PrivShare: Privacy-Preserving Query Processing on Multiple Sources

Xiaoyuan Liu[1], John Yang[1], Tynan Sigg[1], Ni Trieu[2], Dawn Song[1]

[1]*University of California, Berkeley*
[2]*University of Arizona*

Abstract

Data collaboration has become essential in many modern digital applications. Financial accounting, cyber threat intelligence gathering, and even the contact tracing of pandemics require to collect data from different sources. However, existing solutions provide efficiency at the cost of privacy, deployment difficulties, requiring non-collusion assumptions or a trusted execution environment. In addition, targeting specific application scenarios, these schemes often lack generalizability.

In this paper, we design PrivShare, a novel privacy-preserving data infrastructure that enables secure, controllable, and efficient SQL queries to sensitive data from multiple parties. To the best of our knowledge, PrivShare is the first system to process secure multi-party queries using homomorphic encryption. PrivShare is deployment-friendly, secure against malicious adversaries, and its query processing protocol only requires one round of communication.

The technical core of our system is the oblivious evaluation of equality check with grouped binary matching and range query with decomposition, which is used to perform a query predicate. We evaluate the performance of PrivShare by providing a micro-benchmark for its building blocks and investigating two real-world applications. We show that a full deployment of PrivShare potentially supports minute-level secure query processing on millions of records.

1 Introduction

Data collaboration has become imperative in many real-world applications. Today's settings often involve multiple collaborative parties looking to combine intelligence as a means for greater insight. Of the myriad of examples, we are particularly interested in scenarios where these parties are mutually untrusting, but are still interested in combining data and running queries on the conglomerated datasets. In cybersecurity, government agencies often collaborate with one another or hire private contractors to analyze such data. Hospitals and medical practitioners would benefit greatly from jointly measuring and predicting the probability of illness based on a patient's data that may be distributed across multiple institutions. In the financial world, systemic risk or individual credit can be assessed without compromising the confidentiality of the entities being investigated. In these scenarios and many additional practical use cases, there is a clear need for collaborative databases that can preserve the privacy of not only data, but also the queries being run on top of them. Databases that are capable of hiding both the content of an object in addition to compromising behavior such as access patterns are necessary to defend against a wide variety of vulnerabilities and exploits towards breaking private data access.

While many scenarios discussed above have different workflows, most of them can be formalized as the problem of processing secure queries. The data providers have some sensitive data and want to share it in a controllable way, while the client tries to analyze that data without revealing access patterns. Having the obliviousness in running queries, the client is confident to provide sensitive and accurate query details without exposing their preference or interest in data.

By having privacy guarantees for all participants on both sides, we can use this abstraction to cover many real-world applications. Considering multiple mutually-untrusted data providers, we extend the above abstraction to a multiparty setting. Different data providers can collaboratively host a distributed query service but protect their original data records from each other.

Previous privacy query systems [19, 23, 24, 26] may not be perfectly matched with the current cloud infrastructure and suffer from deployment difficulties. To make their construction efficient, some require non-colluding servers [19, 24] and some rely on hardware Trusted Execution Environments (TEE) [23, 26]. In addition, protocols in these systems require multiple rounds of communication and have sophisticated details. Engineers without cryptographic or security knowledge might find them hard to understand, making the industry-level implementation even more challenging.

In this work, we investigate a concise and engineering-friendly solution for secure multi-party query processing. The design goals are three-fold. First, to make our system easy

to understand, we borrow the abstractions about tables and queries from traditional DBMS's. Clients directly run SQL queries to retrieve records and aggregate values from multiple data providers, as if running queries on large tables that are horizontally split. Second, our system is deployment-friendly and does not require non-colluding servers or hardware enclaves. Such a design is better matched with the current cloud infrastructure and provides potential opportunities for scalable distributed deployment. Last, the system should be efficient enough to solve real-world problems. By taking advantage of the recent development of RLWE-based homomorphic encryption schemes and applying our proposed optimizations, we reduce both the computation and communication costs of running secure query processing.

To achieve the goals above, we design and implement a prototype named PrivShare. The complete solution includes server-side daemon services that provide query endpoints and the client-side command-line interface (CLI) that allows running SQL queries on required data. Note that our system is different from encrypted database like CryptDB [21], covering different real-world applications. In our setting, data providers can be varied from financial organizations to medical institutions that hold their secret data. The client run queries over existing data sources instead of creating or managing their private databases.

In summary, we make the following contributions:

- We define the concise abstraction of secure multi-party query processing which covers many real-world applications. We also specify its security properties and discuss its threat model.
- We design the first one-round homomorphic encryption-based query processing protocol over multiple data sources and develop a prototype system PrivShare. It provides strong confidentiality guarantees for both the client and the data provider. It supports rich predicates and common data types in SQL to be expressive. It is efficient enough for building practical solutions and naturally support scaling-out. It is easy to use and does not require non-colluding servers or TEE.
- We develop optimization techniques for efficient equality check with grouped binary matching and range query with decomposition. By taking advantages of the plain text on the server-side, we make the equality check of two 32-bit integers 10× faster compared with [15].
- By providing micro-benchmarks and running secure queries on synthetic workload, we evaluate our prototype system and demonstrate its efficiency for applications in the real world. Our complete implementation will be available on GitHub.

The rest of the paper is organized as follows. In Section 2 we give an overview of the PrivShare abstraction by explaining the workflow, the query model, and the threat model. Then we describe the protocol details for supporting secure query processing in Section 3. In Section 4, we discuss implementation details. We evaluate the system and present micro-benchmarks and case studies in Section 5. We compare related work in Section 5.3 and conclude in Section 6.

2 System Overview and Threat Model

PrivShare includes a client that allows the user to run queries, and many server-side daemon services that provide a query interface on behalf of the data providers. The workflow of PrivShare consists of two phases: a setup (offline) phase and an online phase. In the setup phase, all participants start their system using the same configuration file that contains table schemas and security parameters. Then data providers load data and start a web service. The client also initializes its security context and analyzes the table schemas for later encryption of the queries.

Figure 1 illustrates the online phase of PrivShare. The client uses a previously generated key to encrypt the query, randomly chooses a provider as the master provider, and sends the query to the daemon service from the master provider (fig. 1.①). That service then obliviously processes the query to get an encrypted result. It also forwards the query to other providers (fig. 1.②), obliviously merges the encrypted result received from them (fig. 1.③) with its own result before sending the merged result back to the client (fig. 1.④). The client then decrypts the result to finish this query.

2.1 Security Definition

We consider a set of parties (a client and servers) who want to perform PrivShare. At the end of the protocl, the client only learns the query result on the parties' inputs while servers learn nothing. In the real-world execution, the parties often run protocol in the presence of an adversary who corrupts a subset of the parties. In the ideal execution, the parties interact with a trusted party that executes PrivShare in the presence of a simulator that corrupts the same subset of parties.

There are two classical security models. In the colluding model, we consider a single monolithic adversary that captures the possibility of collusion between the dishonest parties. The protocol is secure if the joint distribution of those views can be simulated. In the non-colluding model, we consider independent adversaries, each captures the view of each independent dishonest party. The protocol is secure if the individual distribution of each view can be simulated.

There are also two adversarial models. In the semi-honest (or honest-but-curious) model, the adversary is assumed to follow the protocol, but attempts to obtain extra information from the execution transcript. In the malicious model, the adversary may follow any arbitrary strategy.

