# Privacy Preserving Joins on Secure Coprocessors

*Yaping Li*

Electrical Engineering and Computer Sciences
University of California at Berkeley



Technical Report No. UCB/EECS-2008-158

http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-158.html

December 14, 2008

Copyright 2008, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**Privacy Preserving Joins on Secure Coprocessors**

by

Yaping Li

B.S. (State University of New York at Stony Brook) 2001

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Doug Tygar, Chair
Professor Dawn Song
Professor David Aldous

Fall 2008

The dissertation of Yaping Li is approved:

Chair Date

Date

Date

University of California, BERKELEY

Fall 2008

**Privacy Preserving Joins on Secure Coprocessors**

Copyright 2008

by

Yaping Li

## Abstract

Privacy Preserving Joins on Secure Coprocessors

by

Yaping Li

Doctor of Philosophy in Engineering-Computer Science

University of California, BERKELEY

Professor Doug Tygar, Chair

The field of *privacy preserving joins* (PPJ) considers the question of how mutually distrustful entities share data in a privacy preserving way such that no party learns more than what can be deduced from its input and output alone. In my thesis, I focus on general join operations involving *arbitrary* predicates. Previous researchers have considered solutions using a *trusted third party* (TTP) and general secure multi-party computation. The former requires a high level of trust in the TTP by all entities. The latter is a well-known theoretical result of computing general joins in a privacy preserving way. However, the computation and communication complexity is normally too high for this approach to be practical.

In my thesis, I explore solutions that strike a balance between the level of required trust and performance. I propose solutions to compute privacy preserving joins efficiently through a trusted third party with secure coprocessors being the only trusted component. I present a rigorous definition of privacy preserving joins under this set-

ting, propose privacy preserving join algorithms and prove their correctness and security. I give explicit expressions for their computation costs, evaluate their performance, and show that the performance is superior than that of secure multi-party computation.

Professor Doug Tygar
Dissertation Committee Chair

To my parents.

# Contents

**List of Figures** **iv**

**List of Tables** **v**

**1 Introduction** **1**
1.1 Background and Motivation . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**2 Related Work** **8**
2.1 Protocol Based Approaches . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Secure Coprocessors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2.1 A Brief History . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2.2 Three Main Features . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2.3 Secure Coprocessor Aided Applications . . . . . . . . . . . . . . . 13

**3 Problem Formulation** **15**
3.1 Desiderata . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2 System Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3 Threat Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3.1 Detecting Malicious Behavior . . . . . . . . . . . . . . . . . . . . 18
3.3.2 Hiding Memory Access Patterns . . . . . . . . . . . . . . . . . . . 18
3.3.3 Authenticated Computation . . . . . . . . . . . . . . . . . . . . . 19
3.4 Design Principles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.4.1 A Straightforward, but Unsafe Algorithm . . . . . . . . . . . . . . 21
3.4.2 An Incorrect Fix . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.4.3 Principles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

**4 First Attempt on Privacy Preserving Joins on Secure Coprocessors** **23**
4.1 Assumptions and Notations . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.2 Definition of Privacy Preserving Joins . . . . . . . . . . . . . . . . . . . 25
4.3 Observations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.4 General Join Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.4.1 Algorithm 1 . . . . . . . . . . 27
4.4.2 A Variant of Algorithm 1 . . . . . . . . . . 32
4.4.3 Algorithm 2 . . . . . . . . . . 32
4.4.4 Parallelism . . . . . . . . . . 37
4.5 Equijoin Algorithms . . . . . . . . . . 37
4.5.1 False Starts . . . . . . . . . . 37
4.5.2 Privacy preserving Sort-Based Join (Safe) . . . . . . . . . . 39
4.6 Performance Analysis . . . . . . . . . . 42
4.6.1 $\gamma = 1$ . . . . . . . . . . 43
4.6.2 General Joins, $\gamma > 1$ . . . . . . . . . . 44
4.6.3 Equijoins, $\gamma > 1$ . . . . . . . . . . 44
4.6.4 Performance Relationship Summary . . . . . . . . . . 44
4.6.5 Comparison with Secure Function Evaluation . . . . . . . . . . 45

**5 Privacy Preserving Joins on Secure Coprocessors 47**
5.1 New Definition of Privacy Preserving Joins . . . . . . . . . . 48
5.1.1 Problems in Previous Definition . . . . . . . . . . 48
5.1.2 Proposed Definition . . . . . . . . . . 50
5.2 Preliminaries . . . . . . . . . . 51
5.2.1 Assumptions and Notations . . . . . . . . . . 51
5.2.2 Oblivious Sort . . . . . . . . . . 52
5.2.3 Generating Random Order . . . . . . . . . . 54
5.3 Privacy Preserving Joins . . . . . . . . . . 55
5.3.1 Algorithm **4** for Secure Coprocessors with Small Memory . . . . . . 55
5.3.2 Algorithm **5** for Secure Coprocessors with Large Memory . . . . . . 57
5.3.3 Algorithm **6** for Trading Privacy Preserving Level with Efficiency . . 58
5.3.4 Comparison of Algorithms **4**, **5** and **6** . . . . . . . . . . 66
5.3.5 Parallelism . . . . . . . . . . 68
5.4 Numerical Results . . . . . . . . . . 69

**6 Conclusions 73**

**Bibliography 76**

**Glossary 82**

**Key Notations 83**

# List of Figures

4.1 Performance Relationship . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

5.1 Communication cost of Algorithm **5** as a function of memory size $M$, under setting $L = 640,000$ and $S = 6,400$. . . . . . . . . . . . . . . . . . . . . . . . . 63
5.2 Communication cost of Algorithm **6** as a function of $\varepsilon$, under setting $L = 640,000$, $S = 6,400$, and $M = 64$. . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.3 Communication cost of Algorithm **6** as a function of $M$, under setting $L = 640,000, S = 6,400$ and $\varepsilon = 10^{-20}$. . . . . . . . . . . . . . . . . . . . . . . . . 65
5.4 Computational cost of Algorithm **6** (logarithmic scale) as a function of privacy preserving parameter $\varepsilon$, under different settings of $L, S$ and $M$. . . . . 71

# List of Tables

5.1 Level of privacy preserving vs. communication cost. . . . . . . . . . . . . . . 67
5.2 Different settings of $L, S$ and $M$ tested in the numerical experiments. . . . . 69
5.3 Communication costs of Algorithm **4**, **5** and **6**, for different settings of $L, S$ and $M$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

## Acknowledgments

I am deeply grateful to my advisor Doug Tygar, who guided me through the uncharted world of research. Doug taught me to seek the very essence of a research problem, his lightning speed of thinking always amazed me. I was deeply touched when he showed up sick in an early meeting and carried on an inspiring discussion when he could hardly speak from his illness. Besides research, I could always count on Doug for his insightful and practical views on life. On top of everything, he set an example for me of ethics and honesty.

I would like to thank my co-authors and colleagues from whom I learned a tremendous amount. Joe Hellerstein collaborated with me on the paper which was my first study on privacy problems in databases. Rakesh Agrawal, Dmitri Asonov, and Murat Kantarcioglu co-authored the paper which became the seed of this thesis. Dawn Song provided help when I needed it most in completing my study. Chris Karlof has been a constant source of support for my various research questions. Minghua Chen worked tirelessly with me on developing ideas and writing a paper that became an important part of this thesis.

I would like to thank my friends who made my journey a joyful one. Chris Karlof gave me countless happy days and helped me grow to be more open-minded, confident, and courageous. Li Yin and Weidong's shared insights on life helped ease my mind when I felt lost. Fang Yu and Guozheng Ge's warm friendship entered my life like a spring breeze. Shuheng Zhou helped me from thousands of miles away and shared my passion for dance.

Minghua, this thesis would not have been possible without you.

I owe eternal gratitude to my family. My loving parents supported me unconditionally both spiritually and financially through my years of study.

Material in this thesis was adapted from [4, 30, 31].

This work was supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422) and the following organizations: Cisco, ESCHER, HP, IBM, Intel, Microsoft, ORNL, Qualcomm, Sun and Symantec, and in part by the National Science Foundation through ITR Award IIS-0205647. The opinions in this thesis are my own and do not necessarily reflect the opinions of any funding sponsor or the US Government.

# Chapter 1

# Introduction

## 1.1 Background and Motivation

The field of *privacy preserving joins* (PPJ) considers the question of how mutually distrustful entities share data in a privacy preserving way such that no party learns more than what can be deduced from its input and output alone. Here are two motivating applications of privacy preserving joins:

- The Transportation Security and Administration and the FBI, have been developing a program to mine data about airline passengers to determine who should be selected for additional screening, or denied the right to board an airplane. Airport security checks use a do-not-fly list. Airlines and government agencies may wish to discover whether people are both on a passenger list and a list of potential terrorists, without revealing their respective lists.

- Epidemiological researchers may wish to study correlations between drug reactions and

some genetic sequences, which may require joining DNA information from a gene bank with patient records from various hospitals. A hospital disclosing patient information will likely violate Health Insurance Portability and Accountability Act [1], and it is desirable to access only matching sequences from the gene bank.

In a relational database, a join operation combines records from two tables, resulting in a new table. Two records are combined and inserted into the new table if the join predicate evaluates positively. Examples of join predicates are equality, greater than, and similarity predicates. Here is an example of similarity predicates. For set-valued attributes, the goal of Jaccard coefficient $> f$ is to find all set pairs where the ratio of the intersection size to union size is greater than a fraction $f$.

A straightforward solution to the PPJ problem relies on a Trusted Third Party (TTP) and requires trust in the TTP. Data owners submit their inputs to a TTP, and the TTP computes the desired join function and distributes the results. But trusting a TTP is problematic. In addition, a TTP is an attractive target, since it contains valuable information.

Another approach involves Secure Multi-party Computation (SMC) and is computationally expensive. SMC allows mutually distrustful parties collectively to perform a computation over their private data [16, 32, 34] such that no party learns more than what can be inferred from its own input and output of the computation alone. Generic protocols were designed to show the plausibility of such approach [16] and solutions to various constrained cases of the more general challenge of SMC were proposed [13, 28]. This approach assumes a low level of trust among the parties. General SMC computation is the

best known result for the problem of computing joins of *arbitrary* predicates in a privacy preserving way. However, the computation and communication complexity of this approach is normally too high for it to be practical.

Researchers have proposed distributed solutions for set intersection [13, 28] where the join predicate can only be equality. Joins involving arbitrary predicates, e.g. $<$, are important as well as fairly common in databases. In this thesis, we study PPJ that compute arbitrary join predicates.

Are there solutions that strike a balance between the level of required trust and performance? In this thesis, we explore answers to this question. We show that

> **Thesis Statement**
>
> Mutually distrustful parties can compute privacy preserving joins efficiently through a trusted third party with secure coprocessors being the only trusted component.

In particular, we present a secure network service for privacy preserving joins that functions as a TTP, with the only trusted component being a *secure coprocessor* [20, 37, 47] . The IBM 4758 [22] and its later generation the IBM 4764 cryptographic coprocessors [23] are examples of commercially available, tamper-responding secure coprocessors. We argue that on the one hand, the trust level required for a secure coprocessor is much lower than that for a completely trusted TTP. On the other hand, the complexity of computing a function on multiple parties' data in a privacy preserving way is much lower than that of the general SMC approach as in [32, 34].

The technical challenge in implementing such a service arises from the following:

- Secure coprocessors have limited capabilities. They rely on the server to which they are attached for disk storage or communication with other machines. They also have small memory (e.g. 4MB in an IBM 4758 and 64MB in an IBM 4764). The factors constraining the memory size are cost and heat dispensation. The trend towards consolidating the secure coprocessor functionality on a single chip also constrains the amount of memory as larger memories reduce the yield.
- While the internal state of a computation within the secure coprocessor cannot be seen from outside, the interactions between the server and the secure coprocessor can be observed.

Simply encrypting communication between the data providers and the secure processor is insufficient. The join computation needs to be carefully orchestrated such that the read and write accesses made by the secure coprocessor cannot be exploited to make unwanted inferences.

Careful orchestration of join computation in the face of limited memory has been a staple of database research for a long time. The goal in the past, however, has been the minimization of I/O to maximize performance. While the I/O minimization is still important, avoiding leakage through patterns in I/O accesses now becomes paramount in designing privacy preserving join algorithms.

## 1.2 Contributions

In this dissertation, we study the problem of privacy preserving joins. We leverage the power of secure coprocessors to design general joins algorithms involving arbitrary predicates. Chapter 2 introduces related work in this area. Chapter 3 formulates the problem of privacy preserving joins on secure coprocessors. Chapter 4 gives a first attempt to define privacy preserving joins on secure coprocessors and propose several algorithms that are provably secure under this definition. Chapter 5 shows how to remove an unnecessary assumption in the previous definition to arrive at a more general formulation.

Chapter 4 presents a secure information sharing service offering privacy preserving joins, built using off-the-shelf secure coprocessors. Examples of such commercially available devices are the IBM 4758 [22] and its later generation the IBM 4764 cryptographic coprocessors [23]. Our design satisfies the desiderata for such a service: we can do general joins involving arbitrary predicates across any number of databases; nothing apart from the result is revealed to the recipients; the only trusted component is the secure coprocessor; and the system is provably secure. We make the following contributions in this part:

- Formulation of the problem of computing join in which the goal is to prevent information leakage through patterns in I/O while maximizing performance.
- Articulation of the criteria for proving the security of a join algorithm in such an environment.
- Development of safe algorithms for different operational parameters and their cost analysis. Algorithm **1** (page 27) and **2** (page 32) are designed for secure coprocessor

with small and large memories respectively. Algorithm **3** (page 39) is a sort-based equijoin algorithm where the join predicate is equality.

In Chapter 4, we make an assumption which leaks a control amount of information to simplify our algorithm design. When joining two tables $A$ and $B$, we assume that the maximum number of tuples in $B$ that match a tuple in $A$ can be leaked. However, leaking such information does not satisfy the most stringent definition of a privacy preserving join: this information cannot be inferred from only the input and output of a protocol alone. In Chapter 5, we lift this assumption to arrive at a more general formulation of the problem. We make the following contributions in this part:

- We formulate the problem of privacy preserving joins in the setting of a secure coprocessor. We remove an assumption in our previous privacy definition in Chapter 4 to arrive at a more general definition.

- We propose three privacy preserving algorithms which compute general joins of arbitrary predicates and prove their correctness and security. Algorithm **4** (page 55) and **5** (page 57) guarantee 100% privacy preserving but have high computation cost. Algorithm **6** (page 58) is able to trade privacy preserving level $1 - \varepsilon$, where $\varepsilon \in [0, 1]$ is design parameter, with communication cost efficiently.

