Privacy Preserving Joins

Yaping Li [1], Minghua Chen [2]

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA 94720, USA
[1]yaping@eecs.berkeley.edu
[2]minghua@eecs.berkeley.edu

***Abstract*— In this paper, we design a system for mutually distrustful entities to perform privacy preserving joins, leveraging the power of a memory-limited secure coprocessor. Under this setting, we critique a questionable assumption in a previous privacy definition [1] that leads to unnecessary information leakage. We then remove the assumption and propose a new definition. Based on this definition, we propose three correct and provable secure algorithms to compute general joins of arbitrary predicates, by utilizing available cryptographic tools in a nontrivial way. We discuss different memory requirements of our proposed algorithms, and explore how to trade little privacy with significant performance improvement. In [2], we evaluate the performance of our algorithms by numerical examples. We also show the performance superiority of our approach over secure multi-party computation in [2].**

I. Introduction

We consider the problem of how entities compute arbitrary join functions over their data in a *privacy preserving* way such that no party learns more than what can be deduced from its input and output alone. Two motivating applications of privacy preserving joins in airport security and national healthcare were presented in [1].

One straight forward solution to perform privacy preserving join is to rely on a Trusted Third Party (TTP) to whom all parties submit their inputs. This TTP then computes the desired function and distributes the results. However, finding such a TTP to be unanimously trusted by all parties is usually difficult due to the high level of trust on the TTP.

Another approach is along the lines of the secure multi-party computation problem where parties collectively perform a computation over their data [3]. This approach assumes a low level of trust among the parties, but its computation and communication complexities are prohibitively high.

In this paper, we provide an alternative solution that strikes a balance between the level of the required trust and performance, by presenting a system that functions as a TTP, with a secure coprocessor being the only trusted component.

A secure coprocessor is a programmable general purpose computing environment that withstands physical and logical attacks. One example of such commercially available devices is IBM 4764 cryptographic coprocessors [4].

As pointed out by Agrawal et al. in [1], developing secure applications on a secure coprocessor faces two main challenges. Firstly, the limited memory capacities of secure coprocessors preclude the trivial solution of performing a computation inside a secure coprocessor with ever growing database size inputs and outputs. Secondly, the access pattern between a secure coprocessor and its host machine may serve as a covert channel to convey useful information to an observer sitting at the host machine. Our contributions are as follows:

- We critique a previous privacy definition in [1] for algorithms designed for secure coprocessors and point out the provably secure algorithms proposed in [1] leak information unnecessarily.
- We propose three provably correct and secure algorithms to compute general joins of arbitrary predicates and discuss trade-offs among the algorithms. In [2], we compare the their performance to the best known result from secure multi-party computation, and show that our algorithms are orders of magnitude faster.

Please refer to [2] for a complete review on the related work and complete details of our solutions and results.

II. Problem Formulation

1) System Overview: Our computation model consists of a service provider and multiple service requestors. A service provider is a host $\mathcal{H}$ to which the secure coprocessor $\mathcal{T}$ is attached. Service requestors are data owners and recipients of join results. The data owners send their data to the service provider which computes the join and distributes the results to the intended recipients. We assume authenticated and secure communication channels between the service provider and individual service requestors [5].

2) Threat Model: We distinguish two types of standard adversary models in this paper: honest-but-curious and malicious adversaries [3]. An honest-but-curious party follows the prescribed protocol properly, but may keep intermediate computation results, e.g. messages exchanged, and try to deduce additional information from them other than the protocol result. A malicious adversary may deviate arbitrarily from the protocol. Such deviation can be detected in a systemic and effective way, as described in [1] and [2]. As such, we assume honest-but-curious adversaries in the rest of our analysis.

3) Definition of Privacy Preserving: The authors in [1] define the safety of a join algorithm designed for a secure coprocessor as follows:

Definition 1: [Safety of a Join Algorithm] Assume we have database relations A, B, C and D, where $|A| = |C|$, $|B| = |D|$, A and C have identical schema, as do B and D. For any given N (representing the maximum number of tuples in B (resp. D) that match a tuple in A (resp. C)), let J_{AC}

(respectively, J_{CD}) be the ordered list of server locations read and written by the secure coprocessor during the join of A (resp. C) and B (resp. D). The join algorithm is safe if J_{AC} and J_{CD} are identically distributed.