We refer reader to [18] for additional discussion of formal security definition. Our scheme PrivShare provides a strong security guarantee in the malicious setting. The PrivShare

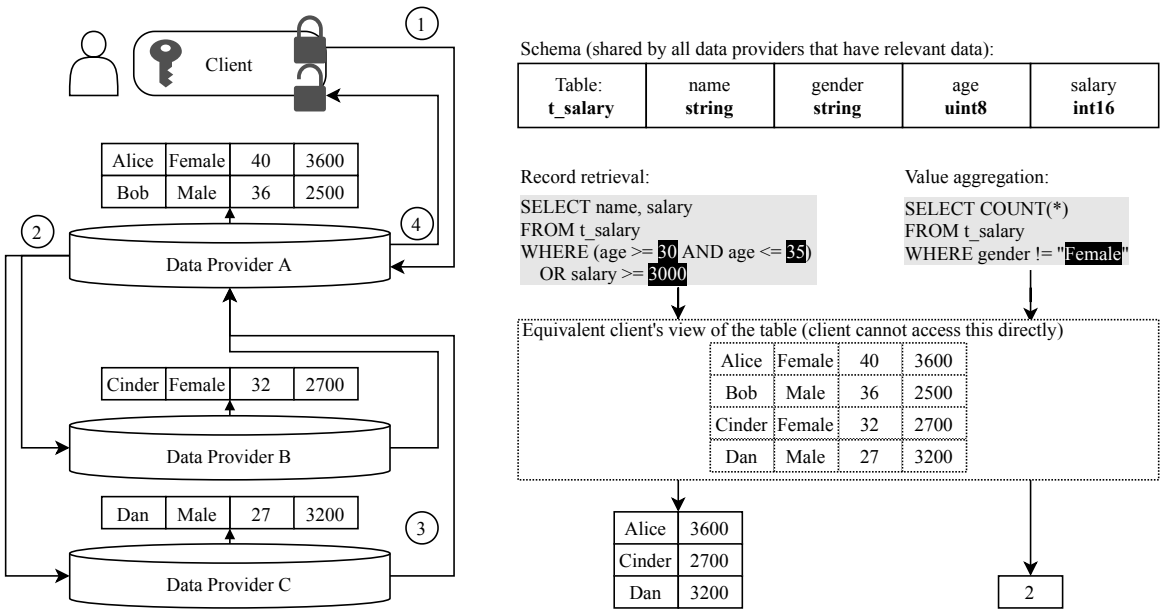

Figure 1: PrivShare overview. Clients can run queries to retrieve records or get aggregation results from three different data providers' query services for their own sensitive data. The SQL details in black background are hidden from all data providers.

is also secure against a colluding coalition between data providers, or a colluding coalition between clients.

2.2 Query model

PrivShare uses a similar data model to other relational databases (e.g. MySQL). In the multi-party setting, the client and all data providers use the same schema. While different data providers host their own sensitive records, the execution result will be as if the query is running on a combination of tables from all data providers. Note that the client does not have direct access to this concatenated table, which leaves the possibility for policy checkers discussed later in this section.

While PrivShare can support different types of database queries, it is specially optimized to perform two common types of queries: record retrieval and aggregation. In record retrieval operations, the client obliviously specifies the required columns and retrieves a small set of records that matches the given predicates. In record aggregation, the client wishes to compute an aggregation function (e.g., a sum or a count) on those matching records and only receives the aggregation value. We target to support rich predicate expressions, covering boolean formulas, negations, and range queries for different data types such as boolean, signed and unsigned integer, string, and enum.

While an aggregation often covers many data records, we consider the application settings where the record retrieval in each query are limited to only a small fraction of the databases, which protects the original data. If the application allows the client access to a large fraction of the original records, the client is able to learn the entire dataset of the data providers by submitting a few queries.

PrivShare targets to hide sensitive query parameters instead of the whole SQL statement. As shown in Figure 1, the values in the two queries' predicates with a black background are hidden from all the data providers during the query processes. The reason is that the structure of the query helps with policy checking and helps with providing efficient solutions without losing actual sensitive information. First, with the revealed query template, the server can implement a separate policy checker to have fine-grained control over the query types. For example, it can allow queries only for aggregation values of certain columns and rejects all queries that retrieve original records. This type of constraint on the query is helpful in many scenarios to better protect the privacy of the data provider. Second, in terms of the computation cost, hiding the complete statement requires the protocol to be as slow as the most time-consuming query types so that the side channel of execution timing does not leak information, which is computationally inefficient. It is hard to even define what the most time-consuming queries are considering the expressiveness of the predicates. Third, in terms of the communication cost, without knowing the query type, the encrypted query result needs to be as large as the retrieval of many records, which is communication inefficient. Although the number of retrieved records is limited, this would still add a large overhead to aggregation operations.

2.3 Threat Model

We now describe how our system addresses four potential threats from both curious clients and providers in Figure 2. We also specify uncovered threats and explain the reason. Last, we discuss the malicious setting and side-channel attacks. We

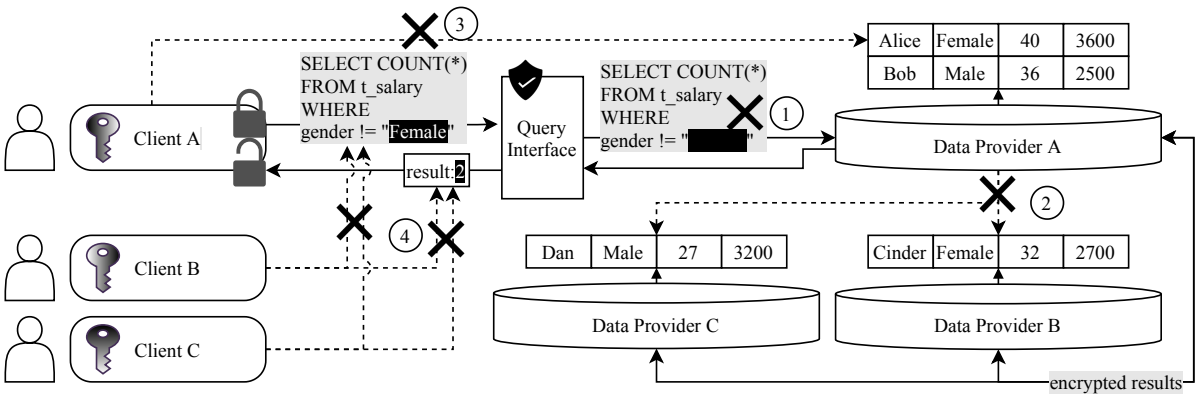

Figure 2: Threat model of PrivShare. The solid lines describe the information flow specified by the secure protocol, while the dotted lines represent illegal access.

assume an authenticated secure channel between each pair of verified participants (e.g., with TLS).

PrivShare hides the client's access patterns from curious data providers. As illustrated in fig. 2.①, no data provider can learn the sensitive parameters in the encrypted query or deduce which data records are selected by those parameters under the standard cryptographic assumptions.

Considering data providers might not trust each other with their sensitive data, PrivShare also prevents one provider from accessing original records from other providers (fig. 2.②). The master service can only merge query results that are securely encrypted under the client's public key.

On the other side, PrivShare also limits the access of the client to only content related to the query (fig. 2.③). For example, if the query type is aggregation, the client can only see the aggregated result without any original records. If the query type is retrieval, the system will not reveal records that do not match the predicates. This is similar to the privacy guarantees in symmetric PIR. Combined with an optional policy checker, such property will prevent curious clients from getting more data than authorized.

