2010-01-01

Information Sharing Across Private Databases

Swapnil Suresh Samant
University of Texas at El Paso, sssamant@miners.utep.edu

INFORMATION SHARING ACROSS PRIVATE DATABASES

SWAPNIL S. SAMANT

Department of Computer Science

APPROVED:

Luc Longpré, Ph.D.

Vladik Kreinovich, Ph.D.

Raed Al Douri, Ph.D.

Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

INFORMATION SHARING ACROSS PRIVATE DATABASES

by

SWAPNIL S. SAMANT, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2010

Abstract

While the explosion of availability of information has become invaluable in modern society, it has also raised valid concerns about erosion of privacy. More and more, different entities are encouraged to share information in order to discover potential security risks, health pattern, or social behavior trends. However, some of this information is sensitive. Owners of information may be unwilling to share their whole databases with other entities, either to protect the privacy of the records, or because of the proprietary nature of the information.

We consider the following problem. How can we compute the intersection and the equijoin of databases owned by different parties, while revealing as little additional information as possible? One can address this challenge in different ways. Some protocols use trusted third parties, whereas some rely on the use of cryptographic techniques to achieve their goal without the use of a third party. An example of such an approach is a protocol proposed by Agrawal, Evfimievski and Srikant. In this thesis, we extend their protocol to apply to the intersection and the equijoin of three databases owned by three different entities. We also demonstrate the protocol with an implementation.

Table of Contents

Chapter 1

Introduction

While the explosion of availability of information has become invaluable in modern society, it has also raised valid concerns about erosion of privacy. More and more, different entities are encouraged to share information in order to discover potential security risks, health pattern, or social behavior trends. However, some of this information is sensitive. Owners of information may be unwilling to share their whole databases with other entities, either to protect the privacy of the records, or because of the proprietary nature of the information.

In this thesis, we are interested in the problem of how to achieve information integration when the part of the data is sensitive. Information integration can be achieved by information sharing. So far most of the work in this field has been related to information integration in databases belonging to a unique entity. Information privacy concerns arise when extending the approach to several entities. So, we are interested in how we can integrate information with minimal information sharing.

The notion of minimal information sharing was addressed by Agrawal et. all. [**?**]. In this work, the authors propose protocols for intersection, equijoin, intersection size and equijoin size of relational databases owned by two different parties. In this thesis we are addressing the minimal information sharing problem in the context of three parties by extending their protocols for intersection and equijoin of databases owned by three different parties.

1.1 Example Scenario

Lets consider a situation in which a corporation (Party A) has a list of individuals on which it wishes to run a background check with the local law enforcement (Party B) for any

convictions and with the immigration department (Party C) for any immigration related delinquency. Neither of the three parties would like to share all their information. Also, Party B and Party C would not like to share information with Party A just for their respective intersecting sets. We would like to design a protocol through which A learns information from B and C only when both the criteria are satisfied and A learns nothing other than the information that it had requested.

1.2 Outline

The rest of the thesis is organized as follows. In Chapter 2 we review some work in the field of minimal information sharing and review related terminology. In Chapter 3 we review the existing minimal information sharing protocols for intersection and equijoin operations across two parties. In Chapter 4 we develop minimal information sharing protocols for intersection and equijoin operations across three parties. In Chapter 5, we review some details about the implementation of these protocols. In Chapter 6, we show an example of commutative encryption and provide excerpts from the source code of our implementation.

Chapter 2

Previous Work

In this chapter we take a look at the previous work in the field of information sharing across private databases. Also, we review terminology relevant to information sharing protocols.

2.1 Previous Work

2.1.1 Trusted Third Party

The minimal information sharing problem for two parties can be solved if the two parties have a common trusted third party. While the two parties don't trust each other, they don't mind sharing their database with the trusted third party. The two parties can send their databases to the third party, who computes the intersection or the equijoin of the databases and sends the result to the requesting party. Confidentiality and integrity of communications can be protected with cryptographic encryption and digital signatures. Some of the approaches for implementing trusted third party based computation services are discussed in [**?**].

One drawback of this method is that it requires a total trust with respect to the third party protecting the confidentiality of the shared databases. It is usually impractical to find such a third party who is shared between the original antagonistic two parties.

2.1.2 Secure Multi-Party Computation

Secure Multi-Party Computation is a paradigm applying to the following situation. The goal of the computation is to compute a function $f(x, y)$ of two parameters x and y. The

value x is known to one party and the value y is known to the other party. We want to compute the result of the functions, but the two parties need to keep their respective inputs secret.

This problem is addressed by Naor and Nissim [**?**] and the book by Goldreich [**?**] has a chapter that discusses this problem in details. Yao [**?**] originally showed that multi-party computations can be accomplished by building a combinatorial circuit, and simulating that circuit using cryptographic tools. Later on, a variant of Yao's protocol with improved characteristics was proposed by Naor, Pinkas and Sumner [**?**].

Our minimal information sharing problem for two parties can be solved with secure multi-party computation as a special case. If we consider the two databases as inputs x and y respectively and f be the function implementing the intersection of the equijoin, the multi-party computation protocol solves the minimal information sharing problem.