The ultimate goal of a privacy preserving join algorithm is to reveal no information other than what can be inferred from the join result. We point out two problems in Definition 1 that cause algorithms satisfying the definition to leak more information than what we expect from this goal.

Firstly, the assumption of the number N permits join algorithms to leak the knowledge of N by definition. Such information leakage is not justified in [1] and might be critical. As shown in later sections, this assumption is also unnecessary.

Secondly, Definition 1 does not explicitly prescribe the join result, which allows algorithms to produce a superset of the real join result and leak information unnecessarily. We show in our technical report [2] that the provable safe algorithms in [1] under Definition 1 leak statistics of the number of joins per tuple in A. This knowledge would not be available to a recipient had it received only the real join tuples.

In terms of performance, The algorithms in [1] outputs a fixed amount of $N|A|$ join tuples regardless of the actual join result size. This is not ideal for most of the applications.

In this paper, we define privacy preserving joins with respect to honest-but-curious adversaries. We distinguish our definition from Definition 1 in three aspects: a) removal of the assumption of N, b) an explicit requirement of a join algorithm to compute exact join results with no additional padding, and c) extension to the multi-party scenario.

According to Definition 2, we say a join algorithm is privacy preserving if its access pattern is independent of the inputs.

Definition 2: [Privacy Preserving Joins]Let $f : D^m \mapsto D$ be an m-way join function where D is any database and $\mathcal{A}$ an algorithm that computes f. Assume arbitrary databases $\bar{A} = (A_1, \ldots, A_m)$ and $\bar{B} = (B_1, \ldots, B_m)$ where $|A_i| = |B_i|$, A_i and B_i have identical schemas respectively, and $|f(\bar{A})| = |f(\bar{B})|$. Let $J_{\bar{A}}$(resp. $J_{\bar{B}}$) be the ordered list of host locations a secure coprocessor reads and writes during the execution of $\mathcal{A}$ on $\bar{A}$ (resp. $\bar{B}$). Then $\mathcal{A}$ is **privacy preserving** if $J_{\bar{A}}$ and $J_{\bar{B}}$ are identically distributed.

III. Preliminaries

A. Assumptions and Notations

Assume J participating databases $D_1, \ldots, D_J$. Let $\mathcal{D} = D_1 \times \cdots \times D_J$, $L = |\mathcal{D}|$, $I = \{1, \ldots, L\}$, and iTuple be an element in $\mathcal{D}$. Let R denote the set of the join results with size S. For ease of exposition, we assume that $\mathcal{D}$ is materialized in $\mathcal{H}$'s memory. When joining J tuples, our proposed algorithms refer to the logical index of the corresponding iTuple in $\mathcal{D}$ instead of the indices of the J tuples in their respective tables.

Let a *decoy* be a string of a fixed pattern with the same length as a real join result. To avoid information leakage, $\mathcal{T}$ outputs a decoy when it needs to output something but there is no real join result. oTuple stands for an output tuple. An oTuple can be either a real join result or a decoy and a tuple can be either an iTuple or oTuple. Without loss of generality, We assume that an iTuple and oTuple have the same constant size. Let M in unit of tuples be the free memory of $\mathcal{T}$ and we assume that M is dedicated to the storage of oTuples. In our discussion of communication cost, we state the cost in terms of tuples.

B. Oblivious Sort

All our algorithms require removing the generated decoys in a privacy preserving way. An oblivious sorting algorithm sorts a list of encrypted elements such that no observer learns the relationship between the position of any element in the original list and the output list. To keep μ elements (e.g. join results) in a list of length ω, we apply oblivious sort on smaller portions of the list repeatedly to achieve high efficiency.