In the previous discussion we only mentioned one client, PrivShare also supports multiple clients. Different clients use different keys for their query so that one client cannot decrypt the query and its result of another client (fig. 2.④). In a restricted setting where a client generates a fresh public-private key for each query, even if the private key of a query is exposed, only the privacy of that query will be affected.

Although PrivShare protects the privacy of both the client and the data providers against the previous four types of threats, the following three scenarios are not captured by our threat model and are out of scope for our problem definition and consideration.

1. If the data provider decides to modify its own data, the client will be unable to detect such modifications because the data providers are responsible for the authenticity of their own data.
2. If one data provider decides to collude with the client, that data provider will have client's SQL access to the other provider's data which violates fig. 2.② The client functionality causes this leakage and there is no way to prevent it without stopping the service of the system. However, even in this case, the colluded data provider still cannot learn everything of the other provider's data because of the protection introduced by fig. 2.③.
3. The randomly selected master data provider not only helps with efficient distribution of the query, but also can potentially hides the source of data from the client. If the client colludes with the master data provider, it could learn which data provider provided a retrieved data record or partial aggregation result for a specific party. As a solution to this, some data providers can merge the query results first before sending back to the master that they do not trust or use a solution similar to [8] to enable anonymous communication. However, this simple extension to mediate different interests between data providers is orthogonal to the proposed approach in the paper and is beyond the scope of this paper.

Our system provide the above privacy guarantees even in the malicious setting. Since the server-side procedure is completely under homormorphic encryption, a malicious provider cannot cause any harm other than returning the invalid results. Since the query template is in plaintext, the client cannot bypass the policy checker to retrieve more data than suggested in the query. We discuss more the security proof of PrivShare in Section 3.6.

To prevent timing analysis based side-channel attacks, our query processing protocol guarantees that the execution procedure is independent of any sensitive parameters. However, the server-side execution time does leak information about table size. One solution is to pad the execution time to a fixed value and wait before sending back the encrypted results.

3 Query Processing

In this section, we describe the secure query processing protocol in PrivShare. Our protocol is concise because unlike prior works [4,19,23,24], it only requires one round of communication between the client and the randomly selected master data provider, which is the same as the normal distributed DBMS. The client uses its secure query planner to generate an encrypted query from a given SQL statement, and then sends this encrypted query to the master data provider. The service from the master data provider executes this encrypted query on its own data to get the encrypted result. It also forwards the query to other data providers and merges all encrypted results. Finally, it sends this result back to the client.

In the following subsections, we first cover the preliminaries of our protocol. Then we specify basic building blocks to support the protocol, including two core techniques: grouped binary matching and range decomposition. After that, we describe how the client generates a secure execution tree with its secure query planner and explain how data providers can use that tree to execute the query. We also describe how the master data provider merges the result when there are multiple data providers.

3.1 Existing Cryptographic Tools

We review the basics of Homomorphic Encryption and Private Information Retrieval used in PrivShare.

3.1.1 Homomorphic Encryption

A Fully Homomorphic Encryption (FHE) scheme is a powerful primitive that allows for (mostly) arbitrary functions to be computed over encrypted values. A homomorphic scheme is defined as a set of 4 algorithms:

- KeyGen(λ) $\rightarrow (pk, sk)$: takes a security parameter λ and generates a public and secret key pair (pk, sk).
- $\text{Enc}_{pk}(m) \rightarrow c$: takes a public key pk and a plaintext m from the plaintext space M, and outputs a ciphertext $c \in C$, where C is the ciphertext space.
- $\text{Dec}_{sk}(c) \rightarrow m$: takes a secret key sk and a ciphertext $c \in C$, and outputs a plaintext $m \in M$. Note that if the input secret key sk corresponds to the public key pk, $m = \text{Dec}_{sk}\big(\text{Enc}_{pk}(m)\big)$. Otherwise, it is unable to decrypt the ciphertext, thus the decryption outputs $\perp$.
- $\text{Eval}_{pk}(F, c) \rightarrow c'$: takes a public key pk, a ciphertext c, and a function F, and outputs a ciphertext c'. Note that for any function F, a plaintext m, and a corresponding public and secret key pair (pk, sk), if $c = \text{Enc}_{pk}(m)$, then $\text{Dec}_{sk}(c') = F(m)$.

While there are many HE schemes (e.g., BGV [6], CKKS [9]), PrivShare is implemented using BFV [14] as we assume that PrivShare's schema only consists of integer type. To handle float number using PrivShare, one can scale up the number to an integer by shift-and-truncate operator.

```
function CLIENT_QUERY(pk, record_index)
    for i = 0 to n − 1
        c[i] = Enc(pk, i == record_index ? 1 : 0)
    return query = c[0]...c[n−1]

function SERVER_ANSWER(query = c[0]...c[n−1], records)
    for i = 0 to n − 1
        a[i] = records[i] ∗ c[i]
    return sum(a[0]...a[n−1])

function CLIENT_OUTPUT(sk, encrypted_sum)
    return Dec(sk, encrypted_sum)
```

Figure 3: XPIR using a HE scheme Gen, Enc, Dec with key pair (pk, sk). The client runs CLIENT_QUERY and CLIENT_OUTPUT. The server runs SERVER_ANSWER. The database has n records.

BFV is a lattice-based scheme and depends on the hardness of Ring Learning with Errors (RLWE), which is the regular Learning with Errors problem specifically for polynomial rings. BFV is a leveled FHE scheme whose parameters determine the allowed multiplicative depth (number of sequential multiplications). This is because each operation incurs noise, as RLWE hardness is based on adding noise to a lattice point. Therefore, it is important to properly select parameters that allow enough noise budget. We explicitly discuss HE parameters used in PrivShare in Section 4.

An important optimization of FHE computation when using BFV is batch encoding based on the Chinese Remainder Theorem (CRT). For a polynomial of degree N where the plaintext modulus parameter is T, CRT batching allows for storing N values modulo T. As long as T is prime and congruent to 1 mod $2N$, the plaintext can be seen as a matrix of size 2 by $\frac{N}{2}$, and operations on the matrix can benefit from SIMD instruction parallelism.

3.1.2 Private Information Retrieval

Private information retrieval (PIR) allows a client to query information from one or multiple servers in a such way that the servers do not know which information the client retrieved. Most PIR protocols can be grouped into information-theoretic PIRs (IT-PIRs) and computational PIRs (CPIRs). The security guarantees for IT-PIR protocols typically rely on multiple non-colluding servers, while CPIR protocols rely on the computational limits of servers.

XPIR [1,7,16] is a CPIR scheme. This scheme does not require database replication, but this comes at the expense of either computation or communication costs. In the original PIR scheme [7,16], a query is a vector with n (number of elements in the database) ciphertexts, where each ciphertext is an encryption of an indicator value of whether the index is the desired record. We call these ciphertexts the indicator vector, and will continue to use this term to discuss our proposed protocol. Figure 3 describes the HE-based XPIR protocol. Recent PIR [2,3,12] reduces communication cost to logarithmic in

the database size while requiring an extra computational cost on the server's side, reflecting a trade-off between bandwidth and computation costs, as discussed in [2].

3.1.3 Boolean Formulas

Supporting AND, OR, and NOT operations between indicator vectors is equivalent to performing these operations element-wise for 1's and 0's under the encryption. Thus, we can perform these operations using additive and multiplicative homomorphic encryption as follows:

$$\begin{aligned} \text{AND}(ind_a, ind_b) &:= ind_a * ind_b \\ \text{NOT}(ind) &:= 1 - ind \\ \text{OR}(ind_a, ind_b) &:= \text{NOT}(\text{AND}(\text{NOT}(ind_a), \text{NOT}(ind_b))) \\ &= ind_a + ind_b - ind_a * ind_b \end{aligned}.$$

Layered combinations of boolean operations in the predicates can be performed by processing the indicators with the same sequence. However, we do optimize the actual execution sequence during query generation. Details are discussed in Section 3.3.