- We give explicit expressions for computation cost of the three proposed algorithms, evaluate their performance together with that of secure multi-party computation, and discuss the trade-off between them. We discuss the tradeoff between privacy preserving level and communication cost in Algorithm **6**.

- We show that our solution to the privacy preserving joins over arbitrary predicates is more efficient than the results from previous work. General multi-party computation is the best known result for the particular problem of computing joins of *arbitrary* predicates in a privacy preserving way. The computation and communication complexities of this approach are normally too high for them to be practical.
- We show that the trust level required for our proposed system with a secure coprocessor is much lower than that for a completely trusted TTP. However, the complexity of computing a function on multiple parties' data in a privacy preserving way is much lower than that of the general secure multi-party computation approach as in [32, 34].

# Chapter 2

# Related Work

## 2.1 Protocol Based Approaches

Privacy preserving joins are a constrained case of the more general challenge of secure multi-party computation (SMC), originally proposed by Andrew C. Yao in a 1982 paper [46]. In that publication, the millionaire problem was introduced: Alice and Bob are two millionaires who want to find out who is richer without revealing the precise amount of their wealth. Yao proposed a solution allowing Alice and Bob to satisfy their curiosity while respecting the constraints.

This problem and result gave way to a generalization called multi-party computation (MPC) protocols [18]. In an MPC problem, a given number of participants $p_1, p_2, \ldots, p_N$ each have a private data value, respectively $d_1, d_2, \ldots, d_N$. The participants want to compute $F(d_1, d_2, ..., d_N)$. An MPC protocol is called *secure* if information learned by each participant $i$ from the protocol could be derived from $d_i$ and $F(d_1, d_2, ..., d_N)$.

Secure multiparty computation protocols exist for all two-party computations as

well as for any multi-party computations with honest majority [16]. Most secure MPC protocols are computationally expensive and are thus mostly of theoretical significance.

To avoid the high computational cost, researchers have proposed various solutions for specialized functions. The problem of *privately computing* various set related operations has generated significant interests in the research community. The work most relevant to this thesis are the three following papers:

- Agrawal, Evfimievski, and Srikant (AES) propose two party protocols for set intersection, intersection size, equijoin, and equijoin size [5]. These protocols use commutative encryption and are provably safe for the honest-but-curious adversarial model. However, the protocols are only safe when an input database contains only unique elements, otherwise an attacker observing a protocol execution may learn statistics on the inputs.

- Freedman, Nissim, and Pinkas propose efficient two party protocols related to set intersection [13]. The proposed solutions include protocols for set intersection and approximating the size of the intersection. Freedman et. al also investigated other variants of the matching problem, including extending the protocol to the multi-party setting as well as considering the problem of approximate matching. These protocols are based on representing sets as roots of a polynomial and exploit the property of homomorphic encryption.

- Inspired by the results of Freeman et al [13], Kissner and Song propose efficient privacy preserving operations on multisets [28]. Kissner and Song point out that Freedman et al's solution does not use properties of polynomials beyond evaluation at given

points. Kissner and Song explore the power of polynomial representation of multisets, using operations on polynomials to obtain efficient, secure, and composable privacy-preserving multiset operations. These multiset operations include the union, intersection, and element reduction operations.

Both Freedman et al's protocol and Kissner et al's protocol are equality based protocols where the join predicate is equality. Joins involving *arbitrary predicates* are fairly common in databases. In this thesis, we study algorithms that compute arbitrary join predicates.

## 2.2 Secure Coprocessors

The solutions in the previous section are protocol-based. Various security and privacy related applications have used a special hardware, a secure coprocessor, in their design. We first give a brief history of secure coprocessors, then describe its main features, and finally elaborate on its various applications, in particular, applications related to privacy preserving computations.

### 2.2.1 A Brief History

In a distributed computing scenario, a common question arises: Why should one trust the computation carried out on a remote machine? Secure coprocessors were proposed as one of the solutions to this problem. This approach systematically amplifies small amounts of hardware security into broader system security to achieve the goal of remote trust.

The IBM 4758 and its later generation the IBM 4764 are two commercially available secure coprocessors. Sean W. Smith was one of the chief researchers who designed the IBM 4758. His book, *Trusted Computing Platforms: Design and Applications* [38], has the development history and technical details on the IBM 4758. Section 2.2.1 and 2.2.2 are adapted from this book.

We now give a very brief history of the research endeavors that preceded the development of the IBM secure coprocessors. Steven Kent's 1980 thesis [27] at MIT systematically explored the use of what he called *tamper-resistant modules (TRMs)* to protect external software. Steve White and Liam Comerford at IBM Watson then followed up the work by Kent and others, with ABYSS (a Basic Yorktown Security System) [43]. Steve White and colleagues then developed and refined the ABYSS design into a much more comprehensive system, Citadel [33, 44]. Some of the Citadel prototypes from IBM became the foundation of the Dyad project, built by Bennet Yee and J. D. Tygar in the early 1990s [40, 41, 47, 48]. Yee and Tygar implement four types of electronic commerce applications on top of a secure coprocessor. The implemented applications include: copy protections for software, electronic cash, electronic contracts, and secure postage.

### 2.2.2 Three Main Features

The IBM 4758 is a tamper-responding, sealed device that has a processor, memory storage, and fast crypto-support. The following three main features collectively enable trusted computation on a remote site: tamper detection/response, secure bootstrapping, and outbound authentication. I elaborate on the three features in the following sections.

### Tamper Detection/Response

An IBM 4758 contains measures to sense when tampering is occurring. Its memory is zeroized and the rest of the device disabled upon detection of tampering. Specifically, an IBM 4758 is wrapped with sensing grids which can detect tampering. The grids of conductors are monitored by circuitry that can detect changes in the properties (open, short, changes in conductivity) of the conductors. The conductors themselves closely resemble the material in which they are embedded—making discovery, isolation, and manipulation more difficult. These grids are arranged in several layers; the sensing circuitry can detect accidental connection between layers as well as changes in an individual layer. Upon detection of tamper, the memory is zeroized which destroys the primary secret of the device, the private key of an RSA or DSA key pair.

### Secure Bootstrapping

Secure bootstrapping allows a secure coprocessor to amplify small amounts of hardware security into broader system security. Specifically, the code blocks are organized into a hierarchy of decreasing privilege levels. A typical hierarchy is Miniboot, OS, and applications with Miniboot having the highest privilege. Each card is shipped with minimum software configuration (Miniboot only). The trust is rooted in trusting the manufacturer, the owner of the Miniboot. Operating system and/or application code is installed with the secure bootstrapping architecture which gradually extends the trust boundary to include OS and subsequently application software.

#### Outbound Authentication

Outbound Authentication (OA) gives the ability for the code running on a secure coprocessor to prove who it is to remote parties. At a high level, the outbound authentication scheme creates chains of signed certificates. Based on the theoretical framework developed by Sean W. Smith [38], when given chains of signed certificated, a relying party will be able to authenticate a particular software entity within a particular untampered platform. Specifically, the outbound authentication mechanism provided by the secure coprocessor ensures that it is indeed executing a known, trusted version of the application code, running under a known, trusted version of the OS, and loaded by a known, trusted version of the bootstrap code within a particular untampered platform.

### 2.2.3 Secure Coprocessor Aided Applications

Many applications have used secure coprocessors in their design. They include auditable digital time stamping [42], secure e-commerce [48], secure fine-grained access control [15], secure data mining [2], and private information retrieval [39]. Tiny trusted devices were used for secure function evaluation in [25].

Using secure coprocessors, Bhattacharjee et al. developed a federated architecture for privacy preserving collaborative data mining and analysis in [8]. The proposed architecture, where a secure coprocessor serves as the federator, allows mutually distrustful enterprises to mine their data in a privacy preserving way. In their setting, a typical computation consists of two steps at the federator. The first is to join the tables residing with different enterprises. The second step is to mine the resulting join table. Bhattacharjee et

al. mentioned in passing that care should be taken to handle joins in a privacy preserving way. However, they did not propose how to achieve this goal. As for the second step, they propose two light weight data mining operations. Their work is not a complete solution to privacy preserving data mining applications in the sense that it does not solve the privacy preserving join problem.

Our work complements [8] in the sense that we address the question of privacy preserving joins. We provide solutions to join various autonomous databases in a privacy preserving way via a secure coprocessor. The join result of our solutions may be further processed and used as the input to the lightweight data mining algorithms in [8].

# Chapter 3

# Problem Formulation

This chapter formulates the problem we address in this dissertation. In Section 3.1, we describe system requirements for privacy preserving joins on a secure coprocessor. In Section 3.2, we give an overview of the system. In Section 3.3, we introduce the threat model. Finally, in Section 3.4, we illustrate using classical nested loop joins some of the subtleties of the problem. This investigation enables us to distill the design principles underlying the proposed algorithms.

## 3.1 Desiderata

A system offering privacy preserving join service has the following desirable attributes:

- The system should be able to handle general joins involving arbitrary predicates. The national security application cited in Chapter 1 requires a fuzzy match on profiles. Similarly, the patient records spread across hospitals may require complex matching

in the healthcare application.

- The system should be able to handle multi-party joins. The recipient of the join result can be a party different from one of the data providers.

- The recipient should only be able to learn the result of the join computation. No other party should be able to learn the result values or the data values in someone else's input.

- The system should be provably secure. The trusted component should be small, simple, and isolated [6].

## 3.2 System Overview

Our goal is to build a privacy preserving join service with a secure coprocessor. Our computation model consists of a service provider and any number of service requestors. A service provider includes a secure coprocessor $\mathcal{T}$ and a host $\mathcal{H}$ to which $\mathcal{T}$ is attached. $\mathcal{H}$ is a general purpose computer which provides additional memory and disk space for $\mathcal{T}$. For simplicity, we refer to $\mathcal{H}$'s memory and disk as its memory in the rest of the dissertation. Service requestors are data owners and recipients of a join result which can be different from the data owners. 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 [12]. Similarly, any temporary value output by $\mathcal{T}$ to $\mathcal{H}$ is also encrypted. We assume that the join algorithms and the join predicates are known to the parties.

Without loss of generality, we will consider the two data provider cases in Chapter 4. Specifically, two parties $P_A$ and $P_B$ that have private relations $A$ and $B$ are participating in the privacy preserving join operation and the result $C$ is sent to the party $P_C$, which is not $P_A$ or $P_B$. In Chapter 5, our treatment of the problem is in terms of arbitrary numbers of data providers.

## 3.3 Threat Model

In our problem setting, the only trusted component is the secure coprocessor. All other components, including $\mathcal{H}$, are untrusted. We assume that no party (including $\mathcal{H}$) can observe the state of the computation inside $\mathcal{T}$ or tamper with the code loaded into it.

We distinguish two types of standard adversary models in this dissertation: Honest-but-curious and malicious adversaries [16]. We show how to reduce a malicious adversary to an honest-but-curious adversary later in this section. Consequently, our algorithm design in the rest of this dissertation will exclusively focus on the honest-but-curious model.

*Honest-But-Curious Adversaries.* In this model, parties follow 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 protocol is privacy preserving if no party may learn additional information other than what can be deduced from its input and output of the protocol.

*Malicious Adversaries.* In this model, parties may deviate arbitrarily from the protocol. In particular, we cannot hope to avoid parties (i) refusing to participate in the protocol, (ii) substituting an input with an arbitrary value, and (iii) prematurely aborting

the protocol. We show how to detect other types of deviations in our problem setting next.

### 3.3.1 Detecting Malicious Behavior

Reducing a malicious adversary to an honest-but-curious one is done by using cryptographic tools on $\mathcal{T}$ to detect malicious behaviors. In our problem setting, an honest-but-curious adversary may observe $\mathcal{H}$'s memory contents and the communications between $\mathcal{H}$ and $\mathcal{T}$ during program execution. A malicious adversary can additionally modify $\mathcal{H}$'s memory contents. In this thesis, we propose to use authenticated encryption (see Section 3.3.3) to detect memory tampering. Upon detection of such tampering, $\mathcal{T}$ terminates the program execution immediately.

### 3.3.2 Hiding Memory Access Patterns

We address three issues in preventing an honest-but-curious adversary from learning additional information by observing $\mathcal{T}$'s program execution. The first is to prevent timing attacks. One example of a timing attack is when an adversary can tell whether two tuples match or not if it observes that $\mathcal{T}$ takes a different amount of time when comparing two tuples that match and ones that do not. The standard approach to avoid timing attacks is to pad the variance in processing steps to constant time by burning CPU cycles as needed [15]. To keep the algorithm descriptions simple, we will not show the steps that burn CPU cycles.

The second issue is to hide $\mathcal{H}$'s memory content from an adversary. This is simultaneously achieved by using the authenticated encryption.

The third issue is to prevent an adversary from learning additional information

from $\mathcal{T}$'s memory access pattern to $\mathcal{H}$. Our definition for privacy preserving joins is built on hiding memory access patterns.

### 3.3.3 Authenticated Computation

Given that nothing but $\mathcal{T}$ is trusted, we have the challenge of validating the authenticity and protecting the secrecy of the computation done by $\mathcal{T}$.

We use the outbound authentication mechanism (Section 2.2.2) provided by the secure coprocessor to ensure that it is indeed executing a known, trusted version of the application code, running under a known, trusted version of the OS, and loaded by a known, trusted version of the bootstrap code.

We assume that $P_A$ and $P_B$ have signed a digital contract [15] prescribing what data can be shared and which computations are permissible. $\mathcal{T}$ holds a copy of the contract and serves as an arbiter of it. Contracts are kept encrypted at the server. At the start of a join computation, $\mathcal{T}$ authenticates the identities of $P_A$ and $P_B$ to ensure that the parties it is interacting with are indeed the ones listed in the contract. Then $\mathcal{T}$ sets up the symmetric keys to be used with $P_A$ and $P_B$ respectively. Each party prepends its relation with the contract ID and encrypts the two together as one message.

We require an encryption scheme that provides both message privacy and message authenticity. Such schemes are called authenticated encryption and include XCBC, IAPM, and OCB [14, 26, 35]. We choose OCB (which stands for "offset codebook") over the other two, as it requires the least number of block cipher operations ($m+2$ block cipher operations to encrypt (resp. decrypt) $m$ plaintext (resp. ciphertext) blocks). It is also provably secure: (a) an adversary is unable to distinguish OCB-outputs from an equal number of

random bits (privacy) and an adversary is unable to generate any valid $\langle$Nonce, Ciphertext, Authentication Tag$\rangle$ triple (authenticity). The indistinguishability from random strings implies that OCB is semantically secure [35], which ensures with high probability that duplicate tuples will be encrypted differently.