Firstly, we create a buffer of $\mu + \Delta$ elements, where Δ is the size of a swap area. We copy $\mu + \Delta$ elements from the source list to the buffer, and obliviously sort them to keep the selected elements in the top μ positions in the buffer. Since at most μ elements can be selected, the bottom Δ elements in the swap area can be overwritten. We copy another Δ elements from the source list, and overwrite the bottom Δ elements. We obliviously sort the buffer again to keep all the selected elements in the top positions.

This process is continued until all elements in the source list is processed. The top μ elements in the buffer are then the desired elements to keep. The total number of comparisons in the whole process, denoted as $\mathcal{C}_{(\omega,\mu)}(\Delta)$, is given by $\mathcal{C}_{(\omega,\mu)}(\Delta) = \frac{\omega-\mu}{\Delta}\frac{\mu+\Delta}{4}[\log_2(\mu+\Delta)]^2$. The number of element transfers is $4\mathcal{C}_{(\omega,\mu)}(\Delta)$.

Given ω and μ, we minimize the total number of element transfers between the secure coprocessor and the host, by carefully selecting Δ. The optimal Δ, denoted as Δ^*, can be found by solving the following optimization problem:

$$\Delta^* = \arg\min_{\Delta>0} \mathcal{C}_{(\omega,\mu)}(\Delta). \qquad (1)$$

IV. Privacy Preserving Joins

In this section, we present three join algorithms designed for a secure coprocessor $\mathcal{T}$, and discuss their communication cost and privacy preserving levels. Please refer to [2] for the algorithm details, communication cost and proof of correctness and security.

A. Algorithm A1: for Secure Coprocessors with Small Memory

Intuitively, if $\mathcal{T}$ always outputs a tuple regardless of whether there is a real join or not, then the communication patterns between $\mathcal{T}$ and $\mathcal{H}$ are independent of the contents of the participating databases. Consequently, an adversary does not learn any information on the contents of the participating databases, by observing the traffic between the $\mathcal{T}$ and $\mathcal{H}$. At the end, $\mathcal{T}$ obliviously filters out the decoys and outputs the real results.

B. Algorithm A2: for Secure Coprocessors with Large Memory

A significant portion of the communication cost of Algorithm **A1** comes from obliviously filtering out the $L - S$

decoys. One way to remove this portion of cost is to only write out the real results as follows. For participating databases $D_1, \ldots, D_J$, $\mathcal{T}$ sequentially reads `iTuples` in a pre-defined order. If the current `iTuple` results in a join result, $\mathcal{T}$ stores the join tuple in its memory.

The secure processor $\mathcal{T}$ writes out the stored M results only after scanning all L `iTuples`. $\mathcal{T}$ keeps repeating this process until it outputs all S join results. Consequently, M results are written out to the host machine every L tuples; the write cycle is L tuples and the write efficiency is $\frac{M}{L}$.

*C. Algorithm **A3**: Trading Privacy with Efficiency*

In Algorithm **A2**, the write cycle is L tuples and the write efficiency is $\frac{M}{L}$. When $M \ll S$, $\mathcal{T}$ spends a large number of write cycles to output all S join results, resulting in a high read cost. One way to improve the efficiency is to shorten the write cycle.

We assume that $\mathcal{T}$ knows L and S. In addition, $\mathcal{T}$ is able to randomly read every `iTuple` once and only once. This can be done by reading the tuples according to a random order generated on-the-fly by a Pseudo Random Number Generator [2].

To achieve better write efficiency, $\mathcal{T}$ processes the L `iTuples` in a random order in blocks of size n. While $\mathcal{T}$ is processing a block, it stores the join results in its memory. $\mathcal{T}$ writes all stored results to $\mathcal{H}$ after finishing processing one block. It repeats the process until completing all blocks. $\mathcal{T}$ then obliviously filters out the decoys and outputs the real results.

If the number of join results K generated for a block is no more than the memory size, i.e., $K <= M$, then $\mathcal{T}$ simply writes out M `oTuples` with $M - K$ decoys. In this case, $\mathcal{T}$ can achieve a write efficiency of M/n, which is better than that of Algorithm **A2**.

However, in the case where $K > M$, $\mathcal{T}$ will not able to output all the join results in one pass and will need to access this block again to output the missing results. These "salvage" actions may lead to information leakage and compromise the privacy preserving property of the join process. We refer to this situation as a *blemish* case.