3.2 Our Efficient Building Blocks

Compared with the XPIR protocol, one major challenge of our protocol is to generate the indicator vectors from both the encrypted predicates from the client, and the plaintext records from the servers. Below we discuss how we support equality check and range queries in query predicates based on FHE.

3.2.1 Equality Check with Grouped Binary Matching

Consider a simplified scenario: the client has an 8-bit unsigned integer v for the column attribute c of the data provider's database. The data provider needs to generate an indicator vector ind that has the same length N as the database. Similar to XPIR, all records $r[i], \forall i \in \{1,...,N\}$, which have a cell $r[i][c]$ equal to v should have an entry of encrypted 1's in that vector and the rest are encrypted 0's. Mathematically, given a public key pk of the client, for all $i \in \{1,...,N\}$

$$ind[i] := \begin{cases} Enc_{pk}(1) & r[i][c] = v \\ Enc_{pk}(0) & otherwise \end{cases}$$

The client can later use that indicator vector on the server-side to retrieve records or run aggregation computation (like a sum). More details are discussed in Section 3.4.

Now the challenge is how the data provider can generate the indicator vector ind given the column attribute c, the record r, and an encrypted v. Considering making basic operations efficient, to avoid expensive primitives like garbled circuit [5, 25], we use a naive method which requires that the client generates $2^8 = 256$ ciphertexts m covering all possible values of that 8-bit unsigned integer v in advance, where

$$m[j] := \begin{cases} Enc_{pk}(1) & j = v \\ Enc_{pk}(0) & j \neq v, 0 \leq j \leq 255 \end{cases}.$$

Then the data provider can generate ind by looping through its record and setting $ind[i] := m[r[i][c]], \forall i \in \{1,...,N\}$. Note that this only works efficient when there is a limited number of possible values.

To allow equality check not only for 8-bit integers but also for normal 32-bit or 64-bit integers, we present a technique called grouped binary matching, which decomposes the task of large equality check into multiple 8-bit equality checks and some AND operations. By splitting the binary representation of the original 32-bit value into four 8-bit groups, we now have four 8-bit integers. The equality of the 32-bit integer holds if and only if the equality for all four 8-bit integers holds. Using 8-bit binary representation, we have:

$$x = v \Leftrightarrow \text{AND} \begin{cases} \text{AND} \begin{cases} x \equiv v \pmod{2^8} \\ \lfloor \frac{x}{2^8} \rfloor \equiv \lfloor \frac{v}{2^8} \rfloor \pmod{2^8} \end{cases} \\ \text{AND} \begin{cases} \lfloor \frac{x}{2^{16}} \rfloor \equiv \lfloor \frac{v}{2^{16}} \rfloor \pmod{2^8} \\ \lfloor \frac{x}{2^{24}} \rfloor \equiv \lfloor \frac{v}{2^{24}} \rfloor \pmod{2^8} \end{cases} \end{cases}.$$

Since we have already supported both 8-bit equality check and the boolean operation of AND, now we support 32-bit equality check. The procedure of 64-bit equality check is similar and requires one additional layer. To support string matching, we use cryptographic hashing function to convert the string into 32-bit or 64-it integer values.

Note that both processing and sending 256 ciphertexts for an 8-bit equality check would significantly slow down the protocol. To avoid this cost, we combine the CRT-based batching with the rotation operation [22] to batch all 256 0's and 1's into one single ciphertext. Furthermore, for the whole query there are many instances of such building block. Under our default security parameters, we can support 64 of such 8-bit building block with only one ciphertext, resulting in a small query size that is communication-efficient. Servers later use the batched query ciphertext by rotating it based on the data record value and masking out the irrelevant locations, which are both efficient operations if implemented carefully.

Among all power-of-two candidates that properly aligned with default integer size, we choose 8 to be the size of the basic building block considering the following two reasons. (1) It should be as large as possible to reduce both the multiplication number and depth. For example, using 4-bit equality check results in 3 layers of 7 multiplications in total, which doubles the computation cost. (2) The 0's and 1's for the query parameter should fit in one ciphertext. The 16-bit equality check requires 4 ciphertexts for each secure parameter (around 7MB) which is communication-inefficient. The rotations for all 2^{16} possible locations also burden the computation. Therefore, an 8-bit equality check yields the lowest communication and computation cost.

Range Query	#8-bit range	#8-bit eq	#AND	#OR	#layer	#layer(opt)
8-bit	1	0	0	0	1	1
16-bit	3	2	2	2	4	4
32-bit	7	6	6	6	7	7
64-bit	15	14	14	14	15	9

Table 1: Operation number required for range decomposition. The optimized multiplication layer is listed in #layer(opt).

3.2.2 Range Query with Range Decomposition

The range queries are challenging because they require comparison between encrypted boundaries and the value in the data record. We first consider a simplified scenario where the client has two 8-bit unsigned integers v_l and v_r and wants to generate an indicator vector that satisfies

$$ind[i] := \begin{cases} Enc_{pk}(1) & v_l \le r[i][c] \le v_r \\ Enc_{pk}(0) & otherwise \end{cases}.$$

The solution is similar to 8-bit equality check described in the section above. The client computes 265 ciphertexts using their public keys:

$$m[j] := \begin{cases} Enc_{pk}(1) & v_l \le j \le v_r \\ Enc_{pk}(0) & otherwise, 0 \le j \le 255 \end{cases}$$

and sends them to the data provider. Then the data provider sets $ind[i] := m[r[i][c]], \forall i \in \{1,...,n\}$, to get the indicator vector without any homomorphic operations.

The challenge is to support range queries for 32-bit/64-bit integers. We omit the discussion for 64-bit integers since it is similar to the solution for 32-bit integers.

Inspired idea from segment tree data structure [11] and our proposed grouped binary matching, we decompose one 32-bit range operation into a combination of multiple 8-bit range operations, multiple 8-bit equality checks, and some AND and OR boolean operations. We specify the complete decomposition in Figure 4.

By the decomposition of long ranges, we convert one 32-bit range operation into 7 8-bit range operations, 6 8-bit equality checks, 6 AND operations, and 6 OR operations. While this works for unsigned integers, for signed integers, we simply apply a range-preserving conversion to remove the sign by adding 0×80000000. We list the operation numbers required for $\{8, 16, 32, 64\}$-bit range decomposition in Table 1. Note that for 64-bit range queries, considering the limited noise budget, we can rebalance the operations to reduce the multiplication level. More details about rebalancing the operations are discussed in Section 3.3.