One drawback of this approach is the high complexity of implementing a generalized secure multi-party computation protocol. The two-party computation approach involves constructing a boolean circuit that computes the function f, and somewhat simulate this circuit using cryptographic tools that hide the value of each bit in the circuit. The approach taken by Agrawal et. all. [**?**] is to develop a protocol that is substantially faster. They mention that in general, communication costs for circuits make them impractical for the minimal information sharing problem.

2.2 Terminology

2.2.1 Honest but Curious Behavior

In this approach the parties involved execute the protocol directly, without the involvement of any other party. We assume the parties are *honest but curious* [**?**]. This means that the parties execute the protocols exactly as specified, except that they record all communications and may attempt to learn additional information by further analysis of the

communications.

2.2.2 Operations

We consider the common database operations from [**?**]: intersection and equijoin. The *join* operation in databases is a well studied concept. In general, it combines records from different database tables. Literature covers different types of joins: inner join, outer join. The equijoin, as described below, is an inner join on *equal* condition.

We consider two parties: A, the receiver and B the sender. Let B have database D_B and let D_B have a table T_B. Also, let A have a database with table T_A. Both tables have a specific attribute $attr$ in their schema. The values of attribute $attr$ belong to a set V. V_B is the set of values that occur in $T_B.attr$ and V_A the set of values that occur in $T_A.attr$. For each $v1 \in V_B$, let $ext(v1)$ be set of all records in T_B where $T_B.attr = v1$, i.e., $ext(v1)$ is the extra information in T_B pertaining to $v1$. Size of the set V_A is represented as $|V_A|$.

2.2.3 Intersection

The intersection of A and B, represented as $A \cap B$, is a set of all values v such that $v \in V_A$ and $v \in V_B$.

2.2.4 Equijoin

The equijoin of A and B, represented as $A \bowtie B$, is a set of all values v such that $v \in A \cap B$ along with $ext(v1)$, $ext(v2)$ for all $v1 \in V_A$, $v2 \in V_B$ and $v1, v2 \in V_A \cap V_B$.

Chapter 3

Information Sharing Between Two Parties

In this chapter we describe existing protocols for information sharing across private databases for two parties. We consider the intersection and equijoin operations.

For all the protocols we consider two parties: R the receiver and S the sender.

3.1 Description of the Problem

The problem described in [**?**] is as follows:

Let there be two parties: R, the receiver and S the sender. Databases D_S and D_R belong to S and R respectively. For a database query Q across tables in the two databases, the result should be computed and returned to R without disclosing any other information to S.

It is not known how to achieve this, so the authors relax some of the criteria and change the problem description as follows:

There are two parties: R, the receiver and S sender. Databases D_R and D_S belong to R and S respectively. For a database query Q across tables from the two databases and some categories of information I, the result should be computed and returned to R without disclosing any other information to S except for the information contained in I.

3.2 Building blocks

3.2.1 Indistinguishability

Definition 1 (Indistinguishability) [**?**] *Let $\mathcal{D}_1$ assign, to every integer k, a distribution on the set $\{0,1\}^k$ of all k-bit numbers. Similarly, let $\mathcal{D}_2$ assign, to every integer k, a distribution on the set $\{0,1\}^k$ of all k-bit numbers. We say that the distribution $\mathcal{D}_1$ and $\mathcal{D}_2$ are* computationally indistinguishable *if for every polynomial-time algorithm $\mathcal{A}$ from binary numbers to truth values, and for any polynomial $p(k)$, for all sufficiently large k, we have*

$$P_r[\mathcal{A}_k(x)|x \sim \mathcal{D}_1] - P_r[\mathcal{A}_k(x)|x \sim \mathcal{D}_2] < \frac{1}{p(k)}$$

where $x \sim \mathcal{D}$ denotes that x is distributed according to $\mathcal{D}$, and $P_r[\mathcal{A}_k(x)]$ is the probability that $\mathcal{A}_k(x)$ returns true.

3.2.2 Commutative Encryption

Informally, we can define Commutative Encryption as a pair of encryption functions f and g such that $f(g(x)) = g(f(x))$. Thus by using commutative encryption we can ensure that any of the party cannot decrypt a value without the help of another party. Also, the order in which they encrypt a value does not affect the final result.

Definition 2 (Commutative Encryption) [**?**] *A* commutative encryption *$\mathcal{F}$ is a computable (in polynomial time) function f : Key $\mathcal{F}$ × Dom $\mathcal{F}$ → Dom $\mathcal{F}$, defined on finite sets, that satisfies all properties listed below. We denote $f_e(x) \equiv f(e,x)$, and use "$\in_r$" to mean "is chosen uniformly at random from".*

1. *Commutativity: For all $e, e' \in$ Key $\mathcal{F}$ we have*

$$f_e \circ f_{e'} = f_{e'} \circ f_e.$$

2. *Each f_e : Dom $\mathcal{F}$ → Dom $\mathcal{F}$ is a bijection.*

3. *The inverse f_e^{-1} is also computable in polynomial time given e.*

4. *The distribution of $\langle x, f_e(x), y, f_e(y)\rangle$ is indistinguishable from the distribution of $\langle x, f_e(x), y, z\rangle$ where $x, y, z \in_r$ Dom $\mathcal{F}$ and $e \in_r$ Key $\mathcal{F}$.*

3.2.3 Hash Function

According to [**?**], a hash function is a function that maps each input value $v \in V$ to a value x in the domain of the commutative encryption function $\mathcal{F}$. The hashed values should not collide and should appear random i.e., there should be no dependency between them that could help encrypt or decrypt one hashed value given the encryption of another.