Encryption under OCB [35] requires an $n$-bit nonce $I$ where $n$ is the block size. The nonce would typically be an identifier selected by the sender. In OCB, two states, Offset and Checksum, are computed accumulatively as blocks are sequentially encrypted. The offset $Z[i]$ is used in encrypting and decrypting block $i$ where $Z[0] = E_k(I \oplus E_k(0^n))$, $Z[i] = f(Z[i-1], i)$ for $i > 0$ and some easily computable function $f(\cdot,\cdot)$. When encrypting a plaintext block $T[i]$, the ciphertext $C[i] = E_k(T[i] \oplus Z[i]) \oplus Z[i]$ for $1 \leq i < m$ where $m$ is the total number of message blocks. The final cipher block $C[m] = T[m] \oplus Y[m][\text{first}|T[m]|\text{bits}]$ where $Y[m] = E_k(\text{len}(T[m]) \oplus g(E_k(0^n)) \oplus Z[m])$, $\text{len}(T[m])$ the length of the final message block, and $g(\cdot)$ some easily computable function. The state Checksum $= T[1] \oplus \cdots \oplus T[m-1] \oplus C[m]0^* \oplus Y[m]$ and the tag $T = E_k(\text{Checksum} \oplus Z[m])[\text{first } \tau \text{ bits}]$ where $C[m]0^*$ represents padding the last cipher block to the block size. The first $\tau$ bits are the authentication tag $T$. The nonce $I$ and the ciphertext $C[1] \cdots C[m-1]C[m]T$ are transferred to the recipient.

When decrypting a ciphertext block $C[i]$, the plaintext $P[i] = E_k^{-1}(\bar{Z}[i] \oplus C[i]) \oplus \bar{Z}[i]$ for $1 \leq i < m$ where $\bar{Z}[i]$ is computed from the received nonce. Let $Y[m] = E_k(\text{len}(C[m]) \oplus g(E_k(0^n)) \oplus \bar{Z}[m])$. $P[m] = C[m] \oplus Y[m][\text{first } |C[m]| \text{ bit}]$. Checksum $= P[1] \oplus \cdots \oplus P[m-1] \oplus C[m]0^* \oplus Y[m]$. Let $T' = E_k(\text{Checksum} \oplus Z[m])[\text{first } \tau \text{ bits}]$. If $T' = T$, then accept the message, otherwise reject.

Since we use authenticated encryption, an adversary who does not know the key cannot impersonate $P_A$ or $P_B$, nor can it tamper with the encrypted tuples in any way that will not be detected. Similarly, for communication of result from $\mathcal{T}$ to $P_C$.

Thus, the only vulnerability that an adversary can hope to exploit is the pattern in the interactions between $\mathcal{H}$ and $\mathcal{T}$. Our algorithms are designed to thwart the adversary from learning anything by observing this interaction.

## 3.4 Design Principles

We first present two straightforward, but unsafe, adaptations of the classical nested loop join algorithm. We discuss them as they help derive the design principles underlying our proposed algorithms.

### 3.4.1 A Straightforward, but Unsafe Algorithm

Here is a straightforward adaptation of the classical nested loop join algorithm. $\mathcal{T}$ first obtains an encrypted tuple of $A$ by sending a read request to $\mathcal{H}$ and decrypts the tuple inside its memory. $\mathcal{T}$ then reads a tuple of $B$, decrypts it, and compares it with the decrypted tuple of $A$. If the match succeeds, $\mathcal{T}$ encrypts the result tuple and outputs it to $\mathcal{H}$ to write to disk. The above step is repeated for the rest of the tuples of $B$ and then the procedure is repeated for the rest of the tuples of $A$.

Unfortunately, this straightforward adaptation is not safe, although the input as well as output values remain encrypted outside of $\mathcal{T}$. An adversary (e.g., $\mathcal{H}$ colluding with $P_A$ who does not receive the join result) can easily determine which encrypted tuples of

$A$ joined with which tuples of $B$, simply by observing whether $\mathcal{T}$ outputted a result tuple before the read request for the next $B$ tuple. If this information becomes available to $P_A$, then $P_A$ can determine which of its tuples have a match with a tuple of $P_B$.

### 3.4.2 An Incorrect Fix

What if $\mathcal{T}$ waits for $M$ tuples (or a random number of tuples $< M$) to be created and then outputs them in a block? Unfortunately, the adversary can still estimate the distribution of matches. In addition, the adversary can also launch timing attacks; since encryption takes significant time, it can determine whether there was a match by monitoring inter-request times for $B$ tuples.

### 3.4.3 Principles

We can derive two important principles from the above discussion:

1. *Fixed Time* The evaluation of the join predicate and the composition of tuples should take same time irrespective of whether the comparison yields a match.

2. *Fixed Size* There should not be any difference in the amount of output produced irrespective of whether the comparison yields a match.

# Chapter 4

# First Attempt on Privacy Preserving Joins on Secure Coprocessors

In this chapter, we attempted an initial definition of privacy preserving joins on secure coprocessors. We proposed three algorithms which are provably secure with respect to our definition and analyzed their performance. However, these algorithms are limited in the sense that they leak a controlled amount of information by definition. We modify our definition in the next chapter and arrive at a more general formulation.

The rest of the chapter is organized as follows. In Section 4.1, we give the simplifying assumptions and notations. In Section 4.2, we define the correctness criteria for proving the safety of the join algorithms. In Section 4.3, we make some observations that apply to all proposed algorithms.

In Section 4.4, we provide two provably safe algorithms for general join in which the matching predicate can be an arbitrary function. They offer a range of performance trade-offs under different operating parameters.

Section 4.5 is devoted to the study of equijoins. Surprisingly, adaptations of classical sort-merge join or hash join turn out to be unsafe. We then provide a safe algorithm.

In Section 4.6, we analyze the performance characteristics of the proposed algorithms.

## 4.1 Assumptions and Notations

To simplify exposition, we will assume that the tuples of $A$, $B$, and $C$ are of the same size and that the free memory of the secure processor can hold at most $M + 2$ such tuples. Note that we need to be able to hold at least two input tuples in memory during the join processing and expressing memory size as $M + 2$ simplifies cost expressions. $N$ is the maximum number of tuples from $B$ that match a tuple from $A$. Our algorithms have been designed to handle the general case where $M < N$. We also assume that $M$ is much smaller than $|A|$ or $|B|$.

We will omit from the algorithms the details of the communication between $\mathcal{H}$, $P_A$, $P_B$, and $P_C$. Assume that $P_A$ and $P_B$ have sent their encrypted relations $A$ and $B$ respectively to $\mathcal{H}$, who has stored them on its local disk. Similarly, $\mathcal{T}$ writes the encrypted join result to $\mathcal{H}$'s disk (invoking the server process running on $\mathcal{H}$), which $\mathcal{H}$ then sends to $P_C$. The algorithms will describe the code executed by $\mathcal{T}$.

We will indicate a transfer of data from $\mathcal{T}$ to $\mathcal{H}$ by prepending the operation

with the keyword `put`; the keyword `get` will indicate a transfer from $\mathcal{H}$ to $\mathcal{T}$. We will use *encrypt*$(\cdot)$ and *decrypt*$(\cdot)$ to denote the encryption and decryption functions respectively. We will ignore the use of keys in these functions. We assume fixed size tuples and that the server knows their size.

We do not discuss issues such as schema discovery and schema mappings. We assume schemas can be shared. The design presented in [3] can be used for this purpose.

## 4.2 Definition of Privacy Preserving Joins

We discussed in Section 3.3 that an adversary can only infer information from the pattern of interactions between the server and the secure coprocessor. Therefore, for an algorithm running on a secure coprocessor to be safe, it must not reveal any information from its accesses to the server. Building upon the definitions in [19], we formalize this intuition as follows:

**Definition 1** (Privacy Preserving Join Algorithms)**.** *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* $N$ *(Section 4.1), 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 privacy preserving if* $J_{AC}$ *and* $J_{CD}$ *are identically distributed.*

If the access pattern is independent of the underlying data then the access pattern will be identical for all the relations that satisfy the conditions given in Definition 1. Therefore, to prove that an algorithm is safe, we will show that the access pattern does not

depend on the data in the underlying relations.

## 4.3 Observations

The following remarks apply to all the proposed algorithms.

**Safeguarding Against Timing Attacks** The standard approach for avoiding timing attacks is to pad the variance in processing steps to constant time by burning CPU cycles as needed [15]. To keep the algorithm descriptions simple, we will not show the steps that burn CPU cycles in any of the algorithms.

**Decoys** Our algorithms encrypt a decoy plaintext and output it if necessary to prevent information leakage. Decoys are decrypted and filtered out by the recipient. They may take the form of a fixed string pattern. The semantically secure encryption generates indistinguishable cipher texts from multiple encryptions of the same plain text, which can be recovered from any one of them at the time of decryption [35].

**Setting $N$** In some applications, $N$ might be known a priori. A safe estimate for $N$ would be $|B|$ but it can hurt performance, particularly if the actual value is much smaller. Guessing $N$ too small and rerunning the algorithm if the actual value happens to be larger leaks information. A safe way to compute exact $N$ would be to run a nested loop join, but without outputting any result tuple. Note that this preprocessing step does not leak information.

**Cost Analysis** We will compare the cost of our algorithms in terms of the number of tuple transfers between the secure processor and the server, assuming disk I/Os can be pipelined with the transfers between the server and the secure coprocessor. Every time the secure processor gets a tuple from the server, it is decrypted. Similarly, a tuple is encrypted before the secure coprocessor outputs it to the server. Thus, the number of transfers between the coprocessor and server also reflects the total number of encryption and decryption operations.

## 4.4 General Join Algorithms

We present two algorithms for general joins in which the join predicate is specified through an arbitrary *match()* function. A join in this general setting requires every tuple of the outer relation to be compared with every tuple of the inner relation [9].

### 4.4.1 Algorithm 1

Algorithm 1 has been designed for secure coprocessors with small memories. It outputs an encrypted join tuple if there is a match and an encrypted decoy of the same size otherwise. Because of semantically secure encryption, all the decoy tuples will look different and an adversary cannot decipher whether there was a match or not.

Using the above strategy, a straightforward algorithm will generate an output of size $|A||B|$. Algorithm 1 generates $N|B|$ output tuples by cleverly using $scratch[]$ array of size $2N$ allocated in $\mathcal{H}'$s memory. In a pass over $B$, after processing every $N$ tuples (a round), $\mathcal{T}$ obliviously sorts $scratch[]$ giving lower priority to decoy tuples. Consequently,

**Algorithm 1** For Secure Coprocessors with Small Memory

```
for each tuple a ∈ A do
  put 2N encrypted decoy tuples to scratch[];
  a_T = decrypt(get a);
  i = 0;
  for each tuple b ∈ B do
    b_T = decrypt(get b);
    if match(a_T, b_T) then
      put scratch[(i mod N) + N] =
        encrypt(join(a_T, b_T));
    else
      put scratch[(i mod N) + N] =
        encrypt(join(decoy, decoy));
    end if
    i = i + 1;
    if i mod N == 0 then
      Obliviously sort scratch[] giving lower priority to decoy tuples;
    end if
  end for
  if i mod N ≠ 0 then
    Obliviously sort scratch[] giving lower priority to decoy tuples;
  end if
  Request H to write first N of scratch[] to disk;
end for
```

any joined tuples in the last $N$ location of *scratch*[] will be moved to the first locations of *scratch*[]. However, because of sorting being oblivious, an adversary cannot know the boundary. After the last round, the first $N$ locations of *scratch*[] will contain only the result tuples and possibly some decoy tuples and the server writes them to disk.

**Oblivious Sorting** 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. Oblivious sorting of a list of $n$ elements using the Bitonic sort algorithm proceeds in stages [7]. Assuming $n$ is a power of 2, at each stage, the $n$ elements are divided into sequential groups of size $2^i$ where $i$ depends on the stage. Within each group, an element is compared with one that is $2^{i-1}$ elements away. Each pair of the encrypted elements is brought into the secure coprocessor, decrypted, compared, and re-encrypted before they are written out to their original positions, and possibly swapped. There are a total of approximately $\frac{1}{2}(\log_2 n)^2$ stages and $\frac{1}{2}n$ comparisons at each stage. Therefore, the cost of oblivious Bitonic sort is $\frac{1}{4}n(\log_2 n)^2$ comparisons and $n(\log_2 n)^2$ element transfers between the secure coprocessor and the server.

**Encryption** Since both $A$ and $B$ are accessed sequentially, they can be encrypted using the procedure described in Section 3.3.3. However, oblivious sorting of *scratch*[] requires non-sequential access to its tuples. We next describe the encryption of tuples in *scratch*[] in the OCB mode. For simplicity, assume that the size of a tuple is the same as the length of one cipher block.

After an oblivious sort, the first $N$ locations of the array *scratch*[] contain the

joined tuples that $\mathcal{T}$ has seen so far and possibly some decoy tuples; the last $N$ locations contain $N$ decoy tuples. Conceptually, the first $N$ tuples in *scratch*[] and the $N$ output tuples from the next round will be treated as one message.

At the end of the last stage of an oblivious sort, $\mathcal{T}$ keeps the following two states for continuing encryption in the next round: an offset $Z[N]$ and a Checksum $= T[1]\oplus\cdots\oplus T[N]$ where $T[i]$ are the plaintext of tuples in the first $N$ locations in *scratch*[]. In the next round, $\mathcal{T}$ encrypts the $N$ output tuples as message blocks $T[N+1]$ through $T[2N]$ and computes a tag for the entire message.

We next describe how to perform encryption and decryption when obliviously sorting *scratch*[]. $\mathcal{T}$ generates a fresh nonce for re-encrypting output tuples at each stage of the Bitonic sort. When comparing a pair of tuples, $\mathcal{T}$ decrypts *scratch*$[i]$ and *scratch*$[j]$, compares them, then re-encrypts them with offsets $\bar{Z}[i]$ and $\bar{Z}[j]$ computed from the fresh nonce for the current stage. $\mathcal{T}$ then computes Checksum = Checksum $\oplus P[i] \oplus P[j]$. At the end of a stage, if $\mathcal{T}$ accepts the $2N$ tuples it just decrypted, it continues to the next step, otherwise, it terminates the computation. After re-encrypting the last tuple for a stage, $\mathcal{T}$ computes the tag for the $2N$ tuples it just encrypted and keeps this tag in the memory for the authentication check at the next stage.

We next investigate the extra cost of encrypting $n$ tuples (elements) non-sequentially. As before, the size of a tuple is the same as the length of one cipher block. In Bitonic sort, an element is compared with one that is half the distance away in the same group. In order to decrypt the $(n/2+1)^{th}$ element without sequentially decrypting every tuple before it, we apply the function $f(\cdot,\cdot)$ $i = n/2$ times to obtain $Z[i+1] = f(\cdots f(f(Z[1],2),3)\cdots,i+1)$.

