al., “How to generate cryptographically strong sequences of pseudo-random bits,” *SIAM J. Comput.*, vol. 13, no. 4, pp. 850–864, 1984.

[7] D. Bogdanov et al., “Sharemind: A framework for fast privacy-preserving computations,” in *Proc. Eur. Symp. Res. Comput. Secur.*, 2008, pp. 192–206.

[8] D. Bogdanov et al., “A practical analysis of oblivious sorting algorithms for secure multi-party computation,” in *Proc. Nordic Conf. Secure IT Syst.*, 2014, pp. 59–74.

[9] M. Burkhart et al., “Fast privacy-preserving top-k queries using secret sharing,” in *Proc. 19th Int. Conf. Comput. Commun. Netw.*, 2010, pp. 1–7.

[10] M. Burkhart et al., “SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics,” in *Proc. USENIX Conf. Secur.*, 2010, Art. no. 15.

[11] R. Canetti., “Security and composition of multiparty cryptographic protocols,” *J. Cryptol.*, vol. 13, no. 1, pp. 143–202, 2000.

[12] R. Canetti et al., “Adaptively secure multi-party computation,” in *Proc. 28th Annu. ACM Symp. Theory Comput.*, 1996, pp. 639–648.

[13] D. Cash et al., “Leakage-abuse attacks against searchable encryption,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2015, pp. 668–679.

[14] H. Chen et al., “Fast private set intersection from homomorphic encryption,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 1243–1255.

[15] J. H. Cheon et al., “Multi-party privacy-preserving set intersection with quasi-linear complexity,” *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. 95-A, no. 8, pp. 1366–1378, 2012.

[16] K. Chida et al., “An efficient secure three-party sorting protocol with an honest majority,” *IACR Cryptol. ePrint Arch.*, vol. 2019, 2019, Art. no. 695.

[17] S. G. Choi et al., “Efficient three-party computation from cut-and-choose,” in *Proc. Annu. Cryptol. Conf.*, 2014, pp. 513–530.

[18] R. M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods*, Berlin, Germany: Springer, 2013.

[19] H. Corrigan-Gibbs et al., “Prio: Private, robust, and scalable computation of aggregate statistics,” in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2017, pp. 259–282.

[20] E. D. Cristofaro et al., “Linear-complexity private set intersection protocols secure in malicious model,” in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2010, pp. 213–231.

[21] E. D. Cristofaro et al., “Fast and private computation of cardinality of set intersection and union,” in *Proc. Int. Conf. Cryptol. Netw. Secur*, 2012, pp. 218–231.

[22] I. Damgård et al., “Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation,” in *Proc. Theory Cryptogr. Conf.*, 2006, pp. 285–304.

[23] C. Dong et al., “When private set intersection meets big data: An efficient and scalable protocol,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2013, pp. 789–800.

[24] R. Egert et al., “Privately computing set-union and set-intersection cardinality via bloom filters,” in *Proc. Australas. Conf. Inf. Secur. Privacy*, 2015, pp. 413–430.

[25] R. Fagin, “Combining fuzzy information from multiple systems,” *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 83–99, 1999.

[26] B. H. Falk et al., “Private set intersection with linear communication from general assumptions,” *18th ACM Workshop Privacy Electron. Soc.*, 2019, pp. 14–25.

[27] M. J. Freedman et al., “Efficient private matching and set intersection,” in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 2004, pp. 1–19.

[28] M. J. Freedman et al., “Keyword search and oblivious pseudorandom functions,” in *Proc. Theory Cryptogr. Conf.*, 2005, pp. 303–324.

[29] J. Furukawa et al., “High-throughput secure three-party computation for malicious adversaries and an honest majority,” in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2017, pp. 225–255.

[30] O. Goldreich et al., “How to construct random functions,” *J. ACM*, vol. 33, no. 4, pp. 792–807, 1986.

[31] O. Goldreich et al., “How to play ANY mental game or A completeness theorem for protocols with honest majority,” in *Proc. Conf. Symp. Theory Comput.*, 1987, pp. 218–229.

[32] D. M. Goldschlag et al., “Onion routing,” *Commun. ACM*, vol. 42, no. 2, pp. 39–41, 1999.

[33] P. Gupta et al., “Obscure: Information-theoretic oblivious and verifiable aggregation queries,” in *Proc. VLDB Endowment*, vol. 12, no. 9, pp. 1030–1043, 2019.

[34] H. Hacigümüs et al., “Executing SQL over encrypted data in the database-service-provider model,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 216–227.