3.3 Intersection

The intersection protocol proposed in [**?**] is as follows:

1. Both S and R apply hash function h to their sets:

   $X_S = h(V_S)$ and $X_R = h(V_R)$.

   Each party randomly chooses a secret key:

   $e_S \in_r$ Key $\mathcal{F}$ for S and $e_R \in_r$ Key $\mathcal{F}$ for R.

2. Both parties encrypt their hashed sets:

   $Y_S = f_{e_S}(X_S) = f_{e_S}(h(V_S))$ and

   $Y_R = f_{e_R}(X_R) = f_{e_R}(h(V_R))$.

3. R sends its encrypted set $Y_R = f_{e_R}(h(V_R))$ to S, reordered lexicographically.

4. (a) S sends its encrypted set $Y_S = f_{e_S}(h(V_S))$ to R, reordered lexicographically.

   (b) S encrypts each $y \in Y_R$ with key e_S and sends back to R the pairs

   $$\langle y, f_{e_S}(y)\rangle = \langle f_{e_R}(h(v)), f_{e_S}(f_{e_R}(h(v)))\rangle.$$

5. R encrypts each $y \in Y_S$ with e_R to get

$$Z_S = f_{e_R}(f_{e_S}(h(V_S))).$$

   Also, from pairs $\langle f_{e_R}(h(v)), f_{e_S}(f_{e_R}(h(v)))\rangle$ obtained in Step 4(b) for $v \in V_R$, it creates pairs $\langle v, f_{e_S}(f_{e_R}(h(v)))\rangle$ by replacing $f_{e_R}(h(v))$ with the corresponding v.

6. R selects all $v \in V_R$ for which $(f_{e_S}(f_{e_R}(h(v)))) \in Z_S$; these values form the set $V_S \cap V_R$.

In this protocol both parties learn the size of the others dataset whereas only the receiver learns the intersection of the dataset of both parties.

3.4 Equijoin

The equijoin protocol proposed in [**?**] is as follows:

Let V_S be the set of values that occur in $T_S.A$ and let V_R be the set of values that occur in $T_R.A$. For each $v \in V_S$, let $ext(v)$ be all records in T_S where $T_S.A = v$.

1. Both S and R apply hash function h to their sets:

   $X_S = h(V_S)$ and $X_R = h(V_R)$.

   R chooses a secret key $e_R \in_r$ Key $\mathcal{F}$ and S chooses two secret keys: $e_S, e'_S \in_r$ Key $\mathcal{F}$.

2. R encrypts its hashed set: $Y_R = f_{e_R}(X_R) = f_{e_R}(h(V_R))$

3. R sends to S its encrypted set Y_R, reordered lexicographically.

4. S encrypts each $y \in Y_R$ with both keys e_S and e'_S, and sends back to R 3-tuple

$$\langle y, f_{e_S}(y), f_{e'_S}(y)\rangle = \langle f_{e_R}(h(v)), f_{e_S}(f_{e_R}(h(v))), f_{e'_S}(f_{e_R}(h(v)))\rangle.$$

5. For each $v \in V_S$, S does the following:

   (a) Encrypts the hash $h(v)$ with e_S, obtaining $f_{e_S}(h(v))$.

(b) Generates the key for extra information using e'_S:

$\kappa(v) = f_{e'_S}(h(v))$.

(c) Encrypts the extra information:

$c(v) = K(\kappa(v), ext(v))$.

(d) Forms a pair $\langle f_{e_S}(h(v)), c(v)\rangle = \langle f_{e_S}(h(v)), K(f_{e'_S}(h(v)), ext(v))\rangle$. The pairs are then shipped to R in lexicographical order.

6. R applies $f_{e_R}^{-1}$ to all entries in the 3-tuples received at Step 4, obtaining

$$\langle h(v), f_{e_S}(h(v)), f_{e'_S}(h(v))\rangle$$

for all $v \in V_R$.

7. R sets aside all pairs $\langle f_{e_S}(h(v)), K(f_{e'_S}(h(v)), ext(v))\rangle$ received at Step 5 whose first entry occurs as a second entry in a 3-tuple $\langle h(v), f_{e_S}(h(v)), f_{e'_S}(h(v))\rangle$ from Step 6. Using the third entry $f_{e'_S}(h(v)) = \kappa(v)$ as the key, R decrypts $K(f_{e'_S}(h(v)), ext(v))$ and gets $ext(v)$. The corresponding v's form the intersection $V_S \cap V_R$.

8. R uses $ext(v)$ for $v \in V_S \cap V_R$ to compute $T_S \bowtie T_R$.

Chapter 4

Information Sharing Between Three Parties

4.1 Description of the Problem

Ideally we would like to achieve the following:

There are three parties: A, the receiver, B and C the two senders. Databases D_A, D_B and D_C belong to A, B and C respectively. For a database query Q across tables from all three databases, the result should be computed and returned to A without disclosing any other information to any of the three parties involved.