Then the second element is compared with the $(n/2+2)^{th}$ element and $Z[2] = f(Z[1], 2)$ and $Z[i+2] = f(Z[i+1], i+2)$, and so on. Thus, within the same group, no additional application of $f(\cdot,\cdot)$ is required except for the first pair. Hence, at a stage in which there are $j$ groups of size $i$ where $ij = n$, the total additional $f(\cdot,\cdot)$ applications is $\frac{1}{2}ij = n/2$. Since there are $\frac{1}{2}(\log n)^2$ stages in Bitonic sort, a total of additional $\frac{n}{4}(\log n)^2$ applications of $f(\cdot,\cdot)$ are needed for sorting a set of $n$ elements compared to sequentially encrypting $n$ elements at each stage.

**Correctness (Proof Sketch)**

For every tuple of $A$, the algorithm goes through the same number of rounds ($\lceil |B|/N \rceil$). In every round, $\mathcal{T}$ outputs the same amount ($N$ tuples) to the same locations of *scratch*[]. After all the rounds are over, $\mathcal{T}$ obliviously sorts *scratch*[], which accesses *scratch*[] independent of the underlying data. Finally, the first $N$ locations of *scratch*[] are always accessed for writing the result tuples to disk. Thus, Definition 1 is satisfied.

**Cost Analysis**

During the execution of Algorithm 1, $\mathcal{T}$ gets $|A|$ tuples from $A$ and $|A||B|$ tuples from $B$. It outputs $2N$ decoy tuples for each $a \in A$, for a total of $2|A|N$ decoy tuples. For each comparison of $a \in A$ and $b \in B$, $\mathcal{T}$ outputs a result tuple, for a total of $|A||B|$ output tuples. For every $a \in A$ and every block of $N$ tuples in $B$, $\mathcal{T}$ obliviously sorts $2N$ tuples, which leads to transferring a total of $2|A||B|(\log_2(2N))^2$ tuples into and out of $\mathcal{T}$'s memory. Finally, the server writes $N|A|$ tuples to disk.

Thus, in terms of the number of tuple transfers in and out of $\mathcal{T}$'s memory, the

complexity of Algorithm 1 is:

$$|A| + 2N|A| + 2|A||B| + 2|A||B|(\log_2(2N))^2.$$

### 4.4.2 A Variant of Algorithm 1

Consider a variant of Algorithm 1 that also matches every tuple of $B$ with every tuple of $A$, but does not use *scratch*[]. Instead, for a given tuple of $A$, it writes $|B|$ tuples, result or decoys, to the memory of $\mathcal{H}$. At the end of the pass, $|B|$ output tuples are obliviously sorted giving lower priority to decoy tuples and only the first $N$ tuples are saved.

The number of tuple transfers in and out of $\mathcal{T}$'s memory will now be $|A|+2|A||B|+|A||B|(\log_2|B|)^2$. Assume $|A| = |B|$, and define $\alpha$ to be $N/|B|$. Clearly, Algorithm 1 outperforms this variant for small values of $\alpha$; we do not discuss it further.

### 4.4.3 Algorithm 2

Algorithm 2 has been designed for secure coprocessors with larger memories. It optimizes the use of the memory of the secure processor to reduce the number of output tuples, while not leaking any information in the process.

Define $\gamma = max(1, \lceil N/(M - \delta)\rceil)$. Here $\delta$ represents the small amount of memory needed for data structures other than those needed for holding the input and result tuples (e.g. counters). For every tuple $a$ of $A$, $\mathcal{T}$ reads entire $B$ a total of $\gamma$ times to find all the matches for $a$. Conceptually, imagine partitioning the tuples from $B$ that match $a$ into $\gamma$ groups of $\lceil N/\gamma\rceil$ tuples each. During pass $i$ over $B$, $\mathcal{T}$ computes the $i^{th}$ group of the matched tuples and outputs them to $\mathcal{H}$ at the end of the pass. Note that unlike the

standard blocked nested loop join in which the input relations are partitioned in chunks of fixed size, the partitioning here is over the matched tuples.

Recall that $N$ is the maximum number of $B$ tuples that match with any of the tuples in $A$. Clearly, there may be tuples in $A$ that match with less than $N$ tuples of $B$. In that case, when $\mathcal{T}$ runs out of real join tuples, it outputs an appropriate number of decoy tuples.

Since both $A$ and $B$ are accessed sequentially and the output tuples are also produced sequentially, they can be encrypted using the procedure described in Section 3.3.3.

**Correctness (Proof Sketch)**

Every tuple of $A$ causes $\gamma$ passes over $B$. After every pass over $B$, $\mathcal{T}$ sends an output of fixed size to $\mathcal{H}$. Thus, the access pattern is independent of the underlying data and Definition 1 is satisfied.

**Cost Analysis**

During the execution of Algorithm 2, $\mathcal{T}$ gets $|A|$ tuples from $A$, $\gamma|A||B|$ tuples from $B$, and outputs $N|A|$ tuples. Finally, the server writes $N|A|$ tuples to disk.

Therefore, in terms of the number of tuple transfers in and out of $\mathcal{T}$'s memory, the complexity of Algorithm 2 is:

$$|A| + N|A| + \gamma|A||B|.$$

**Algorithm 2** For Secure Coprocessors with Larger Memories

```
γ = max(1, ⌈N/(M − δ)⌉); {#passes over B for every A tuple}
blk = ⌈N/γ⌉; {#output tuples in a pass}
for each tuple a ∈ A do
  a_T = decrypt(get a);
  last = 0; {position of the last matched B tuple}
  for i = 1 to γ do
    matches = 0; {#matches in the current pass}
    current = 0; {position of the current B tuple}
    for each tuple b ∈ B do
      b_T = decrypt(get b);
      if current > last and matches < blk then
        if match(a_T, b_T) then
          joined[matches] = encrypt(join(a_T, b_T));
          matches = matches + 1;
          last = current;
        end if
      end if
      current = current + 1;
    end for
    append (blk − matches) encrypted decoy tuples to joined[];
    put joined[] to H and request H to write joined[] to disk;
  end for
end for
```

**Parameter Selection**

We now discuss how to partition $\mathcal{T}$'s memory between the input and the result tuples to minimize the number of transfers between $\mathcal{T}$ and $\mathcal{H}$. Define $F = M + 1 - \delta$ where $\delta$ represents the small amount of memory needed for data structures other than the input and result tuples. We consider separately the following two cases: (1) $N > F$, and (2) $N \leq F$.

For Case (1), blocking of $A$ is not helpful as we will explain momentarily in Section 4.4.3. So, we keep only one tuple of $A$ in memory and our problem becomes optimally partitioning $F$ between the tuples from $B$ and the joined tuples. Let $F = F_b + F_j$ where $F_b$ denotes the number of $B$ tuples and $F_j$ represents the number of joined tuples. The goal is to find $F_b$ and $F_j$ such that the number of transfers for joining an $A$ tuple with $B$ is minimized.

Observe that for each $a \in A$, it is optimal to scan $B$ a total of $\gamma = \lceil N/(M - \delta) \rceil$ times. For each scan of $B$, $\mathcal{T}$ outputs $blk = \lceil N/\gamma \rceil$ joined tuples where $blk < M - \delta$. We allocate $M - \delta - blk$ tuples for $B$ tuples. So the partition is $F_b = M - \delta - blk$ and $F_j = blk$.

For Case (2), we partition the free memory of $\mathcal{T}$ among the tuples in $A$, $B$, and the joined tuples. Let $F = F_a + F_b + F_j$, where $F_a$ denotes the number of tuples from $A$, $F_b$ the number of tuples from $B$, and $F_j$ the number of joined tuples. The goal is to find $F_a$, $F_b$, and $F_j$ such that the number of transfers for joining $A$ with $B$ is minimized.

Observe that when $\mathcal{T}$ can hold more than $N$ tuples, it is optimal if $\mathcal{T}$ scans $B$ at most once for each $a \in A$. Define $Q$ to be the largest integer such that $Q(1 + N) \leq F$, i.e., $\mathcal{T}$ can hold $Q$ tuples in $A$ and all of their matching tuples in $B$ up to $QN$ matching

tuples. Then the optimal way to partition the memory is $F_a = Q$, $F_b = F - Q(1+N)$, and $F_j = QN$.

**Understanding Blocking of $A$**

We next discuss why blocking of $A$ does not result in any performance gain.

Assume that we partition $A$ into blocks of size $K$. For each tuple in a block, $\mathcal{T}$ allocates a piece of memory to hold a maximum of $N' < N$ joined tuples. $\mathcal{T}$ reads into its memory one block $L$ of $A$ at a time. For each $L$, $\mathcal{T}$ scans the entire table $B$ a total of $P = \lceil N/N' \rceil$ times to find a maximum of $PN'$ matching tuples for each tuple in $A$. Pad the matching tuples for each $a \in A$ to a total of $PN'$ tuples. Conceptually, imagine partitioning these tuples into $P$ groups. During each pass $i$ of $B$, $\mathcal{T}$ retains the $i^{th}$ group of the $PN'$ matching tuples for each element in $L$ and outputs to $\mathcal{H}$ the matching tuples at the end of each pass.

The complexity of this algorithm is $\lceil |A|/K \rceil \lceil N/N' \rceil |B|$ where $\lceil |A|/K \rceil$ represents the number of blocks in $A$ and $\lceil N/N' \rceil$ the number of scans of $B$ per block. Assume that $|A|$ is an integer multiple of $K$, and $N$ is an integer multiple of $N'$ and $M$ respectively. Recall that the complexity for Algorithm 2 is $\gamma |A||B|$. Since $KN' < M$, blocking A is computationally more expensive than the non-blocking case. In terms of transfers between $\mathcal{T}$ and $\mathcal{H}$, Algorithm 2 does $|A| + \gamma |A||B| + N|A|$ tuple transfers while the blocking version does $|A| + \lceil |A|/K \rceil \lceil N/N' \rceil |B| + N|A|$ transfers. The non-blocking version performs less transfers.

### 4.4.4 Parallelism

Consider a server which has more than one secure coprocessor attached. It is readily apparent that both the above algorithms (as well as the upcoming Algorithm 3) are easy to parallelize with a linear speed-up in the number of processors.

## 4.5 Equijoin Algorithms

We now investigate the special, but important, case of equijoins. This study turned out to be quite instructive, as we could not enhance some well known algorithms with security features. We first report those false starts and then present a safe algorithm.

### 4.5.1 False Starts

We explore the adaptation of classical sort-merge join, grace hash join, and the idea of commutative encryption from [5, 10, 21].

**Sort-Merge Join (Unsafe)**

Assume $M = 10$ and for a particular tuple $a \in A$ there are 3 matches in $B$. After the third match, when $\mathcal{T}$ reads the next tuple from $B$, it realizes that there are no more matches in $B$ for $a$. Therefore, $\mathcal{T}$ will read the next tuple from $A$. Such an execution will reveal the number of matches for each tuple.

**Hash-based Join (Unsafe)**

We consider the family of grace hash join algorithms [11, 29]. They begin by partitioning $A$ and $B$ into disjoint subsets called buckets. Each bucket contains only tuples with the same hash of the join attribute value. The corresponding buckets are then joined to produce the result.

The algorithm below depicts our attempt to ensure that the partitioning of a relation into bucket does not leak information. The basic idea is to fill any empty space in all other buckets with decoy tuples as soon as one of them becomes full and output all of them to the server.

Unfortunately, the partitioning phase unavoidably leaks partial information. [1]

---

Obliviously shuffle $A$ (see[24]);

**for** each $a \in A$ **do**

  $a_{\mathcal{T}} = decrypt($`get` $a)$;

  Place $encrypt(a)$ into the $i^{th}$ bucket, where $i = hash(a_{\mathcal{T}}.joinattr)$;

  **if** the $i^{th}$ bucket is full **then**

    Fill all other buckets with decoy tuples and output all the buckets to $\mathcal{H}$;

  **end if**

**end for**

---

[1] For example, an adversary can distinguish between a uniformly distributed relation $A$ and a highly skewed one $B$. Let the size of a bucket be $p$ tuples and let the number of buckets be $n$.

When partitioning $A$, all of the buckets will fill up at relatively the same speed. $\mathcal{T}$ will output the buckets after it has read and hashed about $np$ tuples. On the other hand, when partitioning $B$, one of the buckets will fill up much faster than the rest. $\mathcal{T}$ will now output the buckets after reading a little more than $p$ tuples. By observing the difference in the number of tuples $\mathcal{T}$ reads between writes, an adversary may learn partial information about the distribution of the values of the join attribute.

**Commutative Encryption (Unsafe)**

We now consider an algorithm inspired by the idea of commutative encryption used in [5, 10, 21].

The first encryption is done by the data providers before sending their relations to $\mathcal{H}$. Now, $\mathcal{T}$ executes the algorithm below. The key point is that $\mathcal{T}$ employs symmetric encryption [36] using the same key for re-encrypting the two relations.

---

Obliviously shuffle $A$;

**for** each $a \in A$ **do**

  $a_{\mathcal{T}} = \mathit{decrypt}(\texttt{get}\ a)$;

  `put` symmetric encrypt($a_{\mathcal{T}}$);

**end for**

Similarly re-encrypt $B$ using same key;

Do sort-merge join on the encrypted relations; {Can be done by server}

---

Unfortunately, this adaptation is also unsafe (it leaks the distribution of the duplicates).

### 4.5.2 Privacy preserving Sort-Based Join (Safe)

We now present a safe sort-based equijoin algorithm. This algorithm can be viewed as a specialization of Algorithm 1. Assume $\mathcal{T}$ has obliviously sorted $B$. The key insight is that the $B$ tuples that will join with an $A$ tuple will come from at most $N$ consecutive positions in $B$. This observation is used to avoid the processing of $B$ in rounds and obliviously

**Algorithm 3** Sort-Based Join

```
Obliviously sort B on the join attribute;
for each tuple a ∈ A do
  a_T = decrypt(get a);
  put scratch[] = N encrypted decoy tuples;
  i = 0;
  for each tuple b in B do
    b_T = decrypt(get b);
    t = decrypt(get scratch[i mod N]);
    if b_T.joinattr == a_T.joinattr then
      put scratch[i mod N] =
        encrypt(join(a_T, b_T));
    else
      put scratch[i mod N] = encrypt(t);
    end if
    i = i + 1;
  end for
  Request H to write scratch[] to disk;
end for
```

sorting $scratch[]$ after each round. The size of $scratch[]$ now reduces to $N$ tuples.

For every $A$ tuple, Algorithm 3 initializes $scratch[]$ with $N$ decoy tuples. Now, for every tuple that $\mathcal{T}$ reads from $B$, $\mathcal{T}$ also reads a specific location from $scratch[]$ in a circular fashion; for the $i^{th}$ tuple, $\mathcal{T}$ reads $scratch[i \bmod N]$. $\mathcal{T}$ writes back to the same location either the value just read (though encrypted differently so it is indistinguishable to the adversary) or the joined tuple if the tuple from $B$ matches the tuple of $A$. A logical concern is how to avoid overwriting real result tuples from a previous match. The overwriting will never happen because all the real result tuples will be in at most $N$ consecutive positions in $scratch[]$.

