Privacy-Preserving Database System with Hidden Queries

Miaomiao Zhang *mzhang01@manhattan.edu*
Computer Science Department
Manhattan College
Riverdale NY, 10583, United States

Abstract

As the increase of adopting database systems as the key data management technology by organizations for day-to-day operations and decision making, the security and privacy issues of these systems becomes crucial. Achieving privacy-preserving range query efficiently is a difficult challenge in practice. Many privacy-preserving protocols use secure multi-party computation (MPC) as building block, which present elegant privacy and security, but brings too much computation and communication overheads at the same time.

In this paper, we consider the three-party database system model: A client performing privacy-preserving range queries through a Proxy (trusted third party, TTP), to a Server's database. The User does not learn how the query applied on the database, nor any other quarriable attributes that the database may contain. The Proxy does not learn any information about the Server's private data, though he interacts with the Server directly. The Server on the other hand, learns nothing about the User's query.

We propose two practical privacy-preserving query schemes for database system. The basic idea of our schemes is to first convert each data entry into a set of concrete numbers, which is called attribute value numericalization. Then combines in a novel way several efficient cryptographic techniques, such as secure hash function, pseudo-random function, XOR, etc. to check whether the records match a query. The experimental evaluation (using the data sets collected by UCI KDD) of our prototype implementation show that our protocols incur reasonable computation and communicating overhead for added privacy-preserving benefit and perform better than those MPC-based solutions.

Keywords: Privacy-preserving, Database, Hidden Queries, Hash, Efficiency.

1. INTRODUCTION

1.1 Motivation

The digitization of our daily business and lives leads to an explosion in collecting personal data by corporations, individuals and governments. Such information is stored in large databases. Today, more and more applications and services rely on data stored remotely, such as in the cloud, and easy access to these databases will result in an increasing disclosure of private information about individuals. Similarly, many organizations need to protect their proprietary data from unauthorized access and administrators should be able to express various data access control policies. It is important to outfit today's data management infrastructure with methods to limit the disclosure of information. Many software systems request sensitive information from users to construct a query, but privacy concerns can make a user unwilling to provide such information.

For example, consider a medical system which maintains patient records. A potential patient Alice wants to consult about some information about a disease case. So, she makes a query to know whether there exists a record in the database that matches her query. In this example, if a match is found in the database, the system server immediately knows that Alice may have such a disease, even worse, after receiving Alice's query, server can derive additional information about Alice, such

as other health problems that Alice might have difficulty getting employment, insurance, credit, etc. So, patients wish to make query to the medical system in such a way that the server cannot infer which patients have which diseases.

Users are increasingly aware of privacy concerns and the need to maintain privacy in their online activities. Developing of such practical schemes is crucial to maintain user privacy in important application domains like pharmaceutical databases, patient database, online censuses, location-based services, real-time stock quotes, etc. To prevent such attacks, private information retrieval (PIR) was proposed (Chor et al. 1995) in the literature. By using PIR, a client can retrieve data from a database without the database server learning what data is being retrieved. However, most PIR based constructions are expensive, they incur high communication cost for the client, and not suitable to deploy in a database for supporting query processing over large data. More discussions on the related work can be found in Section 7.

In this paper, we are interested in addressing such security problems in a more practical way, i.e., avoid using heavyweight cryptographic operations (such as PIR used) and adopt several efficient cryptographic techniques, such as secure hash function, symmetric encryption, etc.

1.2 Problem Definition

Imagine a database system that consists of a Service Provider (*Server*), who store his private database contents at his own server, a set of *Users*, who want to access some certain categories of records of the database, and a *Proxy*, who is responsible for forwarding *User*s' query to *Server* and routing the corresponding files from Server's database to querying users. This scenario is shown in Figure 1.

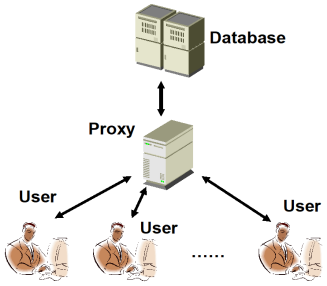

FIGURE 1: A typical example.

In a database system, a table is a set of data elements (values) that is organized using a model of horizontal rows and vertical columns. The columns are identified by attributes (fields) names, and the rows, which also called tuples (records) are identified by the values appearing in a particular column subset which has been identified as a candidate key. Queries are the primary mechanism for retrieving information from a database and consist of questions presented to the database in a predefined format.

Table 1 shows an example of a subset of a table *employee* with the attributes: *employee id (eid)*, *age* and *salary*.

eid	***age***	***salary***
12	40	58K
17	33	69K
35	27	40K
68	27	73K

TABLE 1: Plaintext relation of *employee*.

A record may include the tuple (eid, age, salary), of string or integer types respectively. A range

query retrieves all tuples in the database whose attribute has values in the interval [low, high]. In this example, a range query is defined as a predicate over those three attributes, such as (40 eid 68) and (30 age 50) and (20K *salary* 42K or 50K *salary* 100K). Upon receipt of a query to the *Server*, the *Proxy* performs a transformation on the query, and forward the hidden query to the database. Then the *Server* performs an interval-matchings on the data according to the query and aggregates data to decide which record should be send to the *User*. For example, a record in which the age is 35 and the salary is 40K will be send back to the user through the *Proxy*. Range queries usually involve numeric (or numerical) attributes.

The major security concerns of this system should be confidentiality and privacy.

- Since this is the *Server*'s private database, it may contain sensitive content and both the *Server,* and a querying *User* wish to keep the file content secret. For example, we want to make sure that the valid *User* who make a range query can only access the corresponding matched data. This is referred to as *data confidentiality*.
- Queries can reveal sensitive interest information about the *Users*, in which case the *Users* may wish to keep the query private. For example, when querying the *employee* database, the *User* may not want the *Server* to know which age of people they are interested in. This is referred to as *query privacy*.