It is not known how to achieve this, so, similarly to the two-parties case, we relax some of the criteria and change the problem description as follows:

Let there be three parties A, the receiver, B and C the two senders. Databases D_A, D_B and D_C belong to A, B and C respectively. For a database query Q across tables from all three databases and some categories of information I, the result should be computed and returned to A without disclosing any other information to any of the three parties except for the information contained in I.

4.2 Composition: A Possible Approach to Solving the Problem

In principle, it is possible to solve the above problem by an appropriate composition of the two-parties algorithms. However, as we will show, if we use this composition, then more information is disclosed to each party than necessary.

4.2.1 Intersection

Party A executes the intersection protocols mentioned in Chapter 3 to compute the intersection between its dataset and datasets of party B and party C to get $V_A \cap V_B$ and $V_A \cap V_C$. Then party A computes the intersection between $V_A \cap V_B$ and $V_A \cap V_C$ to get $V_A \cap V_B \cap V_C$.

In this approach, the party A, in addition to the intersection $V_A \cap V_B \cap V_C$, also learns the intersections $V_A \cap V_B$ and $V_A \cap V_C$. It is desirable to avoid disclosing this additional information.

4.2.2 Equijoin

Party A executes the equijoin protocols mentioned in Chapter 3 to compute the equijoin between it's dataset and datasets of party B and party C to get $V_A \bowtie V_B$ and $V_A \bowtie V_C$. Then party A computes the equijoin between $V_A \bowtie V_B$ and $V_A \bowtie V_C$ to get $V_A \bowtie V_B \bowtie V_C$.

In this approach, the party A, in addition to the equijoin $V_A \bowtie V_B \bowtie V_C$, also learns the equijoins $V_A \bowtie V_B$ and $V_A \bowtie V_C$. It is desirable to avoid disclosing this additional information.

4.3 New Approach

Let us show that we can modify the protocols for intersection and equijoin from Chapter 3 so that A does not learn the intersection between its dataset and the datasets of B and

C individually.

The modified protocols are as follows:

For all the protocols we consider three parties A the receiver and B and C the senders. The new algorithms are as follows:

4.4 Intersection

1. All parties apply hash function h to their sets:

   $X_A = h(V_A)$,

   $X_B = h(V_B)$ and

   $X_C = h(V_C)$.

   Each party chooses a secret key:

   $e_A \in_r$ Key $\mathcal{F}$ for A,

   $e_B \in_r$ Key $\mathcal{F}$ for B and

   $e_C \in_r$ Key $\mathcal{F}$ for C.

2. Each party encrypts its hashed set with its encryption key:

   $Y_A = f_{e_A}(X_A) = f_{e_A}(h(V_A))$,

   $Y_B = f_{e_B}(X_B) = f_{e_B}(h(V_B))$ and

   $Y_C = f_{e_C}(X_C) = f_{e_C}(h(V_C))$.

3. B sends its encrypted set $Y_B = f_{e_B}(h(V_B))$ to C, reordered lexicographically.

4. (a) C sends its encrypted set $Y_C = f_{e_C}(h(V_C))$ to B, reordered lexicographically.

   (b) C encrypts each $y \in Y_B$ with key e_C and sends back to B the set,

   $Z_B = f_{e_C}(Y_B) = f_{e_C}(f_{e_B}(h(V_B)))$.

5. (a) B encrypts each $y \in Y_C$ with key e_B to get $Z_C = f_{e_B}(Y_C) = f_{e_B}(f_{e_C}(h(V_C)))$

(b) B computes the intersection of Z_B and Z_C to get $Z_{BC} = Z_B \cap Z_C$.

(c) B send the intersection Z_{BC} to A.

6. A sends its encrypted set $Y_A = f_{e_A}(h(V_A))$ to B, reordered lexicographically.

7. (a) B encrypts each $y \in Y_A$ with key e_B to get

$$Z_A = f_{e_B}(Y_A) = f_{e_B}(f_{e_A}(h(V_A)))$$

(b) B from tuples $\langle f_{e_A}(h(V_A)),\ f_{e_B}(f_{e_A}(h(V_A)))\rangle$ for corresponding $v \in V_A$ and sends these tuples to C.

8. (a) C encrypts the second part of tuples received from B in Step 7(b) with the key e_C to form the tuples $\langle f_{e_A}(h(V_A)),\ f_{e_C}(f_{e_B}(f_{e_A}(h(V_A))))\rangle$.

(b) C sends the tuples formed in Step 8(a) to A.

9. A encrypts each element of the intersection Z_{BC} it received from B in Step 5(c) with the key e_A to obtain $f_{e_A}(Z_B \cap Z_C)$.

10. A computes the intersection of $f_{e_A}(Z_B \cap Z_C)$ with the tuples $\langle f_{e_A}(h(V_A)),\ f_{e_C}(f_{e_B}(f_{e_A}(h(V_A))))\rangle$ it received in Step 8(b), using the second part of the tuple to obtain the intersection of the sets of V_A, V_B, V_C.

11. Using the first part of the tuple A can determine the actual values of the intersection elements.

At the end of the protocol:

A knows :

- $V_A \cap V_B \cap V_C$,
- $|V_B \cap V_C|$.