3.3 PrivShare Query Generation

Having all the necessary building blocks for supporting rich predicates, a secure query planner is responsible for encrypting the query at the core of the client implementation. The planner takes the SQL statement as the input and outputs a

$$v_l \le x \le v_r \Leftrightarrow$$
$$\text{OR}\begin{cases} \text{OR}\begin{cases} \text{AND}\begin{cases} \lfloor \frac{x}{2^{24}} \rfloor = \lfloor \frac{v_l}{2^{24}} \rfloor \\ \text{OR}\begin{cases} \lfloor \frac{v_l}{2^{16}} \rfloor + 1 \le \lfloor \frac{x}{2^{16}} \rfloor \\ \text{AND}\begin{cases} \lfloor \frac{x}{2^{16}} \rfloor = \lfloor \frac{v_l}{2^{16}} \rfloor \\ \text{OR}\begin{cases} \lfloor \frac{v_l}{2^{8}} \rfloor + 1 \le \lfloor \frac{x}{2^{8}} \rfloor \\ \text{AND}\begin{cases} \lfloor \frac{x}{2^{8}} \rfloor = \lfloor \frac{v_l}{2^{8}} \rfloor \\ v_l \le x \le 255 \end{cases} \end{cases} \end{cases} \end{cases} \end{cases} \\ \text{AND}\begin{cases} \lfloor \frac{x}{2^{24}} \rfloor = \lfloor \frac{v_r}{2^{24}} \rfloor \\ \text{OR}\begin{cases} \lfloor \frac{x}{2^{16}} \rfloor \le \lfloor \frac{v_r}{2^{16}} \rfloor - 1 \\ \text{AND}\begin{cases} \lfloor \frac{x}{2^{16}} \rfloor = \lfloor \frac{v_r}{2^{16}} \rfloor \\ \text{OR}\begin{cases} \lfloor \frac{x}{2^{8}} \rfloor \le \lfloor \frac{v_r}{2^{8}} \rfloor - 1 \\ \text{AND}\begin{cases} \lfloor \frac{x}{2^{8}} \rfloor = \lfloor \frac{v_r}{2^{8}} \rfloor \\ 0 \le x \le v_r \end{cases} \end{cases} \end{cases} \end{cases} \end{cases} \end{cases} \\ \lfloor \frac{v_l}{2^{24}} \rfloor + 1 \le \lfloor \frac{x}{2^{24}} \rfloor \le \lfloor \frac{v_r}{2^{24}} \rfloor - 1 \end{cases}$$

Figure 4: Range decomposition (equality checks & range operations are under $\bmod 2^8$).

tree-like data structure called a secure execution tree (Figure 5), where all sensitive query parameters are already encrypted. Here, we describe the details of the generation and encryption procedure.

The secure query planner parses the SQL statement to get the query type as well as the tree of predicates. It then generates a tree-structured execution plan named secure execution tree where each node except for the root in the tree corresponds to an indicator generation operation. Before sending it to a randomly selected master data provider, that tree needs to be revised through 4 stages: pre-expansion optimization, expansion, post-expansion optimization, and encryption.

In the pre-expansion optimization stage, the planner analyzes the high-level equality checks and range operations and tries to reduce the total number of them. For equality checks under the same AND operation, the planner reduces the number of operations by replacing them with a "multiple equality check" operation. Concretely, instead of comparing the original concerned columns one by one, the new operation performs the equality check on the value generated by performing PRF on a concatenation of all concerned columns. The planner also analyzes the range operations to use as few range operations as possible. For example, if $age <= a$ or $age >= b$ appears in the predicates, the planner revises it to be $not(range(a,b))$ to save one range operation. Note that the procedure is independent of the query parameters. So even if $range(a,b)$ does not make sense ($a > b$), we

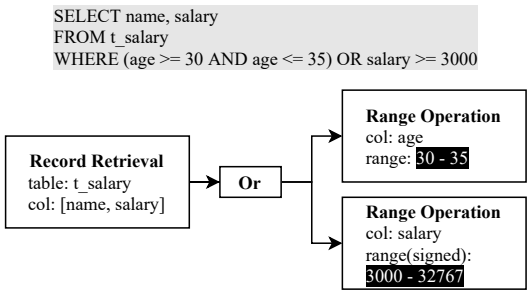

Figure 5: An example of secure execution tree. In the example, the tree was not expanded and encrypted for better illustration.

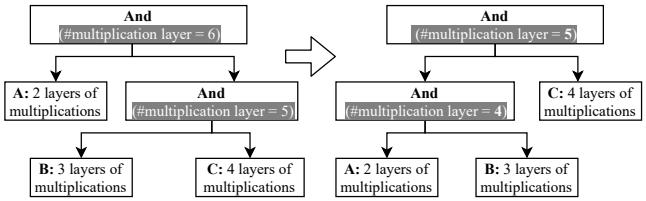

Figure 6: The execution tree rebalancing procedure for a subtree. A, B, and C are children subtrees that have been rebalanced.

still keep it as is and generate $2^8 = 256$ ciphertexts of zeros in the later encryption stage. In this way, the structure of the secure execution tree leaks no information about the sensitive query parameters.

In the expansion stage, the planner loads the table schema from the configuration file and expand operations for long data types into basic 8-bit operations. It uses the grouped binary matching and range decomposition techniques discussed in the previous section. To simplify the execution, it converts all signed predicates, replaces OR operations with AND and NOT operations, and removes pairs of consequent NOT operations. After this stage, the secure execution tree will only contain AND, NOT, and basic unsigned 8-bit operations.

In the post-expansion optimization stage, the planner rebalances the secure execution tree to save noise budgets for HE. As illustrated in Figure 6, it uses a reversed depth-first searching order to greedily rotate the subtree according to the number of multiplications in each child. In addition, it also contains a set of rules to match and convert specific tree structures (e.g. range decomposition) for better optimization.

Finally, in the encryption stage, the planner replaces plaintext values in the leaf node with grouped ciphertexts for 1's and 0's. After that, the clients send the revised secure execution tree along with the ciphertext payload to the master data provider for execution. We discuss the detailed structure for the ciphertext payload in Section 4.

3.4 PrivShare Query Execution

Once the data provider receives secure execution tree and related ciphertexts, it loads the corresponding keys and starts to generate the indicator vectors by iterating through the tree in a depth-first searching order. After accessing all the nodes converted from predicates, the data provider gets the final indicator vector that suggests which records in the database should be selected. The final indicator vector is quite similar to the one in XPIR protocol. However, the main difference is that it might contain multiple entries with 1's encryptions.

3.4.1 Record Aggregation

Record aggregation is relatively straightforward. The COUNT operation is simply a sum of all entries of the indicator vector itself. The SUM operation is a dot product of the indicator vector and values in the original selected column. Combining the result of COUNT and SUM queries can support AVG, the average aggregation. We do identify this as an extra leakage. But considering the expensiveness of homomorphic division, we think in most cases this is a good trade-off.

PrivShare can be also extended to support comparison aggregation operators, MAX and MIN. With the COUNT operator, we support MAX and MIN by replacing it with multiple queries that run the binary search on the existence of column value. For example, assuming column c contains 8-bit unsigned integers, the query "SELECT $MAX(c)$ FROM t" can be replaced by

$$\text{SELECT } COUNT(*) \text{ FROM } t \text{ WHERE } c >= 128,$$
$$\text{SELECT } COUNT(*) \text{ FROM } t \text{ WHERE } c >= 64 \ldots$$

Then we can find the maximum. However, we note that this extension introduces certain leakage from the counting result, and requires $log(b)$ rounds of queries, where b is the bit width of the column data type.

3.4.2 Record Retrieval

Unlike aggregation, record retrieval is non-trivial. We need to retrieve multiple records, but homomorphically adding up all the multiplication results between indicator and column values like XPIR no longer works. On the other side, simply sending back all ciphertexts is privacy-preserving but not realistic, as it leads to communication that has size linear to the database. Such a communication cost is unacceptable.

To deal with this challenge, we take advantage of the assumption that only a small fraction of the record will be retrieved, and propose a simple technique called collision-tolerant folding, inspired by [10]. Instead of adding up all n records into 1 final value, we equally separate the original records into s slots and calculate the sum within each slot. If no slot contains more than 1 selected record, the retrieval will be successful. By allowing a certain degree of collisions, we can significantly raise the success rate.