Hereby, there appear two important problems for such 3-party privacy-preserving database system: Given *Server*, *Proxy* and *User*s,

Problem 1: How to design protocol that supports range query in a privacy-preserving manner in the above application scenario.

A satisfactory solution to this problem should meet the following three requirements: (1) The *User* cannot gain any more knowledge on the database, except the data returned to him. We refer to this requirement as *database privacy*. (2) It should be computationally infeasible for the *Server* to figure out a query. We refer to this requirement as *query privacy*. (3) The overhead of the solution should be marginal. Timely processing of every query is critical for distributed applications. We refer to this requirement as *protocol efficiency*.

In addition to the above three requirements, a desirable solution should not require the *Proxy* to be trusted; otherwise, the solution will be difficult to deploy. Therefore, we have another problem:

Problem 2: Design a protocol to support range query in a privacy preserving manner in the above scenario while *Proxy* is not trusted by either *Server* or *User*s.

We emphasize that the above two problems are different from the secure database query problem which has been the subject of much research (Du et al., 2001). To provide such a user with the means to retrieve data from a database without the database learning any information about the item that was retrieved in a two-party (server and user) model, private information retrieval (PIR) (Beimel et al. 2007, Chor et al. 1995) can be used.

In this work, in order to address such security problems in a more practical way, we want to avoid using those heavyweight cryptographic operations (such as PIR used). We use several lightweight cryptographic building blocks, such as secure hash function, symmetric encryption, etc. Therefore, we formulated the three-party model for the database system by introducing an additional party, the *Proxy*. In our model, The *Server* and the *Proxy* work in a collaborative way that the *Server*'s data which is unrelated to a query will not be revealed to the *User*, and the *User*'s query will not be revealed to the *Server*.

1.3 Contributions

We propose a new database system model which contains three parties: *Server*, *Proxy* and *Users*. Under this new model, we formulate and address two interesting problems, which have not been

studied in the literature. Then we present two practical, secure interactive protocols which support privacy-preserving database query to solve the problems. The protocols take place between the three parties, the Server, in possession of a private database, the Proxy which is a trusted third party (TTP) take responsibility of forwarding data that matched a query from the *Server*, and the *Users*, in possession of queries.

Our protocol does not reveal any data information except the records match the *User*'s query. The *User* does not learn how the query applied on the database, nor any other quarriable attributes that the database may contain. The *Proxy* does not learn any information about the *Server*'s private data, though he is interacting with the *Server* directly. The *Server* on the other hand, learns nothing whatsoever about the *User*'s query.

The problem we deal with is not trivial. Achieving security properties efficiently is a difficult challenge in practice. Many privacy-preserving protocols for PIR use secure multi-party computation protocol as building block, which present elegant for security, but brings much computation and communication overheads at the same time. Our protocol combines a novel way several efficient cryptographic techniques, such as secure cryptographic hash function, pseudorandom function, XOR, etc.

We also present an experiment using the data sets collected by UCI KDD. Experimental evaluation of our prototype implementation demonstrates that our protocol performs significantly better than those MPC base solutions.

1.4 Outline

This paper is organized as follows. We introduce the preliminary in Section 2. In Section 3, we present our first protocol for problem 1, which constitutes database processing and query processing. In Section 4, we present an advanced protocol for problem 2, in which the *Proxy* is not a trusted party. In Section 5, we discuss some practical considerations and security issues. We give the experimental results in Section 6 and analyze the performance. We describe related work in Section 7. Concluding remarks are given in Section 8.

2. PRELIMINARIES

Throughout this paper, attributes are denoted by A_i and attribute values are denoted by lowercase letters such as the *j*-th record value for attribute *i* is a_{ij}. There can be additional content associated with the record that is not included in the quarriable record. We call this information the payload or file of the record. Because the payload is not needed for making interval matching of range queries, in this paper we will only be concerned with the attributes of the database which support range query, and the payload for each record are exist in a plaintext form. In the employee example, the attributes are $A_1 = eid$ (string), $A_2 = age$ (integer) and $A_3 = salary$ (integer). The quarriable record is defined as the tuple (*eid, age, salary*).

2.1 Xhash and Oblivious Comparison

In this section, we consider the following oblivious comparison problem. Suppose we have two parties, denoted *Server* and *Proxy*, where each party has a private number, $N_1$and N_2 respectively. *Server* and *Proxy* want to compare whether $N_1 = N_2$; however, no party wants to disclose its number to the other. In case $N_1 \neq N_2$, no party should learn the value of the other party.

In this paper, we will use a simple and efficient protocol (Liu et al., 2008) based on cryptographic hash, called Xhash, to achieve oblivious comparison. Xhash works as follows. First, *Server* and *Proxy* each choose a secret key K_S and K_P respectively. Second, *Database* sends $N_1 \oplus K_S$ to *Proxy*. Then, *Proxy* computes $HMAC_k(N_1 \oplus K_S \oplus K_P)$ and sends the result to *Database*. Third, *Proxy* sends $N_2 \oplus K_P$ to *Database*. Then, *Database* computes $HMAC_k(N_2 \oplus K_P \oplus K_S)$ and compares it with $HMAC_k(N_1 \oplus K_S \oplus K_P)$, which was received from *Proxy*. Finally, the condition

$N_1 = N_2$ holds if and only if $HMAC_k(N_2 \oplus K_P \oplus K_S) = HMAC_k(N_1 \oplus K_S \oplus K_P)$. Figure 2 illustrates the Xhash protocol.

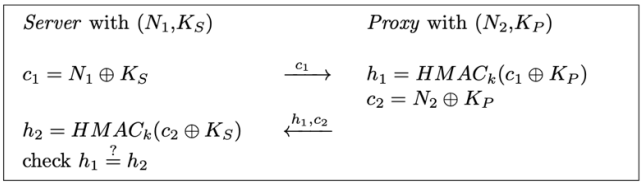

FIGURE 2: The Xhash protocol.