B knows:

- $|V_B \cap V_C|$,
- $|V_A|$,
- $|V_C|$.

C knows:

- $|V_A|$,
- $|V_B|$.

4.5 Equijoin

1. All parties apply hash function h to their sets:

   $X_A = h(V_A)$,

   $X_B = h(V_B)$ and

   $X_C = h(V_C)$.

   A chooses a secret key:

   $e_A \in_r$ Key $\mathcal{F}$ for A,

   B and C choose two secret keys:

   $e_B, e'_B \in_r$ Key $\mathcal{F}$ for B and

   $e_C, e'_C \in_r$ Key $\mathcal{F}$ for C.

2. (a) For each $v \in V_B$, B does the following:

   i. Encrypts the hash $h(v)$ with e_B to get

   $$Y_B = f_{e_B}(h(v))$$

   .

ii. Generates encryption keys for extra information using e'_B:

$$\kappa_B(v) = f_{e'_B}(h(v))$$

iii. Encrypts the extra information using the new keys generated:

$$E_B(v) = K(\kappa_B(v), ext_B(v))$$

iv. Forms pairs $\langle f_{e_B}(h(v)), K(\kappa_B(v), ext_B(v))\rangle$. These pairs are reordered lexicographically using the first element and sent to C.

(b) For each $v \in V_C$, C does the following:

i. Encrypts the hash $h(v)$ with e_C to get

$$Y_C = f_{e_C}(h(v)).$$

ii. Generates encryption keys for extra information using e'_C:

$$\kappa_C(v) = f_{e'_C}(h(v)).$$

iii. Encrypts the extra information using the new keys generated:

$$E_C(v) = K(\kappa_C(v), ext_C(v)).$$

iv. Forms pairs $\langle f_{e_C}(h(v)), K(\kappa_C(v), ext_C(v))\rangle$. These pairs are reordered lexicographically using the first element and sent to B.

3. B encrypts first part of the pair that it received from C using key e_B to obtain:

   $\langle f_{e_B}(f_{e_C}(h(V_C))), K(\kappa_C(V_C), ext_C(V_C))\rangle$.

4. C encrypts first part of the pair that it received from B using key e_C to obtain:

   $\langle f_{e_C}(f_{e_B}(h(V_B))), K(\kappa_B(V_B), ext_B(V_B))\rangle$

   C sends this pair to B.

5. B computes the intersection between the pairs that were computed in Step 3 and the pairs that it received from C in Step 4, using the first part of both pairs, to obtain:

   $\langle f_{e_B}(f_{e_C}(h(V_C))) \cap f_{e_C}(f_{e_B}(h(V_B))), K(\kappa_B(V_B), ext_B(V_B)), K(\kappa_C(V_C), ext_C(V_C))\rangle$

   These tuples are sent to A.

6. A encrypts its hashed set using key e_A and sends it to B, reordered lexicographically:

$$Y_A = f_{e_A}(X_A) = f_{e_A}(h(V_A)).$$

7. B encrypts each $y \in Y_A$ with keys e_B and e'_B to form a 3-tuple:

   $\langle f_{e_A}(h(V_A)), f_{e_B}(f_{e_A}(h(V_A))), f_{e'_B}(f_{e_A}(h(V_A)))\rangle$

   B sends this 3-tuple to C.

8. C encrypts first part of the 3-tuple it received from B in Step 7 using e'_C and the second part using e_C to obtain the 4-tuple:

   $\langle f_{e_A}(h(V_A)), f_{e_C}(f_{e_B}(f_{e_A}(h(V_A)))), f_{e'_B}(f_{e_A}(h(V_A))), f_{e'_C}(f_{e_A}(h(V_A)))\rangle$

   C sends this 4-tuple to A.

9. A applies $f_{e_A}^{-1}$ to the 4-tuple it received in Step 8 to obtain:

   $\langle h(V_A), f_{e_C}(f_{e_B}(h(V_A))), f_{e'_B}(h(V_A)), f_{e'_C}(h(V_A))\rangle$

10. Using the first element of the tuples in Step 5 and the second part of tuples computed in Step 9, A can compute the intersection of A, B and C. Also, using the third and fourth part of the tuple in Step 9, A can decrypt the extra information A received in Step 5.

At the end of the protocol:

A knows :

- $V_A \bowtie V_B \bowtie V_C$,

- $|V_B \cap V_C|$.

B knows:

- $|V_B \cap V_C|$,
- $|V_A|$,
- $|V_C|$.

C knows:

- $|V_A|$,
- $|V_B|$.

Chapter 5

Conclusion

We have modified the protocols for intersection and equijoin, presented in [**?**], to apply to databases owned by three different parties. Compared to using the composition of the two-party protocols within the three parties, less information is disclosed to each party involved. Our protocol for three parties still discloses some information other than the required information. It could be possible to modify the three party protocols to achieve perfect minimal information sharing. Another goal could be to generalize our approach to apply for more than three parties.

Chapter 6

Implementation

Implementation of the protocols described before requires the use of a hash function and a commutative encryption algorithm.

6.1 Hash Function

We have used SHA-256 for computing the hash values of database values.