To ensure authenticated computation, both $A$ and $B$ relations need to be encrypted under OCB mode. Since $A$ is accessed sequentially, it can be encrypted using the procedure described in Section 3.3.3. However, $B$ requires oblivious sorting. Hence, its encryption should use the strategy described for encrypting $scratch[]$ array in Section 4.4.1.

We now describe how to encrypt and decrypt tuples in $scratch[]$ in the OCB mode. We refer to $\mathcal{T}$ reading tuples from 0 to $N-1$ in $scratch[]$ as a round. In each round, $\mathcal{T}$ treats the $N$ tuples written to and read from and $scratch[]$ as one message respectively. In each round, if $\mathcal{T}$ accepts the $N$ tuples it decrypted, it continues to the next round; otherwise it terminates the computation. For the $N$ output tuples in each round, $\mathcal{T}$ encrypts them in the OCB mode with a fresh nonce and the same encryption key.

**Correctness (Proof Sketch)**

Since $B$ is sorted obliviously, this step is safe. After getting a tuple from $B$, $\mathcal{T}$ always reads a specific location from $scratch[]$ and always writes something of the same

size back to the same location. These actions are executed regardless of the content of the underlying relations. Therefore, Definition 1 is satisfied.

**Cost Analysis**

$\mathcal{T}$ first obliviously sorts $B$ leading to a total of $|B|(\log_2 |B|)^2$ tuple transfers. During the rest of the execution, $\mathcal{T}$ gets $|A|$ tuples from $A$ and $|A||B|$ tuples from $B$. For every tuple of $A$, $\mathcal{T}$ outputs $N$ decoy tuples, for a total of $N|A|$ decoy tuples. For every tuple of $A$ and $B$, $\mathcal{T}$ gets a decoy tuple from $\mathcal{H}$ and outputs a result tuple, for a total of $|A||B|$ gets of decoy tuples and $|A||B|$ puts of result tuples. Finally, the server writes $N|A|$ tuples to disk.

Thus, in terms of transfers in and out of $\mathcal{T}$'s memory, the complexity of Algorithm 3 is:

$$|A| + |A|N + |B|(\log_2 |B|)^2 + 3|A||B|.$$

If the data providers can send sorted data to the service, the step of oblivious sorting can be avoided and the complexity becomes:

$$|A| + |A|N + 3|A||B|.$$

## 4.6 Performance Analysis

In this section, we study the performance characteristics of the proposed algorithms. We identify two important parameters:

- $\alpha = N/|B|$

- $\gamma = \lceil N/M \rceil$ (ignoring $1 - \delta$)

Note $\alpha \in [1/|B|, 1]$ assuming there is at least one matching tuple for every $a \in A$, and $\gamma \in [1, |B|]$.

Two other parameters that merit consideration are: (i) the size of the tuples, and (ii) the size of the relations. The first plays an insignificant role in Algorithms 1 and 3.

For Algorithm 2, its effect can be understood by understanding $\gamma$. The running time of the algorithms increases quadratically in terms of the size of the relations, but that is what we expected. It is more interesting to study the performance with respect to $\alpha$ and $\gamma$.

Taking $|A| = |B|$, we rewrite the cost formulas for the three algorithms as follows:

**Algorithm 1** $|B| + 2|B|^2 + 2\alpha|B|^2 + 2|B|^2(\log 2\alpha|B|)^2$

**Algorithm 2** $|B| + \alpha|B|^2 + \gamma|B|^2$

**Algorithm 3** $|B| + 3|B|^2 + \alpha|B|^2 + |B|(\log |B|)^2$

### 4.6.1 $\gamma = 1$

We find that Algorithm 2 dominates the other two algorithms. To see this, set $\alpha$ to 1 (the largest value it can take) for Algorithm 2 and set it to $1/|B|$ (the smallest value) for Algorithms 1 and 3 and examine the cost formulas.

Note $\gamma$ is 1 when the maximum number of $B$ tuples that join with any of the $A$ tuples can fit in the free memory of $\mathcal{T}$. It is interesting that in this case, Algorithm 2 designed for general joins beats a specialized algorithm that works only for equijoins. The relative performance gap increases as the size of the relations increases.

### 4.6.2 General Joins, $\gamma > 1$

Algorithm 1 outperforms Algorithm 2 when $\gamma > 2 + \alpha + 2(\log 2\alpha|B|)^2$. Let's substitute $\frac{1}{|B|}$ for $\alpha$ (the smallest value it can take). Algorithm 1 outperforms Algorithm 2 when $\gamma > 4$, i.e., $N$ is more than 4 times the free memory of the secure coprocessor. For a fixed table size $|B|$, as $\alpha$ increases, $\gamma$ also increases.

### 4.6.3 Equijoins, $\gamma > 1$

Both Algorithms 1 and 3 are insensitive to $\gamma$. To compare them, let us substitute $\alpha$ in the last term in the cost formula for Algorithm 1 with $1/|B|$, the smallest value $\alpha$ takes. We rewrite the cost formula for Algorithm 1 as $|B| + 2|B|^2 + 2\alpha|B|^2 + 2|B|^2$. Then the comparison of Algorithm 3 to Algorithm 1 reduces to comparing $|B|(\log|B|)^2$ and $\alpha|B|^2 + |B|^2$. In this case, Algorithm 3 outperforms Algorithm 1 for any value of $\alpha$ and $|B|$.

Finally, let us compare Algorithm 3 to Algorithm 2. Their cost comparison boils down to comparing $3|B|^2 + |B|(\log|B|)^2$ with $\gamma|B|^2$. When $\gamma \leq 3$, Algorithm 2 outperforms Algorithm 3 regardless of the value of $|B|$. When $3 < \gamma < 4$, Algorithm 3 outperforms Algorithm 2 for sufficiently large $|B|$. When $\gamma \geq 4$, Algorithm 3 outperforms Algorithm 2 whenever $|B| \geq 1$.

### 4.6.4 Performance Relationship Summary

Figure 4.1 summarizes the performance relationship among the three algorithms.

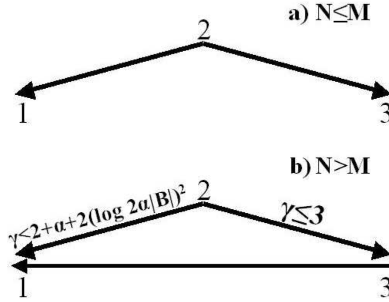


Figure 4.1: Performance Relationship

### 4.6.5 Comparison with Secure Function Evaluation

We now compare the performance of the proposed algorithms to the technique for secure function evaluation (SFE), based on secure circuit evaluation [17, 45]. Since Algorithm 2 performs better than Algorithm 1, we will conservatively compare Algorithm 1 to the most recent (and to the best of our knowledge, most efficient) technique, provided in [32, 34].

We will compare the number of communications in this analysis. We are again being conservative; we are comparing communication between the secure coprocessor and the server it is attached to in the case of Algorithm 1 to the communication across wide-area network in the case of SFE.

Assume $|A| = |B|$ and that each tuple is $w$ bits wide, that the output has $|B|Nw$ bits, and that the circuit for matching two $w$-bit tuples requires $G_e(w)$ gates. Then a secure circuit for general join will have at least $|B|^2 G_e(w)$ gates. Note that $G_e(w) \geq 2w$ in the simple case that two tuples are matched if their $L1$ Norm is smaller than some threshold.

Assume $k_0$ is the number of bits in the supplemental keys used while building

the circuit, the cheating probability of $P_A$ is exponentially small in $l$, and the cheating probability of $P_B$ is exponentially small in $n$. In practice, $k_o \geq 64$ and $l = n \geq 50$.

$P_A$ and $P_B$ need to make at least $|B|w$ 1-out-of-2 oblivious transfers where each oblivious transfer uses one public key encryption, $4l|B|^2G_e(w)$ pseudo-random function evaluations, $2l|B|wN$ public key encryptions for partial proofs of knowledge and gradual opening of commitments, and $nl|B|wN$ public key encryptions for blind signatures.

$P_A$ needs to send $2l$ copies of $4k_oB2G_e(w)$ bit encrypted circuit to $P_B$ and send at least $32lk_1$ bits for each oblivious transfer. Here, $k_1$ is the security parameter for oblivious transfer; $k_1 \geq 100$ in practice. $P_B$ sends $2nl|B|wNk_1$ bit commitments to $P_A$.

Total communication cost can thus be estimated as

$$8lk_o|B|^2G_e(w) + 32lk_1(|B|w) + 2nlNk_1(|B|w).$$

To compare the communication cost of SFE and our solutions in bits, we multiply the cost formula for Algorithm 1 with $w$. Let $k_0 = 64, k_1 = 100, l = n = 50$, their minimum values suggested in [32], and take $G_e(w) = 2w$. For low values of $\alpha$, it can be seen that SFE can be orders of magnitude slower.

# Chapter 5

# Privacy Preserving Joins on Secure Coprocessors

In this chapter, we further study privacy preserving join algorithms on secure coprocessors based on our initial attempt in the previous chapter.

In Section 5.1, we point out two problems in our previous definition of a privacy preserving join in Chapter 4. We then remove the assumption of the number $N$ in this previous definition and arrive at a more general formulation of the problem.

In Section: 5.2, we describe the assumptions, notations, and cryptographic tools we use in this chapter. In Section 5.2.2, we optimize the performance of the oblivious sort algorithm used in Chapter 4. In Section 5.2.3, we describe our usage of a pseudo random number generator to avoid materializing a table of the cartesian product of multiple tables.

In Section 5.3, we propose three provably correct and secure algorithms to compute general joins of arbitrary predicates. Our solutions overcome the challenge of the limited

memory capacity of a secure coprocessor, 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 Section 5.4, we evaluate the performance of our algorithms by numerical examples and show the performance superiority of our approach over that of secure multi-party computation.

## 5.1 New Definition of Privacy Preserving Joins

In this section, we describe the *safety definition* formalized in Chapter 4 and point out three problems, two privacy related and one performance, implied by the definition. We then define what it means for an algorithm running on a secure coprocessor to be *privacy preserving* with respect to an honest-but-curious adversary. For the sake of simplicity, we henceforward only say an algorithm is privacy preserving when the context is clear.

### 5.1.1 Problems in Previous Definition

We stated in Chapter 1 that for an algorithm to be privacy preserving, it must not reveal any information from its accesses to $\mathcal{H}$. The assumption of honest-but-curious adversaries model is implicit in the statement. We quote our definition of a privacy preserving join on a secure coprocessor in Chapter 1 as follows:

**Definition 2.** *[Privacy Preserving Join Algorithms] 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 privacy preserving if* $J_{AC}$ *and* $J_{CD}$ *are identically distributed.*

The fundamental goal of a privacy preserving join algorithm is to reveal no information other than what can be inferred from the join result. Although the above intuition for a privacy preserving join algorithm is correct, we point out two problems in Definition 2 that cause the algorithms satisfying the definition to leak more information than what we expect from the fundamental goal.

Firstly, the assumption of the number $N$ permits join algorithms to leak the knowledge of $N$ by definition. All of the proposed algorithms in 4 produce a fixed output of $N|A|$ equal sized tuples which is a super set of the real join results. An adversary who sits between $\mathcal{H}$ and a recipient of the join result may estimate $N$ once it observes the size of the output, given it knows $|A|$ and the size of a join tuple. Another way for an adversary to learn $N$ is by eavesdropping the communication between $\mathcal{H}$ and $\mathcal{T}$ since $\mathcal{T}$ outputs result tuples to the external memory and disks in batches of $N$. We find it unnecessary and not well justified in practice for a join algorithm to reveal $N$. In particular, there might be cases where $N$ is sensitive information and shall not be made public.

Secondly, lacking of an explicit requirement of a join result allows a recipient to infer more information than had it received exact join results . Definition 2 does not explicitly prescribe the join result which allows the algorithms in Chapter 4 to produce a superset of the real join result and leak information. The algorithms in Chapter 4 are nested loop join based. For a tuple $a \in A$, if $a$ matches $N' < N$ tuples in $B$, these algorithms

generate an extra $N - N'$ *decoy* tuples to pad the output to a group of $N$ tuples. Since decoy tuples are simply fixed string patterns of the same length as real join tuples (see Chapter 4), a recipient is able to determine the number of real joins per group and derive 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 decoy tuples in the final join result incur unnecessary overhead. The algorithms in Chapter 4 output 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 particular, for a highly skewed join result set, this output size may dramatically exceed the size of the original query result. Consider a worst case scenario: let one of the tuples in $A$ match all the tuples in $B$ and none of the rest of the tuples in $A$ matches any tuple in $B$. The proposed algorithms in [4] produce an output of size $|A||B|$ whereas the actual join size is only $|B|$. The situation is aggravated when joining multiple tables $A_1, \ldots, A_i$. In the worst case, the output size can be $|A_1| \cdots |A_i|$ whereas the real join size is merely a small portion of it.

### 5.1.2 Proposed Definition

We define privacy preserving joins with respect to honest-but-curious adversaries. We distinguish our definition from Definition 2 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.

In Definition 3, we are given two sets of input tables $A_i$'s and $B_i$'s where $A_i$ and $B_i$ having the same size and schema respectively, and joining all $A_i$'s produces the same

output size as joining all $B_i$'s over the same query. Definition 3 asserts that an algorithm computing the query in question is privacy preserving if the distribution of the memory access patterns when running the algorithm on inputs $A_i$'s is identical with that when running it on $B_i$'s. Alternatively, we say a join algorithm is privacy preserving if its access pattern is independent of the input tables.

**Definition 3.** *[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.*

## 5.2 Preliminaries

This section describes the assumptions, notations and cryptographic tools used in this chapter.

### 5.2.1 Assumptions and Notations

Assume $J$ participating databases $X_1, \ldots, X_J$. Let $\mathcal{D} = X_1 \times \cdots \times X_J$, $L = |\mathcal{D}|$, $I = \{1, \ldots, L\}$, and `iTuple` be an element in $\mathcal{D}$. Let $R$ denote the set of the join results with size $S$. For the 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.

In real implementation, a logical index can be easily converted into the individual index of each of the $J$ tuples and $\mathcal{D}$ need not be materialized.

Let a *decoy* be a string of a fixed pattern with the same length as a real join result. When encrypted, two decoys appear completely indistinguishable. 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. We assume a constant memory space allocated for `iTuple`s, program code, and other necessary data structure and variables. Let $M$ in unit of tuples be the free memory of $\mathcal{T}$ and we assume that $M$ is dedicated to the storage of `oTuple`s. In our discussion of communication cost, we state the cost in terms of tuples.