The above function HMAC is a keyed-Hash Message Authentication Code, such as HMAC-MD5 or HMAC-SHA1, which satisfies the one-wayness property (i.e., given $HMAC_K(x)$, it is computationally infeasible to compute *x* and *k*) and the collision resistance property (i.e., it is computationally infeasible to find two distinct numbers *x* and *y* such that $HMAC_K(x) = HMAC_K(y)$. Note that the key *k* is shared between *Server* and *Proxy*. Although hash collisions for HMAC do exist in theory, the probability of collision is negligibly small in practice. Furthermore, by properly choosing the shared key *k*, we can safely assume that HMAC has no collision.

To prevent brute force attacks, we need to choose key *K* to be sufficiently long. In our implementation, we choose *K* to be 128 bits. Note that in our framework *x* might be much less than 128 bits. To meet the length of *K* such that *x* can be XORed with *K*, we use a pseudorandom function *prf* to expands *x* into a longer string so that the output matches the key size. Let *x* be the seed and feed it into *prf*. The output $x' = prf(x)$ looks indistinguishable with the real random number of 128 bits long.

The correctness of Xhash follows from the facts that the commutative property of XOR operation (i.e., $x \oplus K_P \oplus K_S = x \oplus K_S \oplus K_P$), and the one-wayness and collision resistance properties of HMAC functions. Note that in the case that $N_1 = N_2$, *Server* can compute the secret key of *Proxy* because $N_2 \oplus K_P \oplus N_2 = K_P$. However, when applying the above oblivious comparison scheme in our protocol, *Server* compares N_2 with many numbers in a set and does not know which number is equal to N_2.

2.2 Prefix Membership Verification

We define two new concepts here: *k*-prefix and prefix family. We call the prefix $\{0, 1\}^k \{*\}^{w-k}$ with *k* leading 0s and 1s followed by *w-k* *s a *k*-prefix. If a value *x* matches a *k*-prefix, the first *k* bits of *x* and the *k*-prefix are the same. For example, if $x \in 01^{**}$ (i.e., $x \in [0100, 0111]$), then the first two bits of *x* must be 01. Given a binary number $b_1 b_2 \ldots b_w$ of *w* bits, the prefix family of this number is the set of *w+1* prefixes $\{b_1 b_2 \ldots b_w, b_1 b_2 \ldots b_{w-1}, \ldots, b_1 {}^* \ldots {}^*, {}^{**} \ldots {}^*\}$, where the *i*-th prefix is $b_1 b_2 \ldots b_{w-i+1} {}^* \ldots {}^*$. We use *PF(x)* to represent the prefix family of *x*. For example, $PF(0101) = \{0101, 010^*, 01^{**}, 0^{***}, {}^{****}\}$. Based on the above definitions, it is easy to draw the following conclusion: Given a number *x* and a prefix $\mathcal{P}$, $x \in \mathcal{P}$ if and only if $\mathcal{P} \in PF(x)$.

2.3 Prefix Numericalization

A prefix numericalization scheme *f* needs to satisfy the following two properties: (1) for any prefix $\mathcal{P}$, $f(\mathcal{P})$ is a binary string; (2) for any two prefixes $\mathcal{P}_1$ and $\mathcal{P}_2$, $f(\mathcal{P}_1) = f(\mathcal{P}_2)$ if and only if $\mathcal{P}_1 = \mathcal{P}_2$.

There are many ways to do prefix numericalization. In this paper, we use the following scheme: Given a prefix $b_1 b_2 \ldots b_k {}^* \ldots {}^*$ of *w* bits, we first replace every * by 0; second, we append $\lceil \log_2(w+1) \rceil$ bits whose value is equal to *k*. For example, 101* is converted to 1010011. After prefix numericalization, each prefix becomes a concrete number.

3. THE PROPOSED SCHEME

3.1 Assumptions and Threat Model

Assumptions: First, we assume that database systems allow each query to be range-based. And even attributes such as name, not typically thought of as numerical, can be indexed and therefore linearized in some fashion. Second, we assume that all parties running the protocols are semi-honest, or honest-but-curious. This means that they execute the protocol exactly as specified, but they may attempt to glean more that the obvious information by analyzing the transcripts. Third, we assume there exists secure channels, which could be achieved using protocols such as SSL.

Threat Model: The *Server* may attempt to break the query request and the *Proxy* may attempt to extract more information both from *Server* and *User*s' sides.

3.2 Database Processing

In the database processing protocol, *Server* first transfers its attribute values into binary forms, then converts each of the binary string to a set of concrete numbers, which is called prefix numericalization. Second, Server applies XOR operation to every set of numbers using its secret key K_S. Third, for each attribute, *Server* sorted all the attribute values and maintain a list of pointers used for binary search. Finally, the *Server* sends the anonymized attribute values together with the lists of pointers and non-private data (payload) to the *Proxy*. Then the *Proxy* further applies XOR and HMAC operations to every number in the received attribute values using its secret key K_P, encrypts the lists of pointers of each attribute with a separate secret key *K*. At last, *Proxy* sends the resulting table and encrypted pointers back to *Server* and keep the non-private data in his own server.

1. Attribute values numericalization

In this step, *Server* converts each attribute values in the database table to a concrete number. Let $\mathcal{N}$ denotes the numericalization function. It works as follows. For attribute A_i, let w_i denote the maximum number of bits used to represent the attribute values. So, for attribute value a_{ij}, where *j* denotes the *j*-th record, it's binary string form is $b_1 b_2 ... b_{w_i} = \mathcal{N}(a_{ij})$. Second, for each binary form of attribute values, the *Server* generates its prefix family $PF(\mathcal{N}(a_{ij}))$. Third, the *Server* converts each prefix to a number using the prefix numericalization scheme *f*. That is append $\lceil \log_2(w_i + 1) \rceil$ bits whose value is equal to the prefix length and get the final result, a numericalized set $f(\mathcal{N}(a_{ij}))$ for each attribute value a_{ij}. Note that, the size of each set is $w_i + 1$. This means the quarriable data will expand $w_i + 1$ for each attribute value a_{ij}. For example, for an age value 35, we first use 8 bits to represent the binary form 00100011. The prefix family $PF(\mathcal{N}(35))$ is shown in Figure 3(b). The final sequences of numbers are shown in Figure 3(c).