6.2 Commutative Encryption

Similarly to what is suggested in [**?**], we used a modified version of RSA for commutative encryption. In RSA, two primes p and q are selected and $n = p * q$. Then, values d and e are selected such that $m^{de} \bmod n = m$. The public key contains d and n. The secret key contains e and n. Values p and q are needed to compute d and e.

In our implementation, all parties know p and q. Since encryption and decryption is always done by the same party, each party computes their own key pair and keep each half of the key pair private. Decryption is needed only in the equijoin protocol.

6.3 Example

Party A generates $p = 331$ and $q = 3$

$n = p * q, n = 993$

$\phi = (p - 1) * (q - 1) = 660$

A shares this p, q, n and ϕ with B.

A chooses it's encryption key e_A and decryption key d_A as follows:

$1 < e_A < \phi$ and e_A and ϕ share no divisors other than 1.

$e_A = 271$

The decryption key d_A is generated as $d_A = {e_A}^{-1} \bmod \phi$

$d_A = 151$

In a similar manner B generates its own encryption and decryption keys:

$e_B = 367, d_B = 223$

Let the message to be encrypted be $m = 5$

A encrypts the message m as:

$c_A = e_A(m) = m^{e_A} \bmod n$

$c_A = e_A(m) = 26$

B encrypts the message m that was encrypted by A: $e_A(m)$ as:

$c_{AB} = e_B(c_A) = {c_A}^{e_B} \bmod n$

$c_{AB} = e_B(c_A) = 749$

A decrypts c_{AB} using its decryption key d_A

$c_B = {c_{AB}}^{d_A} \bmod n$

$c_B = 296$

B decrypts c_B using its decryption key d_B

$m = {c_B}^{d_B} \bmod n$

$m = 5$

Now let B decrypt c_{AB} using its decryption key d_B

$c_A = {c_{AB}}^{d_B} \bmod n$

$c_A = 26$

Now let A decrypt c_A using its decryption key d_A

$m = {c_A}^{d_A} \bmod n$

$m = 5$

Thus, we see that the order of encryption is independent of the order of decryption.

6.4 Code Excerpt

The class used for RSA encryption is as follows:

```
import java.math.BigInteger;
import java.security.*;
import java.security.SecureRandom;
import java.util.Arrays;

public class RSA
{
    private BigInteger encKey;
    private BigInteger decKey;
    public BigInteger largePrimeP;
    public BigInteger largePrimeQ;
    private int bitLength = 1024;
    private BigInteger modulus;
    private final static BigInteger one      = new BigInteger("1");

    private void generateP()
    {
      SecureRandom random = new SecureRandom();
      largePrimeP = BigInteger.probablePrime(bitLength/2, random);
    }
    private void generateQ()
    {
      SecureRandom random = new SecureRandom();
      largePrimeQ = BigInteger.probablePrime(bitLength/2, random);
    }
```

```
private void generateKeys()
{
  SecureRandom random = new SecureRandom();
  BigInteger phi = (largePrimeP.subtract(one)).multiply(
                   largePrimeQ.subtract(one));
  modulus    = largePrimeP.multiply(largePrimeQ);
  encKey  = phi.probablePrime(bitLength/2,random);
  decKey = encKey.modInverse(phi);
}

public BigInteger[] encrypt(BigInteger[] hashValues)
{
  int numValues = hashValues.length;
  BigInteger[] encryptedArray = new BigInteger[numValues];
  for (int i =0; i<numValues;i++ )
  {
      encryptedArray[i] = hashValues[i].modPow(encKey, modulus);
  }
  return encryptedArray;
}

public BigInteger[] decrypt(BigInteger[] encValues)
{
  int numValues = encValues.length;
  BigInteger[] decArray = new BigInteger[numValues];
  for (int i =0; i<numValues;i++ )
  {
```

```
            decArray[i] = encValues[i].modPow(decKey, modulus);
        }
        return decArray;
    }

    RSA(BigInteger valP, BigInteger valQ)
    {
generateP();
generateQ();
        generateKeys();
    }
}
```