We first discuss the basic solution which does not tolerate collisions. The first modification is to shuffle the data record for each query before generating any indicators. It does not affect the query result but makes it possible to rerun the query once the collision happens. After getting the final indicator, we multiply the selected columns in records by the final indicator vector to set unrelated records to zero. We group the

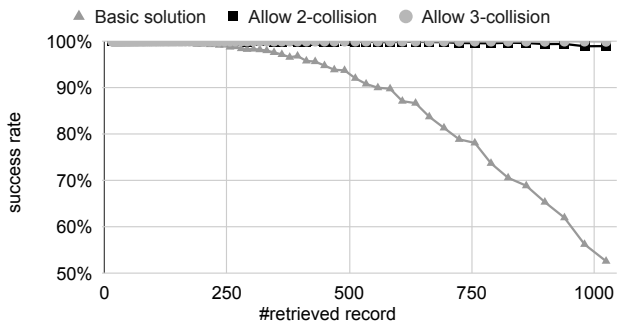

Figure 7: Success rate of record retrieval for different degree of collision tolerance.

records according to the remainder of dividing by slot number and add up all records within the same group. We add a checksum to the record before putting them into the ciphertext so that the client knows the existence of collision during the decryption phase. For the basic solution, a successful retrieval require no collision at all.

To make the basic solution collision-tolerant, we change the data representation. In the basic solution, if two non-zero selected records are added to the same slot, the result will be unrecoverable. Assuming two records are represented by x and y, there is no way to recover them given the sum $x+y$. However, if we represent records as x, x^2 and y, y^2, then given $x+y, x^2+y^2$, we can simply recover two original values by

$$x, y = \frac{(x+y) \pm \sqrt{2(x^2+y^2)-(x+y)^2}}{2}.$$

Although it approximately triples the size of the ciphertexts, the success rate of retrieving same number of records can be significantly higher, which is generally a good trade-off. In this case, the retrieval is successful if at most two records occupy the same slot. It is also possible to allow higher degrees of collisions, but the record number cannot amend for the required extra ciphertext space.

Figure 7 illustrates the success rate of record retrieval with or without our optimization. For one ciphertext that can batch 16,384 records, we have the following two observations. (1) When retrieving a small set of records, for example 128 records, the 2-collision-tolerate solution has a success rate over 99.999% which is much more reliable than the basic solution 99.88%. (2) When retrieving more record (1024 records), the basic solution fails in about half of the cases while the optimized version is still usable with around 99.01% success rate. We note that user can increase the success rate by rerun the query multiple times or explicitly require more ciphertexts, however these solutions introduce certain information leakage about the secure result and are thus not recommended.

3.5 Result Merging

We focused primarily on one master data provider in the previous discussion. In this subsection, we describe the merging process after the master data provider receives encrypted results from other data providers for different query types. The encrypted result will be one ciphertext for record aggregation, and a list (matrix if batching is used) of ciphertexts for record retrieval. Details of the data layout will be explained together with batching optimization in Section 4.

Note that all results are encrypted under the same public key generated by the client. For record aggregation, the master data provider can directly perform necessary aggregation operations on results from other providers. For record retrieval, because the encrypted results use the same slot and column structure, the master data provider continues the folding procedure by aligning the slots and directly adding them up. Since the total number of retrieved records is limited and the shuffling procedure is random, the previous discussion on success rate still holds.

Once the master data provider merges the results according to the query type, it sends the final encrypted result back to the client for decryption, which finishes the whole secure query processing procedure.

3.6 PrivShare Security Discussion

The security of our PrivShare is heavily based on the security of the underlying homomorphic encryption scheme. Denote $\mathcal{C}$ as a coalition of corrupt parties. We exhibit simulators Sim for simulating $\mathcal{C}$. Sim simulates the view of corrupt parties, which consists of $\mathcal{C}$'s randomness, input, output, and received messages. The view of corrupt clients is independent of each other since each client generates and uses their own HE key to encrypt and submit their query. Since our PrivShare does not allow a collusion between corrupt clients and servers, the view of corrupt servers is also independent of that of other corrupt servers as the corrupt servers process the client's query under an encrypted form. Therefore, we only consider a case where $\mathcal{C}$ contains either a corrupt client or a corrupt server.

PrivShare is a one-round protocol. The view of the corrupt client $\mathcal{C}$ only consists of $\mathcal{C}$'s input, output, and the encryption of the query result that is obtained from the master data provider. Consequently, the security of PrivShare in the presence of the malicious client follows in a straightforward way from the security of HE.

Sim simulates the view of corrupt server $\mathcal{C}$ by calling simulators of the query execution (generating the indicator vector based on equality check or range decomposition), the record aggregation/retrieval, the merging process. The produced transcripts from the cryptographic building blocks (e.g. equality check, range decomposition) described in Section 3.2 are indistinguishable from the real execution because pseudo-randomness guarantees of the underlying HE. Therefore, all simulations HE except the record retrieval go through. In the

record retrieval processing, PrivShare applies the collision-tolerant folding technique to every record x in the database. $\mathcal{C}$ might deviate from this execution by using x and $\bar{x}^2$ to present a record x, where $\bar{x} \neq x$. This malicious action results in an incorrect query result obtained by the honest client. However, one can simulate this event by considering that $\mathcal{C}$ changes their input. Following the real-ideal paradigm [18], PrivShare is secure.

Theorem 1. *The construction of PrivShare securely implements the query processing on multiple sources in the presence of any number of malicious clients or malicious data providers, given the secure homomorphic encryption scheme.*

4 Implementation

Based on our protocol discussed in Section 3, we implemented a prototype of PrivShare with C++. Extended from our in-memory relational database, we implemented both a client-side CLI and a server-side web service. We use Microsoft SEAL [22] for homomorphic encryption. For security parameters, we use the BFV scheme with a 2^{14}-degree polynomial and CRT-batching with an 18-bit plaintext modulus, which provides a noise budget that allows at most 11 layers of multiplications. While this is enough for most of the queries, we also have a set of backup security parameters that can be more efficient for simpler queries (2^{13}-degree, 17-bit) or supports at most 29 layers (2^{15}-degree, 12-bit) for complicate queries.

Correctly using the CRT-batching mentioned in Section 3.1 to amortize the multiplication cost is critical for making the implementation efficient. This key observation motivates the protocol designs like using vector operations and having a secure execution tree. In the rest of this section, we explain how we utilize batching operations to amortize both communication and computation cost. We describe the ciphertext data layout and discuss how to compress vector ciphertexts with rotations.

As we mentioned earlier in Section 3.2.1, during the last encryption stage of query generation, instead of generating 256 ciphertexts for each basic building block node, the client creates multiple vectors with 256 entries each and batching the concatenation of them into as few ciphertexts as possible. This way, one ciphertext can cover $2^{14-8} = 64$ basic building block nodes. As a result, in most cases, one query only requires one ciphertext. Later, the data provider extracts the indicator according to the value in the data record by rotating this ciphertext to the correct location and masking it with plaintext multiplications.

While the aggregation operations do not need complicated batching, the data layout in record retrieval requires careful design. As shown in Figure 8, the ciphertext is separated internally to tolerate 2 collisions. Horizontally, multiple ciphertexts cover the complete bit representation of records in the slots, while vertically, more rows allow extra slots to retrieve more records.