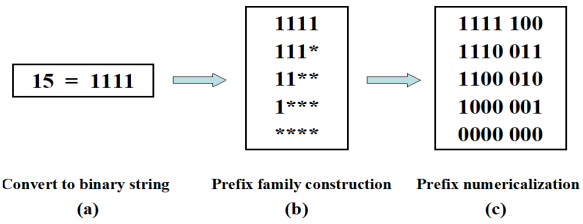

FIGURE 3: Example of attribute value processing by *Database.*

Table 2 and Table 3 show the plaintext form of a database and the data entries after the attribute value numericalization.

A_1	A_2	…	A_i	…	A_n
a_{1j}	a_{2j}	…	a_{ij}	…	a_{nj}

TABLE 2: Plaintext form of the *j*-th record in a database.

A_1	…	A_n
$< PF(\mathcal{N}(a_{1j})) >$	…	$< PF(\mathcal{N}(a_{nj})) >$

TABLE 3: Numericalized form of the *j*-th record.

2. Applying XOR by the *Server*
After the numericalization of attributes values, the *Server* applies XOR to every number in the expanded using its secret key K_S. This step is applying the first step of Xhash protocol.

3. Sorting and pointers generation by the Server
In the last step, each attribute value for A_i has an expansion of $w_i + 1$. To improve the comparison efficiency on the database side, we want to remove the redundant items and put the left numericalized numbers in a regular way. For each attribute A_i, the *Server* sorted all the numericalized numbers in a sequence C_i, then merge the repeated numbers. We utilize sorting algorithm here for later improve the searching speed by applying binary search. For each value c_{ij} in the sorted sequence, we maintain a pointer p_{ij}, which points to the row number where the corresponding numericalized attribute value comes from. Note that, many records in a table may have a same numericalized attribute value, e.g., for attribute *age*, 00000000 0000 will always appears in the numericalized set of each attribute values. So the pointer may point to one or several records of the table.

Since there are *n* attributes in a database, we totally have *n* corresponding sorted sequences $C_1, C_2, ..., C_n$. Figure 5 shows an example for the sorted sequence and the corresponding pointers of attribute *age*.

Finally, the *Server* sends the sorted data lists, together with their pointers and non-private data in the original table form to the *Proxy* for further processing.

Note that the *Server* should do XOR operation before sorting. If the *Server* do the sorting first, then all the numbers will be arranged in the ascending or descending form. We can notice that the binary string with all 0 bits concatenate ending "00", "01" "10" "11" may always or often appear after the numericalization. If XOR the numbers with K_S after sorting (assume ascending form), the first several numbers on the XORed sorting list may reveal some bits of the *Server*'s secret key, which makes it possible for the *Proxy* to attack *Server*'s secret key.

If the Server do XOR before sorting, it obvious that the *Proxy* cannot apply the above attack, either cannot get the quarriable data since he doesn't know the *Server*'s secret key K_S.

4. Applying XOR and HMAC by *Proxy*
Upon receiving the sorted lists of numerical sets of attribute values with pointers and non-private files from the *Server*, the *Proxy* further applies XOR and HMAC to every number in the received sequences using its secret key K_P. We use D_i denotes the resulting sequence for attribute A_i.

5. Encrypt the pointers by *Proxy*
To prevent the *Server* from knowing the files of which user get after the query, the *Proxy* further encrypts each pointer using another secret key *K*.

After *Proxy* applies XOR and HMAC and encrypts the pointers, it sends the resulting sequences together with the encrypted pointers back to the *Database*. The *Proxy* keeps the non-private files in its own server.

A_1	…	A_n
$< PF(\mathcal{N}(a_{1j})) \oplus K_S >$	…	$< PF(\mathcal{N}(a_{nj})) \oplus K_S >$

TABLE 4: XORed form of the *j*-th record.

6. Sorting the hash values by the *Server*
On receiving the lists of hash values from the Proxy, the *Server* sorted each list again for later search. It can also help the *Server* detect hash collision. We will discuss this issue later.

3.3 Query Processing Protocol

In this protocol, each time the *Proxy* receives a query (a range for an attribute or several ranges for several attributes) originated from the user, the *Proxy* first converts the range to binary numbers and then converts to prefixes. Utilizing the prefix numericalization scheme it further converts each prefix to a concrete number. Then the *Proxy* applies XOR to every number converted from the range query using its secret key K_P and send the resulting query number set to the *Server*. On receiving the encrypted query number set, the Server further applies XOR and HMAC to the received numbers, and searches the double encrypted and hashed sequences which is collaboratively generated by both the *Server* and the *Proxy* in the database processing protocol. Finally, the *Server* sends the corresponding encrypted pointers back to the *Proxy* and the *Proxy* decrypts it using his secret *K*.

1. Query Preprocessing by Proxy

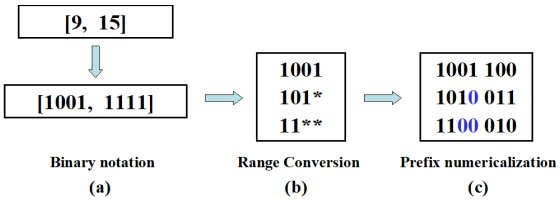

FIGURE 4: Example of range query preprocessing by *Proxy*.

Let $Q = <(i,q_i)>$ denote a range query sent by a *User*, where (i,q_i) denote the range predicate on the *i*-th attribute A_i. And let *d* be the number of attributes related in a query. For each of the *d* fields of a query, the *Proxy* first converts it into the binary form, then generates its corresponding minimum set of prefixes such that the union of the prefixes is equal to the range. Second, the *Proxy* converts each prefix to a number using the prefix numericalization scheme. Third, *Proxy* applies XOR to each number in the numericalized query sets using its secret key K_P. Last, the *Proxy* sends a sequence of *d* sets of numbers, which corresponds to the *d* fields of the query, to the *Server*. For example, given an age query [9, 15], the translation process is shown in Figure 4.