The source code for party A, the receiver, in the intersection protocol is as follows:

```
import java.io.*;
import java.math.BigInteger;
import java.security.*;
import java.security.SecureRandom;
import java.util.*;

public class InvolvedParty
{
    private static BigInteger[][] hashMappingA;
    private static BigInteger[] hashA;

    private static void readDatabase(String databaseAPath)
    {
        try
```

```
    {
        FileInputStream in = new FileInputStream(databaseAPath);
        BufferedReader br = new BufferedReader(new InputStreamReader(in));

        String strLine;
        int numValues;
        strLine = br.readLine();
        numValues = Integer.parseInt(strLine);
        hashMappingA = new BigInteger[numValues][2];

        for (int j = 0; j < numValues; j++)
        {
            strLine = br.readLine();
            hashMappingA[j][0] = new BigInteger(strLine);
        }
        in.close();
    }
    catch(IOException IOE)
    {
        System.out.println(IOE.toString());
    }
}

private static BigInteger[] readFiletoArray(String filePath)
{
    BigInteger[] outArray = null;
    try
    {
```

```
        FileInputStream in = new FileInputStream(filePath);
        BufferedReader br = new BufferedReader(new InputStreamReader(in));

        String strLine;
        int numValues;
        strLine = br.readLine();
        numValues = Integer.parseInt(strLine);
        outArray = new BigInteger[numValues];

        for (int j = 0; j < numValues; j++)
        {
            strLine = br.readLine();
            outArray[j] = new BigInteger(strLine);
        }
        in.close();
    }
    catch(IOException IOE)
    {
        System.out.println(IOE.toString());
    }
    return outArray;
}

private static void computeHash(BigInteger[][] inputValues, String hashFilePath)
{
    //BigInteger[] hashArray;
    int arrLen = inputValues.length;
    //hashArray =  new BigInteger[arrLen];
```

```
    StringBuffer hexString = new StringBuffer();
    hashA = new BigInteger[arrLen];

    try
    {
        MessageDigest algorithm = MessageDigest.getInstance("SHA-256");
        for (int j = 0; j < arrLen; j++)
        {
            algorithm.reset();
            algorithm.update(inputValues[j][0].toString().getBytes());
            byte messageDigest[] = algorithm.digest();
            hexString.delete(0, hexString.length());

            for (int i=0;i<messageDigest.length;i++)
            {
                hexString.append(Integer.valueOf(0xFF & messageDigest[i]));
            }
            hashA[j] = hashMappingA[j][1] = new BigInteger(hexString.toString());
        }
    }
    catch(NoSuchAlgorithmException nsae)
    {
        System.out.println(nsae.toString());
    }
    writeArrayToFile(hashA,hashFilePath);
}

private void waitforInput()
```

```
{
   System.out.println("Press any key to continue....");
   Scanner sc = new Scanner(System.in);
   while(!sc.nextLine().equals(""));
}

private static void writeArrayToFile(BigInteger[] arrayToWrite,String fileName)
{
    int arrLen = arrayToWrite.length;
    try
    {
        File outFile = new File(fileName);
        BufferedWriter out1=null;
        out1 = new BufferedWriter(new FileWriter(outFile,false));
        out1.write(String.valueOf(arrLen));
        out1.newLine();
        for (int j = 0; j < arrLen; j++)
        {
            out1.write(arrayToWrite[j].toString());
            out1.newLine();
        }
        out1.flush();
    }
    catch(IOException IOE)
    {
        System.out.println(IOE.toString());
    }
}
```

```
private static BigInteger[][] formTuple(BigInteger[] arr1, BigInteger[] arr2)
{
    BigInteger[][] tupleArray = new BigInteger[arr1.length][2];

    for (int i=0;i<arr1.length;i++)
    {
        tupleArray[i][0] = arr1[i];
        tupleArray[i][1] = arr2[i];
    }
    return tupleArray;
}

private static BigInteger[]
 extractArrayFromTuple(BigInteger[][] arrTuple, int index)
{
    BigInteger [] outArray = new BigInteger[arrTuple.length];
    for (int i=0;i<arrTuple.length;i++)
    {
        outArray[i] = arrTuple[i][index];
    }

    return outArray;
}

private static void
 writeTupleArrayToFile(BigInteger[][] arrayToWrite,String fileName)
{
```

```
int arrLen;
String strtoWrite;
if (arrayToWrite == null)
    arrLen = 0;
else
    arrLen = arrayToWrite.length;

try
{
    File outFile = new File(fileName);
    BufferedWriter out1=null;
    out1 = new BufferedWriter(new FileWriter(outFile,false));
    out1.write(String.valueOf(arrLen));
    out1.newLine();
    for (int j = 0; j < arrLen; j++)
    {
        strtoWrite = "";
        for (int i=0;i<arrayToWrite[j].length;i++)
        {
            strtoWrite = strtoWrite + arrayToWrite[j][i].toString() + ",";
        }
        out1.write(strtoWrite.substring(0, strtoWrite.length() -1));
        out1.newLine();
    }
    out1.flush();
}
catch(IOException IOE)
{
```

```
        System.out.println(IOE.toString());
    }
}

private static BigInteger[][] readFiletoTupleArray(String filePath, int numItems)
{
    BigInteger[][] outArray = null;
    //BigInteger[] lineArray = null;
    try
    {
        FileInputStream in = new FileInputStream(filePath);
        BufferedReader br = new BufferedReader(new InputStreamReader(in));

        String strLine;
        int numValues;
        strLine = br.readLine();
        numValues = Integer.parseInt(strLine);
        outArray = new BigInteger[numValues][numItems];

        for (int j = 0; j < numValues; j++)
        {
            strLine = br.readLine();
            String lineArray[] = strLine.split(",");
            for (int i=0;i<numItems;i++)
            {
                outArray[j][i] = new BigInteger(lineArray[i]);
            }
        }
```

```
        in.close();
    }
    catch(IOException IOE)
    {
        System.out.println(IOE.toString());
    }
    return outArray;
}

private static BigInteger[] computeIntersection
 (BigInteger[] arr1, BigInteger[] arr2)
{
    BigInteger[] intersectionArray = null;
    ArrayList<BigInteger> intersectionArrayList
     = new ArrayList<BigInteger>();
    for (int i=0; i < arr1.length; i++)
        for (int j=0; j< arr2.length; j++)
        {
            if (arr1[i].equals(arr2[j]))
            {
                intersectionArrayList.add(arr1[i]);
            }
        }
    intersectionArray =
     new BigInteger[intersectionArrayList.size()];
    intersectionArrayList.toArray(intersectionArray);
    return intersectionArray;
}
```

```
private static BigInteger[]
 getMatchingValuesFromTuple
 (BigInteger[] arrayToMatch,
 BigInteger[][] tupleToMatch,
 int tupleIndexToMatch,
 int tupleIndexToReturn)
{
    BigInteger[] matchingValuesArray = null;
    ArrayList<BigInteger> matchingValuesArrayList
     = new ArrayList<BigInteger>();
    for (int i=0;i<tupleToMatch.length;i++)
    {
    for(int j=0;j<arrayToMatch.length;j++)
    {
    if(arrayToMatch[j].equals(tupleToMatch[i][tupleIndexToMatch]))
    {
        matchingValuesArrayList.add(tupleToMatch[i][tupleIndexToReturn]);
    }
    }
    }
    matchingValuesArray= new BigInteger[matchingValuesArrayList.size()];
    matchingValuesArrayList.toArray(matchingValuesArray);
    return matchingValuesArray;
}

public static void main(String[] args)
{
```

```
String strBasePath = "C:\\3party\\";
InvolvedParty partyA =  new InvolvedParty();
BigInteger arrPQ[] = new BigInteger[2];

partyA.readDatabase(strBasePath + "databaseA.txt");
partyA.computeHash(hashMappingA, strBasePath + "hashA.txt");
RSA keyPair = new RSA();
BigInteger[] encValuesA =  keyPair.encrypt(hashA);
arrPQ[0] = keyPair.getP();
arrPQ[1] = keyPair.getQ();
partyA.writeArrayToFile(encValuesA,strBasePath + "hashAencA.txt");
partyA.writeArrayToFile(arrPQ, strBasePath + "primes.txt");
System.out.println("Preprocessing: Generated P and Q.");
System.out.println("Step 1: Generated hash values for A's Data.");
System.out.println("Step 1: A chooses an encryption key.");
System.out.println("Step 2: Encrypted hash values for A's Data.");
System.out.println("Step 6: Send Encrypted hash values to B.");
partyA.waitforInput();
BigInteger[] intBCencBencC = partyA.readFiletoArray(strBasePath
 + "intBCencBencC.txt");
BigInteger[] intBCencBencCencA = keyPair.encrypt(intBCencBencC);
BigInteger[][] hashAencAencBencCTuple =
 partyA.readFiletoTupleArray(strBasePath +
 "hashAencAencBencC.txt",2);
BigInteger[] hashAencA =
 partyA.extractArrayFromTuple
 (hashAencAencBencCTuple, 0);
BigInteger[] hashAencAencBencC =
```

```
            partyA.extractArrayFromTuple
            (hashAencAencBencCTuple, 1);
        BigInteger[] intABCencAencBencC =
            partyA.computeIntersection
            (intBCencBencCencA,
            hashAencAencBencC);
        partyA.writeArrayToFile(intABCencAencBencC,
            strBasePath + "intABCencAencBencC.txt");
        BigInteger[] intABChashAencA =
            partyA.getMatchingValuesFromTuple
            (intABCencAencBencC,
            hashAencAencBencCTuple,1,0);
        BigInteger[] intABChashA = keyPair.decrypt(intABChashAencA);
        BigInteger[] intABC =
            partyA.getMatchingValuesFromTuple
            (intABChashA,hashMappingA,1,0);
        partyA.writeTupleArrayToFile
            (hashMappingA, strBasePath + "hashMappingA.txt");
        partyA.writeArrayToFile
            (intABChashA, strBasePath + "intABCHashA.txt");
        partyA.writeArrayToFile
            (intABC, strBasePath + "intABC.txt");
        System.out.println("Step 9, 10 and 11.");
        System.out.println("Intersection written to intABC.txt");
    }
}
```