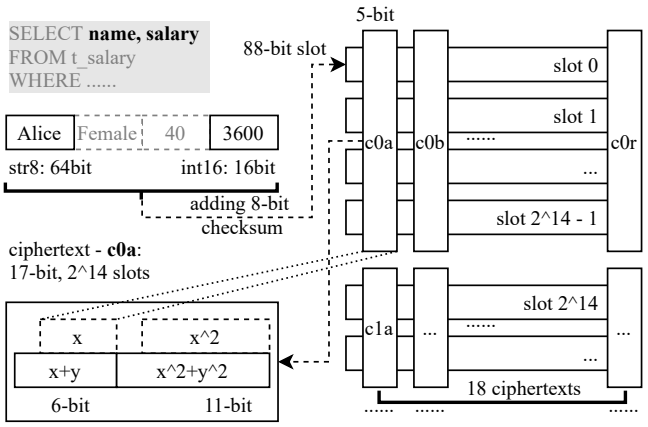

Figure 8: Data layout of ciphertext during record retrieval. We assume an 8-byte name field and a 16-bit salary field. With 8 bits of checksum for collision detection, the 88-bit sequence can be put into 18 ciphertexts that provides 90-bit slots.

The CRT-batching also simplifies the folding procedure. Instead of doing extra rotations inside ciphertext, the data provider only needs to add the complete ciphertexts for shuffled and masked data records. As long as the addition is horizontally aligned, it can execute the folding procedure correctly.

5 Evaluation

We evaluate the performance of PrivShare and we answer the following questions: (1) How long does it take to run normal operations on PrivShare? (2) Is it practical enough for supporting real-world applications? (3)How does it compare with other similar works?

Our evaluations below are based on simulations using our implementation of the protocol. All of the experiments used 2.3 GH-z 8-Core Intel Core i9 machine with 16GB of RAM. For a fair comparison, single-thread performance is reported in most cases. One can expect the processing throughput grows linearly as we increase the number of cores or hosts.

5.1 Micro-benchmark

Table 2 shows single-core micro-benchmarks for the time consumption of different operations used in PrivShare. The 8-bit building block operations are the most time-consuming ones since they involve many rotations (≈59%) and plaintext multiplications (≈24%) for extracting the grouped encryptions of 1's and 0's in the ciphertext according to the values in the data records. However, it still takes less than one millisecond, meaning that on average, we can process more than 2,000 original records per second. Considering that the time to run the equality check on one plaintext record is around 0.181 µs, our 8-bit equality check is about 2500× slower than a no-protection implementation.

Operation		Amortized running time for each record (μs)
HE	Plaintext add	0.08
	Ciphertext add	0.01
	Plaintext mul	0.47
	Ciphertext mul	3.05
	Rotation	1.06
	Relinearization	1.10
Building block	8-bit equality check 8-bit range query	442.25
	Boolean - NOT	0.04
	Boolean - AND	4.12
	Boolean - OR	4.23
Composition	16-bit equality check	888.63
	16-bit range query	2227.97
	32-bit equality check	1781.37
	32-bit range query	5799.41

Table 2: Micro-benchmark for PrivShare.

Note that our solution can benefit directly from concurrency and is scalable by nature because an entity can always act as multiple data providers and split its table into shards. The master data provider sends the query to all other providers, who execute the query in a parallel fashion. It allows our system to support an even larger workload in a distributed setting. A parallel PrivShare implementation on a 32-core host can finish an aggregation with a 32-bit equality predicate for more than 1,000,000 records within 1 minute.

5.2 Evaluation on Real-world Application

Now, we evaluate our PrivShare on two applications: cyber threat intelligence gathering and financial fraud detection.

5.2.1 Collaboration on Cyber Threat Intelligence

One major dataset we used is focused on cyber threat intelligence (CTI) based threat detection. The overarching goal is to detect fraudulent or suspicious cryptocurrency transactions with CTI from multiple parties. Privacy is particularly important in this form of collaboration because while analysts should be able to run queries to identify suspicious addresses, they may not be cleared to know who or what is actually associated with the addresses. We anticipate that such analysis would require use of conditional clauses with aggregation.

To see whether PrivShare is practical for such applications, we investigate an existing use case: matching the suspicious wallet addresses. Existing industry solutions often use centralized infrastructure and hashing-based methods to hide the original records, which are vulnerable to brute-force attacks on the plaintext. As an alternative, PrivShare eliminates such vulnerabilities and naturally supports the distributed setting where all data providers can better protect their valuable cyber threat intelligence.

In our synthesized scenario, 1/4/10 data providers collaboratively host a service of suspicious wallet address identification that checks the existence of a given wallet address in a known wallet address list. We assume 10^5 records for all data providers in total and assume each data provider holds the same number of records. The analyst uses the client CLI of PrivShare to query "SELECT count(*) FROM t_cti WHERE wallet_address="xxx" ".

		1 server	4 servers	10 servers
#record per server		100000	25000	10000
Client	query generation (s)			0.01
	query size (MB)			1.74
Server	query execution (s)	198.80	57.17	28.34
	result size (MB)			1.74
	query merging (s)	0.00	0.92	2.06
Client	result decryption (s)	0.03	0.03	0.03
Offline	key generation (s)			4.80
	key size (MB)			212.97

Table 3: PrivShare's evaluation for aggregation query in CTI application. The offline phase consists of key generation. The online phase consists of the query generation/execution/merging, and result decryption.

The query execution time and communication payload size of PrivShare are listed in Table 3. In terms of the communication cost, both the secure query and the encrypted result are encrypted in one ciphertext that takes 1.7 MB. The security execution tree only takes a few KBs. Assuming 10 Mbps point-to-point network bandwidth between all nodes, the analysts should be able to receive and decrypt the final result within 37 seconds for the 10-server case. For cases with fewer servers, the running time can be further reduced with more optimizations like multi-threading in the implementation.

5.2.2 Financial Fraud Detection

To study how PrivShare supports real-world financial applications, we use a dataset generated from [17] with 595k records of bank payment to simulate a fraud detection system using data from multiple parties. While the CTI case demonstrates the performance for equality check in record aggregation, in the financial case we focus on record retrieval with rich predicates that contain range queries.

To be more specific, we assume that 4 banks decide to use PrivShare to enable a fraud detection data analytic service. Each of them agrees to provide limited access to either an aggregation or less than 1% original records in all its 149k records. PrivShare ensures the constraint by using a separate policy checker and limiting the number of retrieved ciphertexts. We consider a query, "SELECT age, category FROM t_finance WHERE fraud=1 AND amount>6000", that retrieves a detailed description of known fraud payment for better data insight. For example, for this specific query, the resulting 25 records help analysts easily deduce that all these large fraud payments belong to "es_travel" categories and most of them have a short account age. Since merchant, customer id, and specific payment amount are not involved in the query, the privacy of the data provider is protected.

The performance of the query is shown in Table 4. Com-

		4 servers
#record per server		148661
Client	query gen (s)	0.01
	query size (MB)	1.74
Server	query execution (s)	441.63
	result size (MB)	12.19
	query merging (s)	6.45
Client	result decryption (s)	0.23
	#result record	25

Table 4: PrivShare's evaluation for record retrieval in financial application in the online phase. The offline phase consists of key generation. Its performance is the same in Table 3.

pared with CTI example, there are two major differences. (1) As explained in Section 4, we need 7 ciphertexts for the 2-collision-tolerant encoding of 8-bit age field, 16-bit enumerate field, and 8-bit checksum required in the query. Not only the query result size becomes larger, all procedures using query results as the input are also more time-consuming. (2) The execution takes longer time because the 16-bit range query on the "amount" column is decomposed into many smaller building blocks. Note that we treat the float numbers in the original dataset as integers to save the required bit width.