2. Query Preprocessing by *Server*
After the *Server* receives the query as a sequence of numbers from the *Proxy*, the *Server* further applies XOR and HMAC using its secret key K_S. Let R_i denotes the resulting (hidden) query sets for attribute A_i. Because of the commutativity property of Xhash, the *Server* can do the search and comparison on the encrypted quarriable attributes values in the table. Recall that there are *n* attributes in a database. A record $(r_1,r_2,...,r_n)$ matches a query $Q = <(i,q_i)>$ for attribute A_i if and only if the condition $r_i \in q_i$ holds.

3.4 Searching and Comparing by the Server

As mentioned above, there is data expansion and data redundancy in the database table after applying prefix numericalization. It's not efficient and impractical to go through all the numericalized attribute values in the encrypted sequences. We can use binary search here to achieve an elegant searching speed.

According to the prefix membership verification, we know that if we find a number in the sequence D_i exact matches a number in the hidden query set R_i, then the row numbers which the corresponding pointer point to are satisfied with this range query to attribute A_i.

After the *Server* find out all the records that match a query, it sends the corresponding encrypted

pointers to the *Proxy*. The *Proxy* then decrypts it using its secret key *K*. According to the decrypted pointers, the *Proxy* send the matched files back to the *User*. For example, if a *User* makes a query of [8,13] to the data table in Figure 5. The *Proxy* first generates the numericalized query set <1100 011, 1000 010>. Searching the sorted list, we can see that row 2 is the record which the user can retrieve files.

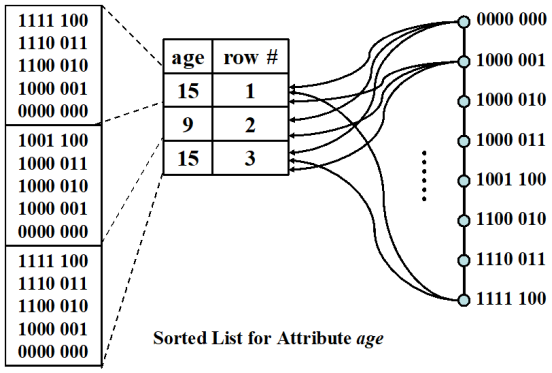

FIGURE 5: Example of database search based on sorted pointers.

4. AN ADVANCED SCHEME SUPPORTING HIDDEN QUERIES

To solve ***Problem 2***, we present an advanced protocol which can support hidden queries to the *Proxy* in this section. Based on the assumptions of last scheme, we further assume that the *Proxy* is not trusted by both the *Server* and the *User*s. They want to prevent the *Proxy* known about their quarriable data and queries. We also assume that the *User*s and *Server* share a common key κ. We group all the users into *m* small groups. For each group, the members share a same group key κ_i with the *Server*. A secure key distribution scheme such as (Neuman et. al. 1994, Ganesan et al. 1995, Reiter et al. 1996, Wong et al. 2000) suitable for the specific requirements (e.g., scalability) of an organization can be used for this purpose. We will discuss the group key management issue later. Our assumptions are reasonable in practical scenarios. This advanced solution is based on the basic scheme given in the previous section and a secure encryption algorithm, denoted by *E*. The protocol runs as follows.

4.1 Database Processing

When the *Server* wants to establish the quarriable tables, after the attribute values numericalization, see Table 3, the *Server* first applies encryption on all the numbers using the shared key κ_i. The resulting quarriable table shows in Table 5. That means, for each group, the *Server* should maintain a corresponding table in the processing phase.

A_1	…	A_n
$< E_{K_i}(PF(\mathcal{N}(a_{1j})))>$	…	$< E_{K_i}(PF(\mathcal{N}(a_{nj})))>$

TABLE 5: The *j*-th record after encryption with the shared key.

…	A_i	…
…	$< E_{K_i}(PF(\mathcal{N}(a_{ij}))) \oplus K_S >$	…

TABLE 6: The XORed *j*-th record after encryption with the shared key.

Then the *Server* apply XOR on all the attribute values of this table using its private key K_S. The results are shown in Table 6. After that, the Server sorted all the numbers for each attribute and generate the pointer lists the same way as in last scheme. Finally, the *Server* sends these sorted

encrypted data lists to the *Proxy* together with their pointers and non-private data for further processing.

The *Proxy*'s operations are the same as in basic solution. The *Proxy* applies XOR and HMAC, encrypts the pointers, keeps the non-private data and sends the hashed lists together with the encrypted pointers back to the *Server*.

The *Server* will further sort each list again for later search.

4.2 Query Processing

When a *User* makes a query to the *Server* and doesn't want to reveal the query to both the *Proxy* and *Server*, it will do the query preprocessing first. So compared with our basic solution, Query Processing will be divided into three phases, which will be operated by *User*, *Proxy* and *Server* step by step separately.

1. Query Processing by the *User*:
For each of the *d* fields of a query, the *User* first convert it into the binary form, then generates its corresponding minimum set of prefixes such that the union of the prefixes is equal to the range. Second, *User* converts each prefix to a number using the prefix numericalization scheme. Third, encrypt each number in these *d* sets using his shared group key κ_i and send the encrypted number sets to the *Proxy*. In addition, the Proxy should attach his group number to the *Proxy*.

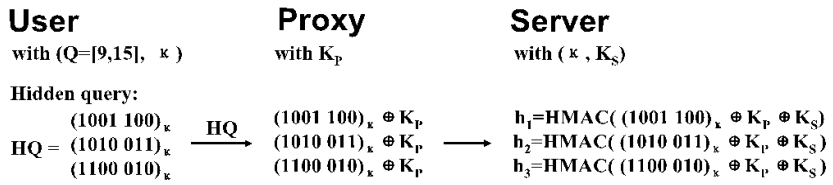

FIGURE 6: A query example includes three steps.