[35] K. Hamada et al., “Practically efficient multi-party sorting protocols from comparison sort algorithms,” in *Proc. Int. Conf. Inf. Secur. Cryptol.*, 2012, pp. 202–216.

[36] C. Hazay et al., “Scalable multi-party private set-intersection,” in *Proc. IACR Int. Workshop Public Key Cryptogr.*, 2017, pp. 175–203.

[37] Y. Huang et al., “Private set intersection: Are garbled circuits better than custom protocols?,” in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012.

[38] R. Inbar et al., “Efficient scalable multiparty private set-intersection via garbled bloom filters,” in *Proc. Int. Conf. Secur. Cryptogr. Netw.*, 2018, pp. 235–252.

[39] M. Ion et al., “On deploying secure computing commercially: Private intersection-sum protocols and their business applications,” *IACR Cryptol. ePrint Arch.*, vol. 2019, 2019, Art. no. 723.

[40] M. S. Islam et al., “Access pattern disclosure on searchable encryption: Ramification, attack and mitigation,” in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, vol. 20, 2012, pp. 12.

[41] W. Jiang et al., “Transforming semi-honest protocols to ensure accountability,” *Data Knowl. Eng.*, vol. 65, no. 1, pp. 57–74, 2008.

[42] S. Kamara et al., “Scaling private set intersection to billion-element sets,” in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2014, pp. 195–215.

[43] F. Kerschbaum, “Collusion-resistant outsourcing of private set intersection,” in *Proc. Annu. ACM Symp. Appl. Comput.*, 2012, pp. 1451–1456.

[44] F. Kerschbaum ., “Outsourced private set intersection using homomorphic encryption,” in *Proc. 7th ACM Symp. Inf. Comput. Commun. Secur.*, 2012, pp. 85–86.

[45] L. Kissner et al., “Privacy-preserving set operations,” in *Proc. Annu. Int. Cryptol. Conf.*, 2005, pp. 241–257.

[46] V. Kolesnikov et al., “Efficient batched oblivious PRF with applications to private set intersection,” *IACR Cryptol. ePrint Arch.*, vol. 2016, 2016, Art. no. 799.

[47] V. Kolesnikov et al., “Efficient batched oblivious PRF with applications to private set intersection,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 818–829.

[48] V. Kolesnikov et al., “Practical multi-party private set intersection from symmetric-key techniques,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 1257–1272.

[49] P. H. Le et al., “Two-party private set intersection with an untrusted third party,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2019, pp. 2403–2420.

[50] R. Li et al., “An unconditionally secure protocol for multi-party set intersection,” in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2007, pp. 226–236.

[51] Y. Li et al., “Delegatable order-revealing encryption,” in *Proc. 7th ACM Symp. Inf. Comput. Commun. Secur.*, 2019, pp. 134–147.

[52] Y. Lindell ., “Secure multiparty computation (MPC),” *IACR Cryptol. ePrint Arch.*, vol. 2020, 2020, Art. no. 300.

[53] F. Liu et al., “Encrypted set intersection protocol for outsourced datasets,” in *Proc. IEEE Int. Conf. Cloud Eng.*, 2014, pp. 135–140.

[54] S. Madden et al., “TAG: A tiny aggregation service for ad-hoc sensor networks,” in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2002.

[55] D. Many, M. Burkhart, and X. Dimitropoulos, “Fast private set operations with sepia,” *ETZ G93*, 2012.

[56] G. S. Narayanan et al., “Multi party distributed private matching, set disjointness and cardinality of set intersection with information theoretic security,” in *Proc. Int. Conf. Cryptol. Netw. Secur.*, 2009, pp. 21–40.

[57] T. Nishide et al., “Multiparty computation for interval, equality, and comparison without bit-decomposition protocol,” in *Proc. Int. Workshop Public Key Cryptogr.*, 2007, pp. 343–360.

[58] B. Pinkas et al., “Faster private set intersection based on OT extension,” in *Proc. USENIX Secur. Sump.*, 2014, pp. 797–812.

[59] B. Pinkas et al., "Phasing: Private set intersection using permutation-based hashing," in *Proc. USENIX Secur. Symp.*, 2015, pp. 515–530.

[60] B. Pinkas et al., "Scalable private set intersection based on OT extension," *ACM Trans. Priv. Secur.*, vol. 21, no. 2, pp. 7:1–7:35, 2018.

[61] B. Pinkas et al., "SpOT-Light: Lightweight private set intersection from sparse OT extension," in *Proc. Annu. Int. Cryptol. Conf.*, 2019, pp. 401–431.