From these two examples, we can see that the single-thread PrivShare can already support secure multi-party query processing with 10^5-level workloads within tens or a few hundreds of seconds. With its support for scaling-out in nature, PrivShare can easily support minute-level processing on millions of records with a few more cores or servers.

5.3 Comparison with Other Work

As the first system to process secure multi-party queries using homomorphic encryption, we have not found any existing work that targets the same set of security goals as PrivShare. Nonetheless, there are some secure MPC-based solutions [4, 20, 23] that also process multi-party queries and have broader functionalities (e.g. *JOIN*). But they only target a subset of our security goals, require multiple rounds of communication, and have limitations on the input size. For two-party secure query processing, there are works using homomorphic encryption [15], non-colluding server assumption [19, 24] or trusted execution environment [13, 26]. In this subsection, we compare both the security guarantees and the performance of those works with PrivShare.

SMCQL [4] focuses on hiding multiple data providers' inputs from each other, which is similar to our second security guarantee in fig. 2.②. However, it does not provide query obliviousness and does not protect the privacy of data providers from the client. It also has a query planner that translates plaintext queries into distributed secure MPC steps using both garbled circuits and ORAM. Those techniques allow SMCQL to run cross-table operations like *JOIN* which PrivShare currently does not support. However, they heavily limit the input size for each party. An 8-server SMCQL setup with each party supplying 650 records would result in a running time that is more than 10,000 seconds. In contrast, PrivShare supports a larger input size of hundreds of thousands of records and can finish the query processing within a few minutes even in a single thread mode. Conclave [23] improves the performance of SMCQL by using the selectively-trusted party assumption, allowing certain information leakage between data providers. However, such relaxation of security guarantee only works for a subset of the applications that PrivShare supports. Furthermore, it requires careful design to avoid critical leakage in real-world scenarios.

Senate [20] is a recent work that targets to support multi-party SQL queries with MPC techniques in the presence of malicious adversaries. Similar to SMCQL, it also focuses on protecting privacy between data providers instead of between data providers and the client. Their multi-way set union circuit primitive provides functionalities similar to the default table combination in PrivShare. In terms of performance, it also suffers from a limited input size. In addition, the multi-way set union building block cannot efficiently support an input size larger than 800 in a 16-party setting.

Splinter [24] is a recent system that uses function secret sharing to hide query parameters in a non-colluding server setting. Opaque [26] is designed on top of the distributed hardware enclaves, the threat model of it mainly focuses on root adversaries from the cloud provider, and mitigation of access pattern attack. Both of them are one to two orders of magnitude faster than PrivShare while handling two-party query processing, but at the cost of a very restricted deployment environment and a weaker threat model.

XPIR [1] is a practical protocol that allows a client to privately retrieve database records. However, XPIR scheme only retrieves an entire single record based on its index number. SealPIR [3] and Ali *et al.* [2] focus on improving CPIR by compressing and batching queries using probabilistic batch codes (PBCs). Despite the similarity in techniques, here we focus on the comparison for secure query processing systems and skip details for those PIR schemes.

6 Conclusion

In this paper, we design PrivShare, a multi-party database for secure query processing. By using our proposed techniques including grouped binary matching, range decomposition, and collision-tolerant folding, the protocol achieves practical performance in real-world applications without the non-colluding assumption or hardware enclaves.

The current PrivShare focuses on data models for horizontal split which is common in many real-world settings. In the future, we will explore data models for vertical split when JOINs are needed. We will also continue to evaluate our protocol in various multi-party settings to validate its design and further improve the overall performance.

References

[1] AGUILAR-MELCHOR, C., BARRIER, J., FOUSSE, L., AND KILLIJIAN, M.-O. Xpir : Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies 2016* (08 2015).

[2] ALI, A., LEPOINT, T., PATEL, S., RAYKOVA, M., SCHOPPMANN, P., SETH, K., AND YEO, K. Communication–computation trade-offs in pir. Cryptology ePrint Archive, Report 2019/1483, 2019. https://eprint.iacr.org/2019/1483.

[3] ANGEL, S., CHEN, H., LAINE, K., AND SETTY, S. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), pp. 962–979.

[4] BATER, J., ELLIOTT, G., EGGEN, C., GOEL, S., KHO, A. N., AND ROGERS, J. Smcql: Secure query processing for private data networks. *Proc. VLDB Endow. 10*, 6 (2017), 673–684.

[5] BEAVER, D., MICALI, S., AND ROGAWAY, P. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1990), STOC '90, ACM, pp. 503–513.

[6] BRAKERSKI, Z., GENTRY, C., AND VAIKUNTANATHAN, V. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277.

[7] CACHIN, C., MICALI, S., AND STADLER, M. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques* (Berlin, Heidelberg, 1999), EUROCRYPT'99, Springer-Verlag, p. 402–414.

[8] CHAUM, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (1981), 84–90.

[9] CHEON, J. H., KIM, A., KIM, M., AND SONG, Y. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017* (Cham, 2017), T. Takagi and T. Peyrin, Eds., Springer International Publishing, pp. 409–437.

[10] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 321–338.

[11] DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. C. *More Geometric Data Structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 211–233.

[12] DONG, C., AND CHEN, L. A fast single server private information retrieval protocol with low communication cost. In *Computer Security - ESORICS 2014* (Cham, 2014), M. Kutyłowski and J. Vaidya, Eds., Springer International Publishing, pp. 380–399.

[13] ESKANDARIAN, S., AND ZAHARIA, M. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow. 13*, 2 (Oct. 2019), 169–183.

[14] FAN, J., AND VERCAUTEREN, F. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch. 2012* (2012), 144.

[15] KIM, M., LEE, H. T., LING, S., AND WANG, H. On the efficiency of fhe-based private queries. *IEEE Transactions on Dependable and Secure Computing 15*, 2 (2016), 357–363.

[16] KUSHILEVITZ, E., AND OSTROVSKY, R. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science* (1997), pp. 364–373.

[17] LOPEZ-ROJAS, E. A., AND AXELSSON, S. Banksim: A bank payments simulator for fraud detection research inproceedings. In *26th EuropeanModeling and Simulation Symposium, EMSS* (2014).

[18] ODED, G. *Foundations of Cryptography: Volume 2, Basic Applications*, 1st ed. Cambridge University Press, New York, NY, USA, 2009.

[19] PAPPAS, V., KRELL, F., VO, B., KOLESNIKOV, V., MALKIN, T., CHOI, S. G., GEORGE, W., KEROMYTIS, A., AND BELLOVIN, S. Blind seer: A scalable private dbms. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 359–374.

[20] PODDAR, R., KALRA, S., YANAI, A., DENG, R., POPA, R. A., AND HELLERSTEIN, J. M. Senate: A maliciously-secure mpc platform for collaborative analytics. USENIX Security 2021, 2021. https://eprint.iacr.org/2020/1350.

[21] POPA, R. A., REDFIELD, C. M., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 85–100.

[22] Microsoft SEAL (release 3.0). http://sealcrypto.org, Oct. 2018. Microsoft Research, Redmond, WA.

[23] VOLGUSHEV, N., SCHWARZKOPF, M., GETCHELL, B., VARIA, M., LAPETS, A., AND BESTAVROS, A. Conclave: Secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[24] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splinter: Practical private queries on public data. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 299–313.

[25] YAO, A. C. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (1986), pp. 162–167.

[26] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 283–298.