2. Query Processing by the *Proxy*:
After receiving the encrypted form of query from the *User*, the *Proxy* applies XOR to each number using its secret key K_P and sends the results to the *Server* together with the *User*'s group number. The *Proxy* cannot figure out the *User*'s query since he doesn't know the then *User*'s key.

3. Query Processing by the *Server*:
In this phase, the operations of Server are the same as in the basic solution. Figure 6 gives an example of the three steps of query processing.

4.3 Searching and Comparing by the Server

On receiving the group number of the User passing through the *Proxy*, the *Server* should first determine the processed data for the *User*'s group to do the search.

The searching and comparing phases are the same as the previous scheme.

5. DISCUSSIONS

5.1 Other Practical Considerations

1. Database Update
When the *Server* updates its database, the *Server* and the *Proxy* don't need to run the whole database processing protocol again. Instead, they only need to process the updated data records.

When adding a new record to the database, the *Server* first do numericalization for each attribute values in this record. Then XOR the numbers with its secret key K_S and construct new pointers to store the corresponding row numbers for them. After that, the *Server* send all the numbers with

pointers and the non-private files of this row to the *Proxy*.

Upon receiving the data from the *Server*, the *Proxy* apply XOR and HMAC to the XORed numbers, encrypts the pointers, add the non-private file directly to the payload table he kept. Then return the hash values together with the encrypted pointers to the *Server*.

The *Server* need to insert the new computed hash values to the related attribute hash lists. It first applies binary search on the list, if a hash value already exists on a list, concatenate the hash value's corresponding encrypted pointer to the pointer for the existed hash value. If a hash value has not appeared on the list, insert it into the proper location together with its encrypted pointer. Hence the update operation of adding a new record is done. The update process for deleting a record is the same.

2. Treatment of string
So far, all the examples we give are for integer attribute values. As mentioned in our assumption, the attributes such as name, not typically thought as numerical, can be indexed in some fashion. Here we show how to apply range query on the strings in our protocol.

Since we know that each letter has a unique ASCII code of 8 bits. Let's take attribute *name* for example. We concatenate all the ASCII codes of letters in a name, then padding it to form a certain length of binary string, e.g., 128 bits. When a query is made for the name of a beginning letter "A", we should first convert "A" into its ASCII form "01100101", then concatenate 120 "*"s to convert it into a range (a prefix) which includes all the 128 bits binary strings of names with a beginning letter "A". It obvious that we can apply the query on the attributes of string format.

5.2 Security Properties

The protocol presented in Section 4 is secure in the semi-honest model, i.e., under the assumption that participants faithfully follow the protocol, but may attempt to learn extra information from the protocol transcript.

1. Privacy preserving
In our first scheme, the basic scheme, after the server processing protocol, the *Proxy* learns only the size of the database. It's difficult for the *Proxy* to learn other information such as the values of the private attribute values, the non-private files' relation with the attribute values. The *Server*'s privacy is protected by the Xhash protocol. After the XOR operation by the *Server*, the resulting sequences seem to be random to the *Proxy*. The *Proxy* cannot gain any information about the original attribute values unless he gets the *Server*'s secret key K_S. In addition, as we mentioned before, the *Server* do the sorting after the XOR operation can prevent the partial disclose of the *Server*'s secret key. On the other hand, the *Server* learns nothing about the pointers and their relationship to the attribute values. Since the *Proxy* applies XOR and the one-way hash function. Due to the one-way property, it is computationally difficult for the *Server* to figure out the *Proxy*'s secret key K_P or the XORed values he send to the *Proxy*. Hereby, the *data confidentiality* is guaranteed.

After a query protocol, the *User* gets the information which fits for his query ranges, and the Server learns nothing about the query of the User. Due to the query preprocessing operations, the *User*'s range queries are mapped to hash values, the one-wayness of hash function made it difficult for the *Server* to figure out the original query. What's more, after the comparison, the *Server* can only get the encrypted pointers which points to the satisfied data entries of the query. It is computational difficult for the *Server* to decrypt the pointer, because only the *Proxy* has the key. Since the *Proxy* will decrypt the pointers and send the related non-private files back to the *User*, the *Server* doesn't know what's the exact files returned to the *User*, hence unable to guess the rough range of the *User*'s query. It's clear that the *query privacy* is achieved in our scheme.

In the advanced protocol, we further prevent the *Proxy* know about the *User*'s query. To prevent this, the *User* first encrypted his query with the shared key. Hence what the *Proxy* received is a

hidden query of the *User*, the *Proxy* cannot figure out the *User*'s query since he doesn't know the shared key. Although the *Proxy* can learn the size of the hidden query, but it useless to figure out it, since the size of the hidden query does not only depend on the query range. Since other processing phases are similar to the basic scheme, the *data confidentiality* and *query privacy* can be guaranteed in a same way.

2. Hash collision
The chance of having hash collisions for HMAC is extremely small. However, to be on the safe side, we propose the following solution to the problem. Our solution is based on the observation. In the database processing phase in our protocol, the *Proxy* will apply XOR and hash operation on a sorted list without redundancy. When he returns the hash lists back to the *Server*, the *Server* will sort the hash lists again. If a hash collision happens, the sorted hash values will have redundancy (two neighbors on the sorted list are of the same value). This fact can be observed by the *Server,* and he can easily detect whether hash collision happens.

Note that the *Proxy* should maintain both the input and output lists of HMAC for observation. In the database update phase, if a hash value already exists on a list, the *Proxy* should first compare the input values of HMAC to make sure if this is a real hash collision or a redundancy.

In the case that hash collision does happen, the *Server* and the *Proxy* can simply redo the data processing phase, in which they will choose different secret keys and henceforth the hash collision is most likely removed.