It is certainly a design goal to minimize the probability of such a blemish case. Let $x(n)$ be a random variable denoting the number of join results in n randomly selected `iTuples`. The probability of $x(n) \leq M$ is the same as the probability of having at most M balls of a certain color out of n balls, which are selected from L balls in a non-replacement fashion. By a simple counting argument, this probability is given by

$$P[x(n) \leq M] = \frac{1}{\binom{L}{n}} \sum_{k=1}^{M} \binom{L-S}{n-k} \binom{S}{k} \qquad (2)$$

The probability for a blemish case to happen, i.e., at least one of the blocks contains more than M join results, is bounded by $\frac{L}{n} P[x(n) > M]$, the so called union bound. We denote this bound as $P_M(n)$. It is then crucial to make $P_M(n)$ acceptably small.

Let $1-\varepsilon$ be a privacy preserving parameter where $\varepsilon \in [0,1]$ can be chosen to be arbitrarily small. The optimal block size, denoted by n^*, is the minimum n that satisfies $P_M(n) < \varepsilon$. n^* can be found by solving the following problem:

$$n^* = \arg\min_{n>0} n \text{ subject to } P_M(n) < \varepsilon. \qquad (3)$$

The significance of n^* is that, if $\mathcal{T}$ processes `iTuples` by this optimal block size n^*, then a blemish case will happen with probability at most ε.

Following the above analysis, we propose Algorithm **A3** with a privacy preserving guarantee of probability $1-\varepsilon$ where $\varepsilon \in [0,1]$.

*D. Comparison of Algorithms **A1**, **A2** and **A3***

We compare levels of privacy preserving and communication costs of Algorithms **A1**, **A2**, and **A3** in Table I.

	Privacy Preserv- -ing Level	Communication Cost
A1	100%	$2L + \frac{L-S}{\Delta^*}(S+\Delta^*)[\log_2(S+\Delta^*)]^2$
A2	100%	$S + \lceil\frac{S}{M}\rceil L$
A3	$(1-\varepsilon)\times$ 100%	$2L + \lceil\frac{L}{n^*}\rceil M +$ $\frac{\lceil\frac{L}{n^*}\rceil M - S}{\Delta^*}(S+\Delta^*)[\log_2(S+\Delta^*)]^2$ (for the case $\varepsilon \neq 0$ and $M < S$)

TABLE I

LEVEL OF PRIVACY PRESERVING VS. COMMUNICATION COST.

In Table I, Algorithm **A1** and **A2** guarantee 100% privacy preserving, while Algorithm **A3** guarantees $(1-\varepsilon)\times$ 100% privacy preserving. However, as ε can be chosen to be arbitrarily small to meet practical needs, Algorithm **A3** is practically as secure as Algorithm **A1** and **A2**.

In general, Algorithm **A1** has the highest communication cost among the three algorithms. The communication cost of Algorithm **A2** mainly depends on the write efficiency $\frac{M}{L}$, while the communication cost of Algorithm **A3** mainly depends on the cost associated with oblivious filtering.

For large L and small M with respect to S, the write efficiency of an algorithm dominates the communication cost. Algorithm **A2** has a low write efficiency; hence, Algorithm **A3** outperforms Algorithm **A2** in terms of communication cost. In cases where M is close to S, Algorithm **A2**'s write efficiency M/L is high with respect to the optimal value $S/\ L$. Consequently, Algorithm **A2** might have less communication cost than Algorithm **A3**.

REFERENCES

[1] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li, "Sovereign joins," in *ICDE 2006*, Atlanta, Georgia, April 2006.

[2] Y. Li and M. Chen, "Privacy preserving joins," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-137, Nov 2007. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-137.html

[3] O. Goldreich, *Foundations of Cryptography*. Cambridge University Press, Aug. 2001, vol. 1: Basic Tools.

[4] IBM Corporation, "IBM 4764 Model 001 PCI-X cryptographic coprocessor," 2005.

[5] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. on Info. Thry.*, vol. IT-22, no. 6, pp. 644–654, November 1976.