### 5.2.2 Oblivious Sort

We use oblivious sorting as a cryptographic building block in our proposed algorithms. In this section, we optimize the performance of the oblivious sort used in Section 4.4.1 of Chapter 4.

Our algorithms require removing the generated decoys in a privacy preserving way. Assume a list of encrypted output tuples on $\mathcal{H}$'s memory, some of them are real join results and some are decoys. One way to remove the decoy tuples is as follows: $\mathcal{T}$ reads a tuple, decrypts it, and outputs nothing if it is a decoy. Otherwise it re-encrypts the tuple and outputs the encrypted tuple. An adversary that observes earlier program execution knows which `iTuple` produces the tuple that $\mathcal{T}$ just read. It will be able to determine that this

particular `iTuple` does not produce a match if it sees no output at this time. What if $\mathcal{T}$ reads a block of $K$ tuples at a time, then outputs the real joins? Unfortunately, the adversary can still estimate the distribution of the matches.

We propose using oblivious sort to remove the decoys in a privacy preserving and efficient way. Suppose we want to keep $\mu$ target elements in a list of length $\omega$ and remove the rest. The target elements are the real join results. A straightforward way is to obliviously sort the entire list, separate the unwanted elements from the rest, and remove them. This results in a cost of $\frac{1}{4}\omega(\log_2 \omega)^2$ comparisons and $\omega(\log_2 \omega)^2$ element transfers between the secure coprocessor and the host. Alternatively, we propose applying oblivious sort on smaller portions of the list repeatedly to improve efficiency.

First, we create a buffer of $\mu+\Delta$ elements, where $\Delta$ 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 target elements in the top $\mu$ positions in the buffer. Since at most $\mu$ elements are kept, the bottom $\Delta$ elements in the swap area can be overwritten. We copy another $\Delta$ elements from the source list, and overwrite the bottom $\Delta$ elements. We obliviously sort the buffer again to keep all the wanted elements in the top positions.

This process is continued until all elements in the source list are processed. The top $\mu$ elements in the buffer are now the desired elements to keep. During the process, we need to obliviously sort the buffer $\frac{\omega-\mu-\Delta}{\Delta}+1$ times, and each time we need to perform $\frac{\mu+\Delta}{4}[\log_2(\mu+\Delta)]^2$ comparisons. Therefore, the total number of comparisons, 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 merely $4\mathcal{C}_{(\omega,\mu)}(\Delta)$

Given $\omega$ and $\mu$, it is possible to minimize the total number of element transfers between the secure coprocessor and the host, by carefully selecting $\Delta$. The optimal $\Delta$, denoted as $\Delta^*$, can be found by solving the following optimization problem:

$$\Delta^* = \arg\min_{\Delta > 0} 4\mathcal{C}_{(\omega,\mu)}(\Delta). \tag{5.1}$$

Since $\Delta^*$ also minimizes $\log[\mathcal{C}_{(\omega,\mu)}(\Delta)]$, $\Delta^*$ can be found by solving

$$\frac{\partial}{\partial \Delta} \log[\mathcal{C}_{(\omega,\mu)}(\Delta)] = \frac{\mu}{\Delta} - \frac{2}{\log_2(\mu + \Delta)} = 0.$$

As such, $\Delta^*$ is the first quadrant intersection point of the two curves $\frac{\Delta}{\mu}$ and $\frac{\log_2(\mu+\Delta)}{2}$, and does not depend on $\omega$.

### 5.2.3 Generating Random Order

Algorithm **6** in Section 5.3 needs to randomly access every tuple in $\mathcal{D}$ exactly once. $\mathcal{T}$ could generate a random permutation over $I$, then access $\mathcal{D}$ based on this random order. Materializing a random permutation over a large index set is slow and requires a lot of storage space. We propose using a Pseudo Random Number Generator (PRNG) to generate a random permutation on the fly.

A special PRNG, Maximal Linear Feedback Shift Register (MLFSR) with $n$ internal states generates all possible integers in $\{1, \ldots, 2^n - 1\}$ before it produces repeated values. For an index set $I$, we choose an MLFSR with $l$ internal states where $l$ is the smallest integer such that $2^l - 1 \geq |I|$. Calling the MLFSR $2^l - 1$ times will eventually generate every number in $\{1, \ldots, 2^l - 1\}$ exactly once. A generated number that is outside $I$ is simply discarded.

## 5.3 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 level. In the algorithm description, function satisfy($\cdot$) takes an `iTuple` as input and outputs true if the `iTuple` satisfies the join predicate and false otherwise. The function join($\cdot$) takes an `iTuple` as input and returns a join result.

For simplicity, we treat only the plaintext databases and skip the discussion of the cryptographic operations including encryption, decryption, and MAC authentication throughout our description of the algorithms. These related topics have been covered in Section 5.2.

### 5.3.1 Algorithm 4 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 $\mathcal{H}$. We turn the intuition into Algorithm **4**.

For participating databases $X_1, \ldots, X_J$, $\mathcal{T}$ sequentially reads `iTuple`s in a predefined and fixed order, writes a join result to $\mathcal{H}$ if the current `iTuple` leads to one, and writes a decoy otherwise. Hence, Algorithm **4** always outputs $L$ `oTuple`s, regardless if there are only $S$ real results.

After reading all tuples, $\mathcal{T}$ has output all $S$ real results and $L-S$ decoys. Next, $\mathcal{T}$

**Algorithm 4** : For Secure Coprocessors with Small Memory

```
for each iTuple do
  if satisfy(iTuple) then
    write join(iTuple) to H
  else
    write a decoy to H
  end if
end for
optimally oblivious sort outputs giving priority to decoys
remove decoys and output S results
```

filters out the decoys in `oTuples` by obliviously sorting the `oTuples`, giving priority to the decoys. $\mathcal{T}$ then reads in all `oTuples`, removes the decoy tuples, and outputs the real joins.

The advantage of this algorithm is that it only requires a memory size of two, of which one for an `iTuple` and the other for an `oTuple` and also a memory size of two during the oblivious shuffling phase. Meanwhile, this implies that Algorithm **4** does not take advantage of a large memory, resulting in significant communication cost when $S$ is much smaller than $L$, which is typically the case.

**Proof of Security**

*Proof.* From the description of Algorithm **4**, the communication patterns between $\mathcal{T}$ and $\mathcal{H}$ are determined by the input size $L$ and the output size $S$ alone, and thus are independent of the contents of $X_1, \ldots, X_J$. Algorithm **4** satisfies Definition 3 and is privacy preserving. □

**Communication Cost**

The communication cost is $L$ for read and write respectively, and $\frac{L-S}{\Delta^*}(S+\Delta^*)[\log_2(S+\Delta^*)]^2$ for oblivious sort, where $\Delta^*$ is the optimal swap size given by Eqn. 5.1 with $\omega = L$ and $\mu = S$. We summarize the total communication cost in the following:

$$2L + \frac{L-S}{\Delta^*}(S+\Delta^*)[\log_2(S+\Delta^*)]^2. \tag{5.2}$$

### 5.3.2 Algorithm 5 for Secure Coprocessors with Large Memory

We observe that a significant portion of the communication cost of Algorithm **4** is from obliviously filtering $L - S$ decoys. One way to remove this portion of cost is to only write out the real results by doing the following: for participating databases $X_1, \ldots, X_J$, $\mathcal{T}$ sequentially reads `iTuple`s in a pre-defined order. If the current `iTuple` leads to a join result, $\mathcal{T}$ stores it in its memory.

If $\mathcal{T}$ flushes all $M$ results in its memory to $\mathcal{H}$ whenever the memory is full, assuming $\mathcal{T}$ processes $N$ `iTuple`s between two flushes, then an adversary who observes the communication patterns knows that there are $M$ results in these $N$ `iTuple`s. This information leakage undermines the privacy preserving property of the algorithm.

To address this problem, $\mathcal{T}$ can write out the stored $M$ results only after scanning all $L$ `iTuple`s. $\mathcal{T}$ 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 writing cycle is $L$ tuples and the writing efficiency is $\frac{M}{L}$.

To avoid recording the same join result twice, $\mathcal{T}$ records the index of the `iTuple` that leads to the previous join result, and only starts to store a result if the current index

exceeds the recorded one. The resulting Algorithm **5** is shown below.

### Proof of Security

*Proof.* The secure coprocessor reads $L$ `iTuple`s sequentially a total of $\lceil S/M \rceil$ times. After each but the last scan of the $L$ `iTuple`s, the secure coprocessor outputs $M$ result tuples. It outputs $L - (\lceil \frac{S}{M} \rceil - 1)M$ tuples after the last scan. The communication patterns are determined by the input size $L$, the output size $S$, and the memory size $M$ of the secure coprocessor, and thus is independent of the contents of $X_1, \ldots, X_J$. Algorithm **5** satisfies Definition 3 and is privacy preserving. □

### Communication Cost

The write cost $S$ of Algorithm **5** is clearly minimal. The read cost is $\lceil \frac{S}{M} \rceil L$, because $\mathcal{T}$ spends $\lceil \frac{S}{M} \rceil$ write cycles to output all $S$ results to $\mathcal{H}$, and $\mathcal{T}$ reads all $L$ `iTuple`s in each cycle. Hence, the total communication cost is then given by

$$S + \lceil \frac{S}{M} \rceil L. \tag{5.3}$$

As the memory size $M$ increases, the communication cost decreases roughly proportionally to $1/M$ and the cost reduction is more significant in the region where $M$ is small, as illustrated in Figure 5.1. The communication cost approaches the minimum $S + L$, as $M$ approaches $S$.

## 5.3.3 Algorithm 6 for Trading Privacy Preserving Level with Efficiency

In Algorithm **5**, during each write cycle, $\mathcal{T}$ has to read all `iTuple`s before it outputs the $M$ results stored in its memory. When $M \ll S$, $\mathcal{T}$ spends a large number of write cycles

**Algorithm 5** : For Secure Coprocessors with Large Memory

```
// lindex: largest index of iTuple that leads to a join
// pindex: index of iTuple of previous join
lindex := 0; pindex :=-1
while pindex < lindex do
  prepare to read one round of iTuple in a fixed order
  for each iTuple do
    if satisfy(iTuple) && current index > pindex then
      store join(iTuple) in memory
      if current index > lindex then
        lindex := current index
      end if
    end if
    if memory is full then
      write M results to H
      pindex := current index
    end if
  end for
end while
output S results
```

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 $S$, and is able to randomly read every `iTuple` once and only once. This can be done by reading the tuples according to an order generated by a PRNG, as we discussed in Section 5.2.3.

To achieve better write efficiency, $\mathcal{T}$ can first partition $L$ randomly ordered `iTuple`s into $\frac{L}{n}$ segments, each containing randomly selected $n$ tuples. As $\mathcal{T}$ processes a segment, it stores the join results in its memory. $\mathcal{T}$ writes all stored results to $\mathcal{H}$ after finishing processing one segment. It repeats the process until completing all segments. $\mathcal{T}$ then obliviously filters out the decoys and outputs the real results.

If the number of join results generated for a segment is no more than the memory size, i.e. $K <= M$, then $\mathcal{T}$ simply writes out $M$ `oTuple`s with $M - K$ decoys. In this case, $\mathcal{T}$ can achieve a write efficiency of $M/n$, which is better than the one of Algorithm **5**.

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 segment again to output the missing results. Alternatively, $\mathcal{T}$ can use Algorithm **5** to re-output all the join results. Nevertheless, these "salvage actions may lead to information leakage and compromise the privacy preserving property of the join process. We refer to this case 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 `iTuple`s. Denote the event of having $k$ results in these $n$ `iTuple`s. The probability of $x(n) = k$ is the same as the probability of having $k$ balls of 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

$$P[x(n) = k] = \frac{1}{\binom{L}{n}} \binom{L-S}{n-k} \binom{S}{k}, \tag{5.4}$$

As such, $P[x(n) \leq M]$ is given by

$$P[x(n) \leq M] = \frac{1}{\binom{L}{n}} \sum_{k=1}^{M} \binom{L-S}{n-k} \binom{S}{k} \tag{5.5}$$

The probability for a blemish case to happen, i.e., at least one of the segments 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 $P_M(n)$ be acceptably small.

Intuitively, the larger the segment size $n$, the higher the chance a blemish case happens, and the less privacy preserving a join process is. Meanwhile, a larger $n$ also implies fewer decoys generated to pad the output for each segment to $M$ `oTuple`s, which in turn reduces the cost of obliviously filtering the decoys for final output. We see a clear trade-off between efficiency and level of privacy preserving in the process described above.

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

$$n^* = \arg\min_{n>0} n \ \text{ subject to } P_M(n) < \varepsilon. \tag{5.6}$$

The significance of $n^*$ is that, if $\mathcal{T}$ processes `iTuple`s by this optimal segment size $n^*$, then a blemish case will happen only with probability $\varepsilon$.

Following above analysis, we propose Algorithm **6** with a privacy preserving guarantee of probability $1 - \varepsilon$ where $\varepsilon \in [0, 1]$.

**Algorithm 6** : For Trading Privacy Preserving Level with Efficiency

```
screen all iTuples to get L and S
compute n* from Eqn. 5.6
use PRNG to generate a random order for reading iTuples
p1 :=0; p2 :=0
for each iTuple do
  increment p2
  if satisfy(iTuple) then
    record join(iTuple) in memory
  end if
  if p2 − p1 == n* || p2 == L then
    output max(S, M) real results and decoys to H
    p1 := p2
  end if
end for
optimally oblivious sort output giving priority to decoys
remove decoys and output S results
```

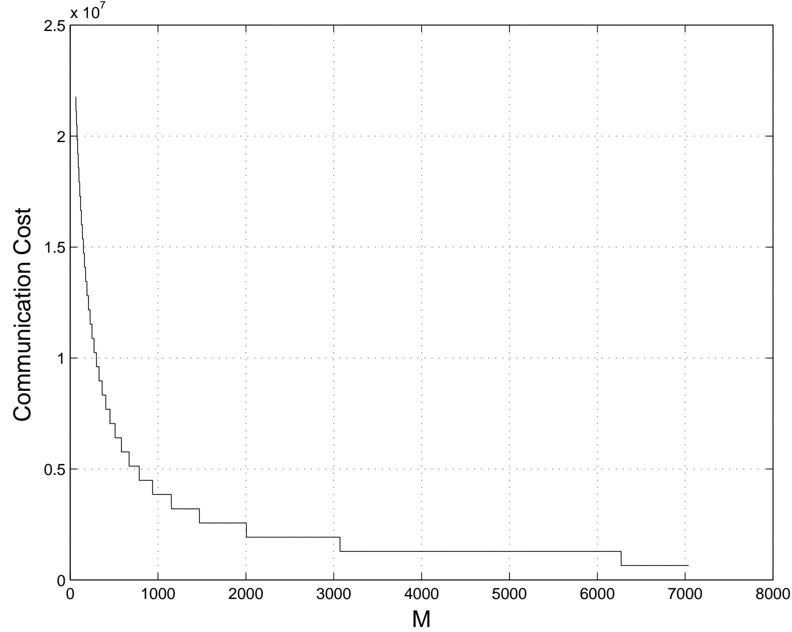


Figure 5.1: Communication cost of Algorithm **5** as a function of memory size $M$, under setting $L = 640,000$ and $S = 6,400$.