3. Group key management
In the advanced scheme, we group all the users into *m* small groups. And for each group, the members share a same key with the Server. For security concern we need to update the group key periodically to prevent the brute force attack to the shared keys. Also considering the dynamic membership of a group, the group key needs to be updated upon each user's join to prevent the new user from accessing the past group key. Similarly, upon each user's departure, the key needs to be updated to prevent the leaving user from accessing the future group key. Thus, group members need to agree upon the same key management protocol for key establishment and update. Sometimes the group key management protocol is also referred to as the group key agreement.

Collusion can also be a problem. If a user of one group is compromised by an adversary or the *Proxy*, it will lead to the leakiness of the group key. Hereby, we are going to apply a secure group key management protocol with Collusion-Scalability.

The topic of secure group communications has been investigated a lot. How to distribute a secret to a group of users has been addressed in the cryptography literature. Considering the need for frequent key changes and the associated scalability problem for a large group, we are going to apply a hierarchical approach which developed by (Wong et al., 2000). Hence, the confidentiality issue of the shared key can be guaranteed by the secure group key management protocol we used.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the privacy-preserving database query protocol. We implemented our protocol using Java 17. Our experiments were carried out on a laptop PC running Windows 10 with 8G memory and 11th Gen Intel Core i3 processor. This is a realistic approximation of what the *User* might use, but we expect that the *Server* and the *Proxy* would maintain more powerful dedicated servers to manage the database and apply queries.

In the experiment, we use HMAC-SHA1 algorithm and AES to encrypt the pointers. In our analysis of scaling behavior, we use the data sets downloaded from UCI KDD Archive with various attributes and records and measured the computation time of each party separately. In the database setup phase, the computation time of the *Proxy* includes the XOR and HMAC, and encryption of pointers,

while the computation time of the Server comes from the numericalization, XOR and twice sorting. Query time is the more important metric since it dictates how long the three party must maintain a connection. Database update calculations can be performed during idle times when the CPU is in low demand.

The database processing time of our basic scheme with an input table (100003) is 4.4 seconds. Adding one more entry to the existing database (13) is 11ms. Table 7 shows the query processing time for a 10000-row table.

For the advanced solution, the average querying time is shown in Table 8. The performance of the advance scheme is very close to the basic scheme. Because in the advanced scheme, for each query processing, we need one more encryption by the shared key, which is based on the XOR operation.

We also analyzed the scaling behavior of the database processing algorithm. We find that the *Server*'s algorithm and *Proxy*'s algorithm scale linearly with the number of rows in the database. Both algorithms' computation time depends linearly on the size of the database.

We also evaluate the scaling behavior of query algorithm. The computation of query algorithm is independent of the attributes. But not scales linearly with the row numbers. This is because the query processing time also depends on the query range. If the query range is large, it will need more processing time.

For the database update algorithm, it scales linearly with the number of rows in the database.

The performances show that the computation cost of executing our privacy-preserving database query protocols perform efficiently in practical scenarios.

QUERY	name	age	salary
TIME	0.9ms	0.5ms	5.6ms

TABLE 7: Average querying time of the advanced scheme based on 10000 tests.

7. RELATED WORKS

Yao first investigated the secure two-party computation problem (Yao et al., 1986). This problem was later generalized to multiparty computation (MPC). Du (Du et al., 2001) et al. provides a good review of secure multi-party computation (MPC) problems and apply it to solve the secure remote database query problem. They have developed three models for secure remote database access and presented a class of problems and solutions for these models: Private Information Matching (PIM) Problem, Secure storage Outsourcing (SSO) Model and Secure Storage and Computing Outsourcing (SSCO) Model. Although the authors claimed that the solutions for the above three problems (Du et al., 2001) are practical, there are just a theoretical result compared with the general solution from multi-party computation. Their protocols are still not suitable for practical use.

Our work differs from Du's work in many ways. First, our framework involves three parties, while the Proxy is a trusted third party used for forwarding matched message. Second, the computational cost of our framework is low due to the use of efficient Xhash, while computational cost of Du's solutions is expected to be high due to the extensive use of PKI certificates.

The private information retrieval (PIR) problem has been widely studied. A PIR protocol allows a user to access k ($k>1$) duplicated copies of data, and privately retrieve one of the n bits of the data in such a manner that the databases cannot figure out which bit the user has retrieved. The trivial solution has an $O(n)$ communication complexity. Much work has been done for reducing this communication complexity (Chor et al. 1998, Di-Crescenzo et al. 1998, Gertner et al. 1998, Ishai et al. 1999, Kushilevitz et al. 1997). Tillem et al. (Tillem et al. 2016) proposed a PIR scheme

supporting range queries, and Wang et al. (Wang et al.) proposed PIR schemes providing functionality approaching standard SQL, including range queries.

Recently, Hayata et al. (Hayata et al. 2020) propose a simple extension of the standard PIR security notion to range queries, give a simple generic construction of a PIR scheme meeting the stronger security notion, which has a round complexity logarithmic in the size of the database. They also propose a more efficient direct construction based on function secret sharing, the round complexity of the latter is constant. Finally, we report on the practical performance of our direct construction.

Our work is related to privacy-preserving inter-database operation, which is another interesting problem to design protocols for distributed computation of relational intersections and equi-joins such that each site gains no information about the tuples at the other site that do not intersect or join with its own tuples. Such protocols form the building blocks of distributed information systems that manage sensitive information, such as patient records and financial transactions, that must be shared in only a limited manner.

Agrawal, Evfmievski, and Srikant presented a general approach (Agrawal et al., 2003) based on a commutative encryption scheme in a symmetric way. The commutative property of the encryption scheme allows for two parties to obliviously compare two values. If the equality comparison fails, no information about the values is learnt by the other party. This approach also solves the problem at the cost of high computation overheads, since commutative encryption is a kind of public key encryption algorithm.

Unlike the general solution outlined in (Agrawal et al. 2003), Liang and Chawathe presented protocols in an asymmetric way (Liang et al., 2004): They use blind signatures to protect the privacy of one party and one-way hash functions to protect the privacy of the other. Their solution is the state-of-art for such privacy-preserving inter-database operation problem.