References

[AES03] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 86–97, Jun. 2003.

[AML01] S. Ajmani, R. Morris, and B. Liskov. A Trusted Third-Party Computation Service. Technical Report MIT-LCS-TR-847, MIT, 2001.

[Gol04] O. Goldreich, *The Foundations of Cryptography*, Vol. 2, Cambridge University Press, Cambridge, Massachusetts, 2004.

[NN01] M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. *Proceedings of the ACM Symposium on Theory of Computing*, pages 590–599, 2001.

[NPS99] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 129–139, Nov. 1999.

[Yao86] A. C. Yao. How to generate and exchange secrets. *Proceedings of the 27th Annual Symposium on Foundation of Computer Science*, pages 162–167, 1986.

Curriculum Vitae

Swapnil S. Samant was born on September 24, 1980 in Mumbai (Bombay), India. He earned his Bachelor's degree in Computer Engineering from Mumbai University, India.

Swapnil moved to El Paso, TX in the Fall of 2002. During the course of his graduate studies he worked as a Research Assistant at the Department of Civil Engineering and as a Systems Administrator at the Facilities Services department at the University of Texas at El Paso. For the past couple of years he has been working with Texas Transportation Institute in El Paso as a Software Applications Developer and plans to continue working for the same employer after his graduation.

Swapnil had always been interested in databases and got a chance to integrate this with privacy issues to come up with his Master's thesis.

Present address: 8601 Dyer St.,
El Paso, Texas 79904