### Proof of Correctness and Security

*Proof.* In the algorithm description, the communication patterns between $\mathcal{T}$ and $\mathcal{H}$ during the entire process are only functions of $L$, $S$ and $M$, and thus are independent of the contents of $X_1, \ldots, X_J$.

Algorithm **6** outputs all $S$ real results and is correct if a blemish case does not occur. Hence, its correctness is guaranteed with probability $1 - \varepsilon$.

When a blemish case occurs, $\mathcal{T}$ will not be able to record and output all the results for the current segment. Consequently, the number of total real results written to $\mathcal{H}$ is less than $S$ and the join process is not correct. In this case, $\mathcal{T}$ needs to perform "salvage" actions, which might leak additional information about the contents of participating databases. However, the probability for such events to happen is bounded by $\varepsilon$.

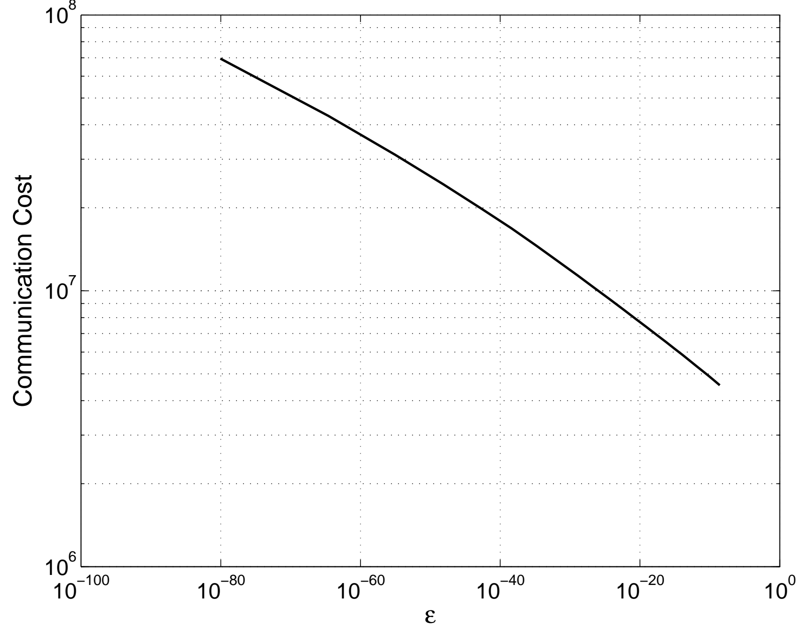


Figure 5.2: Communication cost of Algorithm **6** as a function of $\varepsilon$, under setting $L = 640,000$, $S = 6,400$, and $M = 64$.

Based on the above argument, Algorithm **6** is privacy preserving with probability at least $1 - \varepsilon$. □

**Communication Cost and Trade-off between Privacy Preserving Level and Efficiency**

The read cost of Algorithm **6** is merely $2L$: $L$ for screening and $L$ for actual processing. The write cost of A3 consists of two portions: one of outputting the results and the other of oblivious sort. The first portion costs $\lceil \frac{L}{n^*} \rceil M$; the oblivious sort part costs

$$\frac{\lceil \frac{L}{n^*} \rceil M - S}{\Delta^*}(S + \Delta^*)\lceil \log_2(S + \Delta^*) \rceil,$$

where $\Delta^*$ is the size of the swap area that minimizes the cost for obliviously filtering $\lceil \frac{L}{n^*} \rceil M - S$ decoys. We compute $\Delta^*$ by solving the problem in Eqn. 5.1, with $\omega = \lceil \frac{L}{n^*} \rceil M$

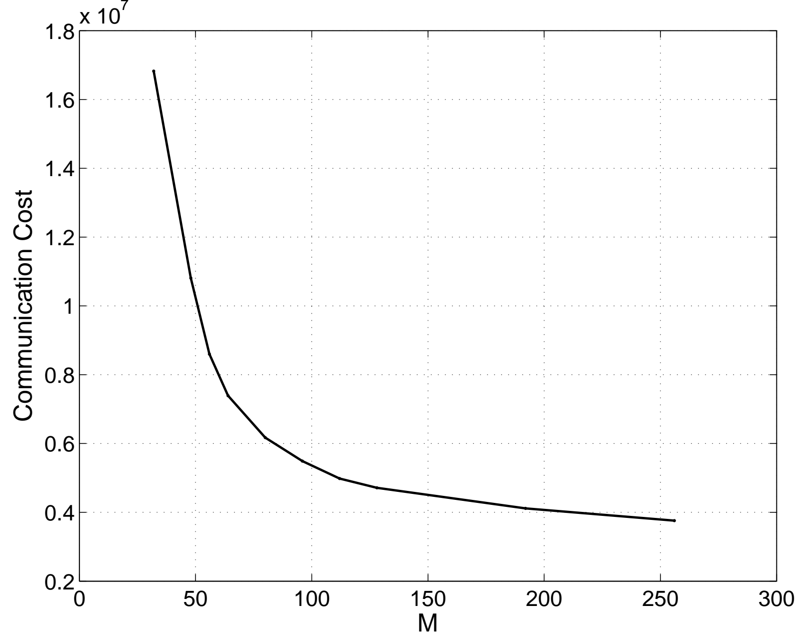


Figure 5.3: Communication cost of Algorithm **6** as a function of $M$, under setting $L = 640,000, S = 6,400$ and $\varepsilon = 10^{-20}$.

and $\mu = S$. The total communication cost of Algorithm **6** is then given by

$$2L + \lceil \frac{L}{n^*} \rceil M + \frac{\lceil \frac{L}{n^*} \rceil M - S}{\Delta^*}(S + \Delta^*)[\log_2(S + \Delta^*)] \tag{5.7}$$

for the case where $\varepsilon \neq 0$ and $M < S$.

As $\varepsilon$ increases, the privacy preserving level drops. Algorithm **6** can increase the segment size $n^*$ while still keeping the chance of encountering blemish cases being less than $\varepsilon$. Consequently, the communication cost of Algorithm **6** deceases monotonically as $\varepsilon$ increases, as shown in Figure 5.2.

Based on the same argument, the communication cost of Algorithm **6** increases as $\varepsilon$ decreases. In the extreme case where $\varepsilon = 0$ and $M < S$, $n^*$ can only be $M$. In this case, $\mathcal{T}$ writes out one real result or decoy upon processing every `iTuple`, and Algorithm **6** reduces to Algorithm **4**.

From Figure 5.2 we observe that decreases in the communication cost of Algorithm **6** is less significant as $\varepsilon$ approaches 1. For example, the cost reduction is more than $1.3 \times 10^7$ as $\varepsilon$ increases from $10^{-60}$ to $10^{-50}$, while the reduction is only less than $10^7$ as $\varepsilon$ increases from $10^{-20}$ to $10^{-10}$. This implies that it is more profitable to trade privacy preserving level with efficiency when $\varepsilon$ is small than when it is large.

As the memory size $M$ increases, Algorithm **6** can increase the segment size $n^*$ while still maintaining the same privacy preserving level $1-\varepsilon$. In the case where $M \geq S$ and $n^* = L$, Algorithm **6** ends up outputting all $S$ results at the end of the screening process[1] and the communication cost is the minimum $L + S$. This relation between communication cost and $M$ is illustrated in Figure 5.3.

Similar to the relationship between cost reduction and $\varepsilon$, the cost reduction is more significant for a small $M$ (with respect to $S$) than that for a large $M$. Hence, to improve efficiency, upgrading memories yields more gain when $M$ is small than when it is large, with respect to a target $S$.

### 5.3.4 Comparison of Algorithms 4, 5 and 6

We compare levels of privacy preserving and communication costs of Algorithms **4**, **5**, and **6** in Table 5.1.

As seen from Table 5.1, Algorithm **4** and **5** guarantee 100% privacy preserving, while Algorithm **6** guarantees $(1 - \varepsilon)\times$ 100% privacy preserving, which is by design. However, as $\varepsilon$ can be chosen to be arbitrarily small to meet practical needs, we believe Algorithm

[1] It is easy to see $\mathcal{T}$ can record join results in its memory during the screening process. If the memory is not full after screening all `iTuples`, then $\mathcal{T}$ knows it has recorded all $S$ results and is ready to output them.

| | Privacy Preserv- -ing Level | Communication Cost |
|---|---|---|
| **4** | 100% | $2L + \frac{L-S}{\Delta^*}(S+\Delta^*)[\log_2(S+\Delta^*)]^2$ |
| **5** | 100% | $S + \lceil \frac{S}{M} \rceil L$ |
| **6** | $(1-\varepsilon)\times$ 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 5.1: Level of privacy preserving vs. communication cost.

**6** is practically as secure as Algorithm **4** and **5**. For example, if $\varepsilon = 10^{-10}$, then on average, Algorithm **6** performs a "salvage" action once every $10^{10}$ trials.

In general, Algorithm **4** has the highest communication cost among the three algorithms. Directly comparing the communication costs of Algorithm **5** and **6** is difficult. The communication cost of Algorithm **5** mainly depends on the write efficiency $\frac{M}{L}$, while that of Algorithm **6** 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 **5** has low write efficiency; hence, Algorithm **6** outperforms Algorithm **5** in terms of communication cost. This is illustrated in the next section.

In cases where $M$ is close to $S$, Algorithm **5**'s write efficiency $M/L$ is high with respect to the optimal value $S/\ L$. Consequently, Algorithm **5** might have less communication cost than that of Algorithm **6**. Algorithm **5** might be attractive in some scenarios

considering its preservation of privacy and ease of implementation for it does not require oblivious sort or random access to `iTuple`s.

### 5.3.5 Parallelism

When more than one $\mathcal{T}$ is available at the same $\mathcal{H}$, it is possible for the proposed algorithms to run parallelly to speed up the join process. Assume a total of $P$ secure coprocessors.

For Algorithm **4**, parallelly processing `iTuple`s and generating `oTuple`s can be achieved by simply partitioning the `iTuple`s into $P$ sets and allocating one set to one coprocessor to process. Oblivious filtering out decoys in parallel requires a parallel bitonic sort. Assume we have $N$ items to sort on $P$ secure coprocessors where $N > P$. Each secure coprocessor has about $N/P$ items and first sorts them locally using sequential bitonic sort. Then the $P$ secure coprocessors sort the $P$ sorted lists using bitonic sort and treats each list as one single element.

For Algorithm **5**, one $\mathcal{T}$ serves as the coordinator of parallelism. It screens the `iTuple`s and calculates the output size $S$. Without loss of generality, we assume that $S$ divides $P$, and so denote $blk = \frac{S}{P}$. The coordinator then asks the $i^{th}$ $\mathcal{T}$ to output a total of $blk$ join results starting from $[(i-1)blk]^{th}$ join results. It is necessary that all $\mathcal{T}$s read the `iTuple`s in the same predefined and fixed order. Algorithm **5** enjoys a linear speed up in performance.

For Algorithms **6**, the input partitioning is done through the use of the maximal LFSR. All $\mathcal{T}$ seed their maximal LFSR with the same value respectively such that all the LFSRs will generate the same sequence of random numbers. Each $\mathcal{T}$ is then responsible

for a particular range of the sequence of random numbers generated; it only processes the `iTuple`s with an index that falls in its respective range. Parallelism in processing `iTuple`s and generating `oTuple`s can be achieved. Again, Algorithm **6** uses a parallel bitonic sort to exploit the power of multiple secure coprocessors.

## 5.4 Numerical Results

In this section, we present numerical analysis of the proposed algorithms, as well as a general-purpose secure multi-party computation (SMC) algorithm, in the case of joining two equal-size databases $X_1$ and $X_2$ privately where no information other than prior information $L$, $S$ and $M$ is leaked. We do not compare the performance of our algorithms with that of the algorithms in [4] since they have different assumptions and provide different levels of privacy. We consider three different settings of $L, S$ and $M$ to study how different memory, input and output sizes affect the cost of each algorithm.

| | setting 1 | setting 2 | setting 3 |
|---|---|---|---|
| $L$ | 640K | 640K | 2.56M |
| $S$ | 6.4K | 6.4K | 25.6K |
| $M$ | 64 | 256 | 256 |

Table 5.2: Different settings of $L, S$ and $M$ tested in the numerical experiments.

We explain the purposes of pairs of the settings in Table 5.2. Setting 2 has a memory size four times of that of Setting 1, with everything else being the same. We wish to study how each algorithm responds to changes in $\mathcal{T}$'s memory size. Setting 2 and 3 have

the same memory size, while the input and output sizes of Setting 2 are a quarter of those of Setting 3 respectively.

We choose the algorithm in [32, 34] as the reference SMC algorithm. It is most efficient to the best of our knowledge. As suggested in [32], its communication cost is given by

$$\xi_1 \kappa_0 L G_e(\varpi) + 32\xi_1\kappa_1(\varpi\sqrt{L}) + 2\xi_2\xi_1\kappa_1(S\varpi), \tag{5.8}$$

where $\kappa_0 = 64, \kappa_1 = 100$ are two security parameters, $G_e(\varpi) = 2\varpi$, $\varpi$ is the length of tuples in bits, $\xi_1$ and $\xi_2$ are the parameters to control the privacy preserving level. As the communication cost we compute here is in terms of tuples, $\varpi$ simply takes the value of one. To have a privacy preserving level of $1 - 10^{-20}$, we take $\xi_1 = \xi_2 = 67$. We compute the SMC's communication cost using Eqn. 5.8 with the above parameter settings.

Algorithm **4**, **5** and **6** perform privacy preserving joins at different costs. The costs of Algorithm **4** and **5** are computed straightforwardly using the corresponding formulas in Table 5.1. The cost of Algorithm **6** is computed as a function of the privacy preserving parameter $\varepsilon$.

For arbitrary $\varepsilon \in [0, 1]$, we compute $n^*$ by solving the problem in Eqn. 5.6. Upon knowing $n^*$, we find the optimal swap size $\Delta^*$ for an oblivious sort by solving the optimization problem in Eqn. 5.1, and compute the cost of Algorithm **6** using the corresponding formula in Table 5.1. Under the example settings, we show the minimum communication cost as a function of $\varepsilon$ in logarithmic scale in Figure 5.4.