Huberman, Franklin, and Hogg have discussed a problem very similar to ours in the context of recommendation systems (Huberman et al., 1999, 2000). Their protocol is used to find people with common preferences without revealing what the preferences. In a database context, Lindell and Pinkas have addressed the same privacy concerns in (Lindell et al., 2000). Their paper addresses the privacy-preserving set union problem. The central idea is to make all the intermediate values seen by the players uniformly distributed. But all these works are theoretical results. Their solutions are not practical from the efficiency angle.

8. CONCLUSING REMARKS

In this paper, we consider a three-party database system model and present two practical secure protocols for database system. Our basic scheme enables a *User* to apply a query through a trusted party, the *Proxy*, on the *Server*'s database in a privacy-preserving manner. In the advanced scheme the *Proxy* does not need to be a trusted third party, the *User* can make a hidden query through the *Proxy*. As we are aiming at practically oriented database schemes, besides privacy, efficiency is a major challenge. Different from previous related work, we avoid using heavyweight building blocks that used in PIR etc. Instead, to achieve oblivious comparison, our scheme uses an efficient Xhash protocol which is based on lightweight cryptographic hash function and symmetric encryptions. We implemented our protocols and conducted experiments on real life data sets. It turns out that our schemes perform well, which makes them very attractive in practice.

9. REFERENCES

Agrawal, R., Evfimievski, A. and Srikant, R. (2003). Information sharing across private databases. International Conference on Management of Data, 132–143.

Beimel, A. and Stahl, Y. (2007). Robust Information Theoretic Private Information Retrieval. *J. Cryptol.*, 20(3):295–321.

Chor, B., Kushilevitz, E., Goldreich, O. and Sudan, M. (1998). Private information retrieval. Journal of the ACM (JACM), 45:965 – 981.

Chor, B., Goldreich, O., Kushilevitz, E. and Sudan, M. (1995). Private information retrieval. *In FOCS*, pages 41–50.

Di-Crescenzo, G., Ishai, Y. and Ostrovsky, R. (1998). Universal service-providers for database private information retrieval. *In Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing (PODC'98)*, 91–100, New York, NY, USA.

Du, W. (2001). A Study of Several Specific Secure Two-Party Computation Problems. PhD thesis, Purdue University, West Lafayette, Indiana.

Du, W. and Atallah, M. J. (2001). Secure multi-party computation problems and their applications: a review and open problems. *In Proceedings of the 2001 workshop on New security paradigms (NSPW'01)*, pages 13-22, New York, NY, USA. ACM.

Ganesan, R (1995). Yaksha: augmenting kerberos with public key cryptography. *In Symposium on Network and Distributed System Security (SNDSS'95)*.

Gertner, Y., Goldwasser, S., and Malkin, T. (1998). A random server model for private information retrieval or how to achieve information theoretic PIR avoiding database replication. *In Randomization and Approximation Techniques in Computer Science*.

Gertner, Y., Ishai, Y., Kushilevitz, E., and Malkin, T. (1998). Protecting data privacy in private information retrieval schemes. *In Proceedings of the thirtieth annual ACM symposium on Theory of computing (STOC'98)*, pages 151-160, ACM.

Hayata, J. & Schuldt, J. & Hanaoka, G. & Matsuura, K. (2020). On Private Information Retrieval Supporting Range Queries. *Computer Security – ESORICS 2020, 25th European Symposium on Research in Computer Security*, Proceedings, Part II, pages 674-694.

Hayata, J., Schuldt, J.C.N., Hanaoka, G., Matsuura, K. (2020). On Private Information Retrieval Supporting Range Queries. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds) Computer Security – ESORICS 2020. ESORICS 2020. Lecture Notes in Computer Science(), vol 12309. Springer.

Hogg, T., Huberman, B. A., and Franklin, M. (2000). Protecting privacy while sharing information in electronic communities. *In Proceedings of the tenth conference on Computers, freedom and privacy (CFP'00)*, pages 73-75.

Huberman, B. A., Franklin, M. and Hogg, T. (1999). Enhancing privacy and trust in electronic communities. *In ACM Conference on Electronic Commerce*, pages 78–86.

Ishai, Y. and Kushilevitz, E. (1999). Improved upper bounds on information-theoretic private information retrieval. *In Proceedings of the thirty-first annual ACM symposium on Theory of computing (STOC'99)*, pages 79-88.

Kushilevitz, E. and Ostrovsky, R. (1997). Replication is NOT needed: SINGLE database, computationally- private information retrieval. *In IEEE 38th Annual Symposium on Foundations of Computer Science*, pages 364-373.

Liang, G. and Chawathe, S. S. (2004). Privacy-preserving inter-database operations. In Intelligence and Security Informatics, volume 3073, pages 66–82. Springer Berlin/Heidelberg.

Lindell, Y. and Pinkas, B. (2000). Privacy preserving data mining. *In Advances in Cryptology (CRYPTO'00)*, pages 36 – 54.

Liu A. and Chen, F. (2008). Vguard: Collaborative enforcement of firewall policies in virtual private

networks. *In Twenty-Seventh Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'08)*.

Neuman, B. C. and Tso, T. (1994). Kerberos: an authentication service for computer networks. *Communications Magazine*, IEEE, 32:33–38.

Reiter, M. K., Franklin, M. K., Lacy, J. B. and Wright, R. N. (1996). The omega key management service. *In ACM Conference on Computer and Communications Security*, pages 38-47.

Tillem, G., Candan, O.M., Savaş, E., Kaya, K. (2016). Hiding access patterns in range queries using private information retrieval and ORAM. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 253–270. Springer, Heidelberg.

Wong, C. K., Gouda, M. and Lam, S. S. (2000). Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8:16–30.

Wang, F., Yun, C., Goldwasser, S., Vaikuntanathan, V., Zaharia, M. (2017). Splinter: practical private queries on public data. In: NSDI, pp. 299–313

Yao, A. C. (1986). How to generate and exchange secrets. *In Symposium on Foundations of Computer Science*, pages 162-167.