[62] B. Pinkas et al., "PSI from PaXoS: Fast, malicious private set intersection," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2020, pp. 739–767.

[63] S. Qiu et al., "Identity-based private matching over outsourced encrypted datasets," *IEEE Trans. Cloud Comput.*, vol. 6, no. 3, pp. 747–759, 2018.

[64] P. Rindal et al., "Malicious-secure private set intersection via dual execution," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 1229–1242.

[65] A. Shamir .,, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[66] J. Vaidya et al., "Privacy-preserving top-K queries," in *Proc. 21st Int. Conf. Data Eng.*, 2005, pp. 545–546.

[67] J. Vaidya et al., "Secure set intersection cardinality with application to association rule mining," *J. Comput. Secur.*, vol. 13, no. 4, pp. 593–622, 2005.

[68] N. Volgushev et al., "Conclave: Secure multi-party computation on big data," in *Proc. 14th EuroSys Conf.*, 2019, pp. 3:1–3:18.

[69] C. Wang et al., "Secure ranked keyword search over encrypted cloud data," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, 2010, pp. 253–262.

[70] F. Wang et al., "Splinter: Practical private queries on public data," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2017, pp. 299–313.

[71] A. C. Yao, "How to generate and exchange secrets (extended abstract)," in *Proc. 27th Annu. Symp. Foundations Comput. Sci.*, 1986, pp. 162–167.

[72] E. Zhang et al., "Efficient multi-party private set intersection against malicious adversaries," in *Proc. ACM SIGSAC Conf. Cloud Comput. Secur. Workshop*, 2019, pp. 93–104.

[73] Q. Zheng et al., "Verifiable delegated set intersection operations on outsourced encrypted data," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2015, pp. 175–184.

Shantanu Sharma received the MTech degree in computer science from the National Institute of Technology, Kurukshetra, India, in 2011, and the PhD degree in computer science from Ben-Gurion University, Israel, in 2016. He is an assistant professor with the Department of Computer Science, New Jersey Institute of Technology, USA. During his Ph.D. degree, he worked with Prof. Shlomi Dolev and Prof. Jeffrey Ullman. He was awarded a gold medal for the first position in his MTech degree. His research interests include secure and privacy-preserving database systems, trustworthy smart spaces, and distributed algorithms.

Yin Li received the BSc and MSc degrees from Information Engineering University, Zhenzhou, in 2004 and 2007, respectively, and the PhD degree in computer science from Shanghai Jiaotong University (SJTU), Shanghai, in 2011. He is an associate professor with the School of Computer Science and Technology, Dongguan University of Technology, China. Previously, he was an associate professor with the Department of Computer Science and Technology, Xinyang Normal University, China. He was a postdoc with the Ben-Gurion University of the Negev, Israel. His current research interests include algorithm and architectures for computation in finite field, computer algebra, and secure cloud computing.

Sharad Mehrotra (Fellow, IEEE) received the PhD degree in computer science from the University of Texas, Austin, in 1993. He is a distinguished professor with the Department of Computer Science, University of California, Irvine. Previously, he was a professor with the University of Illinois at Urbana Champaign. He has received numerous awards and honors, including the 2011 SIGMOD Best Paper Award, 2007 DASFAA Best Paper Award, SIGMOD test of time award, 2012, DASFAA ten year best paper awards for 2013 and 2014, 1998 CAREER Award from the US National Science Foundation (NSF), and ACM ICMR best paper award for 2013. His primary research interests include the area of database management, distributed systems, secure databases, and Internet of Things. He is an IEEE Fellow and an ACM Fellow.

Nisha Panwar received the MTech degree in computer engineering from the National Institute of Technology, Kurukshetra, India, in 2011, and the PhD degree in computer science from Ben-Gurion University, Israel, in 2016, where he worked with Prof. Shlomi Dolev and Prof. Michael Segal. She is an assistant professor with Augusta University, Georgia. She was a postdoc with the University of California, Irvine, USA. Her research interests include security and privacy issues in IoT systems, as well as, in vehicular networks, computer network and communication, and distributed algorithms.

Peeyush Gupta received the MTech degree in computer science from the Indian Institute of Technology, Bombay, India, in 2013. He is currently working toward the PhD degree advised by Prof. Sharad Mehrotra with the University of California, Irvine, USA. His research interests include IoT data management, time series database systems, and data security and privacy.

Dhrubajyoti Ghosh is currently working toward the PhD degree, advised by Prof. Sharad Mehrotra, with the University of California, Irvine, USA. His research interests include data security and privacy.