As seen from Figure 5.4, for the same amount of increase in $\varepsilon$, the cost reduction of Algorithm **6** in setting 1 with a smaller memory size $M$, is more significant than that in

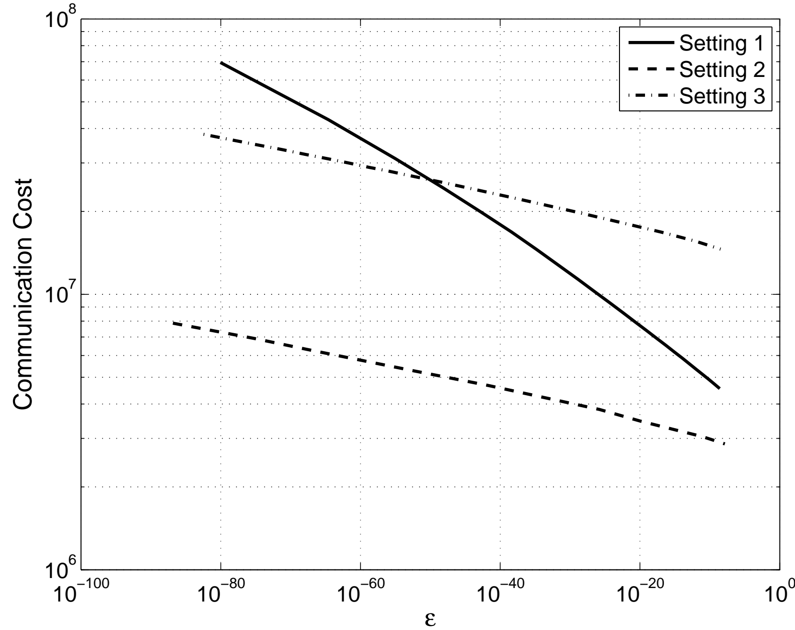


Figure 5.4: Computational cost of Algorithm **6** (logarithmic scale) as a function of privacy preserving parameter $\varepsilon$, under different settings of $L, S$ and $M$.

setting 2 with a larger $M$. This implies that tuning privacy preserving level is more effective in reducing the communication cost of systems with a small $M$ with respect to the number of join results $S$, which we believe is typical in practice.

In practice, an $\varepsilon$ in the range of $10^{-10}$ to $10^{-20}$ achieves a good level of privacy preserving. We choose $\varepsilon$ to be $10^{-10}$ and $10^{-20}$ to show the performance of Algorithm **6** under different privacy preserving levels. The results are shown in Table 5.3, together with the cost of Algorithm **4**, **5** and **6**.

As seen from Table 5.3, regardless of the increase in memory size $M$, the cost of Algorithm **4** is the highest and remains the same with fixed input and output sizes. The cost of Algorithm **5** changes inversely proportional to $M$. With a strict privacy preserving level of $\varepsilon = 10^{-20}$, Algorithm **6** has the minimal cost which is orders of magnitudes less than those

| | setting 1 | setting 2 | setting 3 |
|---|---|---|---|
| SMC in [32] | $1.1 \times 10^{10}$ | $1.1 \times 10^{10}$ | $4.5 \times 10^{10}$ |
| **4** | $2.3 \times 10^{8}$ | $2.3 \times 10^{8}$ | $1.2 \times 10^{9}$ |
| **5** | $6.4 \times 10^{7}$ | $1.6 \times 10^{7}$ | $2.6 \times 10^{8}$ |
| **6** ($\varepsilon = 10^{-20}$) | $7.4 \times 10^{6}$ | $3.4 \times 10^{6}$ | $1.8 \times 10^{7}$ |
| **6** ($\varepsilon = 10^{-10}$) | $4.6 \times 10^{6}$ | $2.8 \times 10^{6}$ | $1.5 \times 10^{7}$ |
| Cost reduction: **6** ($\varepsilon = 10^{-20}$) v.s. **5** | 88% | 79% | 93% |

Table 5.3: Communication costs of Algorithm **4**, **5** and **6**, for different settings of $L, S$ and $M$.

of the other two algorithms in the experiments. Notice even the most expensive Algorithm **4** already outperforms the reference SMC algorithm by at least one order of magnitude in terms of communication cost[2]. This supports our argument that our proposed algorithms is much more efficient than SMC.

The last row in Table 5.3 represents the cost reduction of Algorithm **6** with $\varepsilon = 10^{-20}$ against Algorithm **5**. We observe that the advantage of Algorithm **6** over Algorithm **5** is more significant when $M$ is much less than $S$, and when the problem scale is large, i.e., when $L$ and $S$ are large. We believe this is typical in most practical scenarios. These observations confirm our discussions in Section 5.3.4.

[2] Careful readers might notice the cost of SMC is for communication between participating databases, while the costs of the proposed algorithms are for communication between $\mathcal{T}$ and $\mathcal{H}$. In practice, the former could be more significant than the latter in nature and a direct comparison between them is unfair for our proposed algorithm. For simplicity, we do not differentiate these two types of communications. The observations and conclusions shall still hold if we do.

# Chapter 6

# Conclusions

This thesis explores some issues associated with privacy preserving joins. But many areas remain to be explored. Here, we list a few particularly interesting challenges:

- Our proposed algorithms use nested loop joins. Nest loop joins impose a lower bound on the communication cost. An intuitive way to construct privacy preserving join algorithms is to modify existing efficient join algorithms, such as sort merge join and hash join. However, we showed informally in Section 4.5.1 (page 37) that such adaptation leaks information. It would be interesting to formally prove a lower bound on the communication cost for privacy preserving joins in a secure coprocessor setting.

- Before we prove a lower bound on the performance, are there faster algorithms than what we have proposed? Among the proposed algorithms, Algorithm A **6** is efficient for practical input and output sizes. However, it makes two passes over the cartesian product of the two input tables. A one pass algorithm would dramatically reduce the I/O overhead. Does a one pass algorithm exit?

- We propose three algorithms for different design parameters for our general notion of privacy preserving joins in Chapter 5. These three algorithms compute joins involving arbitrary predicates. Typically, efficient algorithms exist for specific secure multiparty computations. Are there more efficient algorithms to compute specific joins, e.g., one of the most common joins, equijoins?

- We study join operations in this thesis. It would be interesting to develop algorithms for other database operations, particularly aggregation. Aggregation queries output statistics over the join of two tables. It is not necessary to materialize the join result, but only to give statistics over the join table. In this case, we only need to worry about leaking information when accessing the input tables, but not the output tables. Do efficient algorithms exist for this simplified task?

- We explore the parallelism issue when more than one secure coprocessors are attached to a host (page 37 and 68). Algorithms A**2** and A**5** are easy to parallelize with a linear speed-up in the number of processors. Algorithms A**1**, A**3**, A**4**, and A**6** use bitonic sort [7] in a secure coprocessor setting to sort data obliviously. We explain how to sort obliviously in parallel using bitonic sort (page 68). However, implementing a parallel bitonic sort is tricky due to synchronization issue. Synchronization can also add significant complexity. It would be interesting to design an efficient parallel bitonic sort algorithm, implement it, and study its performance.

- We employ a computation model which involves minimum interaction between data owners and the privacy preserving service provider. In this model, data owners send data to the service provider which computes the join and returns the result to des-

ignated recipients. Other possible computation models may involve data owners in the computation process such that they collaborate with the service provider to compute the join. Are there more efficient privacy preserving join algorithms in this more interactive computational models?

- We make certain privacy and performance trade-offs in some of our algorithms. Algorithms A**1**, A**2**, and A**3** leak a controlled amount of information. Algorithm A**6** trades little privacy with significant performance gain. We wonder if other interesting trade-offs exist such that relaxing the privacy requirements in certain ways yields considerable amount of performance gain.

- We give analyses on the performance of our algorithms. It would be interesting to implement our algorithms on an IBM secure coprocessor and study the real performance and see how the measured performance relate to our prediction.

# Bibliography

[1] Health Insurance Portability and Accountability Act of 1996, United States Public Law 104-191.

[2] N. Abe, C. Apte, B. Bhattacharjee, K. Goldman, J. Langford, and B. Zadrozny. Sampling approach to resource light data mining. In *SIAM Workshop on Data Mining in Resource Constrained Environments*, 2004.

[3] R. Agrawal, D. Asonov, P. Baliga, L. Liang, B. Prost, and R. Srikant. A reusable platform for building sovereign information sharing applications. In *1st Workshop on Databases in Virtual Organisations*, June 2004.

[4] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In *Proceedings of the 22nd International Conference on Data Engineering*, Atlanta, Georgia, April 2006.

[5] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data*, San Diego, CA, June 2003.

[6] J. P. Anderson. Computer security technology planning study. Technical Report ESD-

TR-73-51, Vol II, HQ Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, MA, Oct. 1972.

[7] K. E. Batcher. Sorting networks and their applications. In *Proc. of AFIPS Spring Joint Comput. Conference, Vol. 32*, 1968.

[8] B. Bhattacharjee, N. Abe, K. Goldman, B. Zadrozny, V. R. Chillakuru, M. del Carpio, and C. Apte. Using secure coprocessors for privacy preserving collaborative data mining and analysis. In *Proc. of the 2nd international workshop on Data management on new hardware*, 2006.

[9] K. C. Chang and S. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data*, Madison, Wisconsin, June 2002.

[10] C. Clifton, M. Kantarcioglu, X. Lin, J. Vaidya, and M. Zhu. Tools for privacy preserving distributed data mining. *SIGKDD Explorations*, 4(2):28–34, Jan. 2003.

[11] D. J. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. of the 11th Conference on Very Large Databases, Stockholm*, 1985.

[12] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[13] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology — EUROCRYPT 2004*, pages 1–19. Springer-Verlag, May 2004.

[14] V. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes, 2000.

[15] K. Goldman and E. Valdez. Matchbox: Secure data sharing. *IEEE Internet Computing*, 8(6):18–24, 2004.

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

[17] O. Goldreich. *Foundations of Cryptography*, volume 2: Basic Applications. Cambridge University Press, May 2004.

[18] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proc. of 19th STOC*, pages 218–229, 1987.

[19] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

[20] P. Gutmann. An open-source cryptographic coprocessor. In *USENIX*, 2000.

[21] B. A. Huberman, M. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *Proc. of the 1st ACM Conference on Electronic Commerce*, pages 78–86, Denver, Colorado, November 1999.

[22] IBM Corporation. IBM 4758 Models 2 and 23 PCI cryptographic coprocessor, 2004.

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

[24] A. Iliev and S. Smith. Privacy-enhanced credential services. In *2nd Annual PKI Resarch Workshop, NIST, Gaithersburg*, Apr. 2003.

[25] A. Iliev and S. Smith. More efficient secure function evaluation using tiny trusted third parties. Dartmouth Computer Science Technical Report TR2005-551, Department of Computer Science, Dartmouth University, 2005.

[26] C. S. Jutla. Encryption modes with almost free message integrity. *Lecture Notes in Computer Science*, 2045:529–544, 2001.

[27] S. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, 1980.

[28] L. Kissner and D. Song. Privacy-preserving set operations. In *Advances in Cryptology — CRYPTO 2005*.

[29] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1), March 1983.

[30] Y. Li and M. Chen. Privacy preserving joins. In *Proc. of the 24th Int'l Conference on Data Engineering (ICDE 2008)*, Cancun, Mexico, April 2008.

[31] Y. Li, J. D. Tygar, and J. M. Hellerstein. Private matching. In D. Lee, S. Shieh, and J. D. Tygar, editors, *In Computer Security in the 21st Century*, pages 25–50. Springer, 2005.

[32] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Usenix Security '2004*, Aug. 2004.

[33] E. R. Palmer. An introduction to Citadel—a secure crypto coprocessor for workstations. Technical Report RC18373, IBM T. J. Watson Research Center, 1992.

[34] B. Pinkas. Fair secure two-party computation. In *Advances in Cryptology – EUROCRYPT'2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 87–105. Springer-Verlag, May 2003.

[35] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security*, 6(3):365–403, August 2003.

[36] B. Schneier. *Applied Cryptography.* John Wiley, second edition, 1996.

[37] S. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. Research Report RC 21102, IBM T.J. Watson Research Center, Yorktown Heights, New York, Feb. 1998.

[38] S. W. Smith. *Trusted Computing Platforms: Design and Applications.* Springer Science+Business Media, Inc., 233 Spring Street, New York, NY 10013, USA, 2005.

[39] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3), Sept. 2001.

[40] J. D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *Proc. of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Networked Multimedia Environment*, April 1993.

[41] J. D. Tygar, B. Yee, and A. Spector. Strongbox: Support for self-securing programs. In *Proc. of the (First) USENIX UNIX Security Workshop*, August 1988.

[42] WetStone Technologies. TIMEMARK timestamp server, 2003.

[43] S. R. White and L. Comerford. ABYSS: A trusted architecture for software protection. In *Proc. of the IEEE Symposium on Security and Privacy*, 1987.

[44] S. R. White, S. H. Weingart, W. C. Arnold, and E. R. Palmer. Introduction to the Citadel Architecture: Security in Physically Exposed Environments. Technical Report RC16672, IBM T. J. Watson Research Center, 1991.

[45] A. C. Yao. How to generate and exchange secrets. In *Proc. of the 27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Canada, October 1986.

[46] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *FOCS 1982*, 1982.

[47] B. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon Univerisity, May 1994.

[48] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *the 1st USENIX Workshop on Electronic Commerce*, July 1995.

# Glossary

**OA** The Outbound Authentication (OA) mechanism provided by the secure coprocessor ensures that it is indeed executing a known, trusted version of the application code, running under a known, trusted version of the OS, and loaded by a known, trusted version of the bootstrap code within a particular untampered platform.

**PPJ** Privacy Preserving Joins (PPJ) enable mutually distrustful entities to join their data in a privacy preserving way such that no party learns more than what can be deduced from its own input and output of the join computation alone.

**SMC** In Secure Multi-party Computation (SMC) mutually distrustful parties collectively perform a computation over their private data such that no party learns more than what can be inferred from its own input and output of the computation alone.

**TTP** In cryptography a trusted third party (TTP) is an entity which facilitates interactions between two parties who both trust the third party; they use this trust to secure their own interactions.

# Key Notations

$\perp$ Our algorithms encrypt a decoy plaintext $\perp$ and output it if necessary to prevent information leakage. Decoys are decrypted and filtered out by the recipient. They may take the form of a fixed string pattern.

$\gamma$ Define $\gamma = max(1, \lceil N/(M - \delta)\rceil)$.

$\mathcal{T}$ $\mathcal{T}$ represents a secure coprocessor.

$A$ Party $P_A$ owns a private relation $A$.

$a$ Tuple $a$ represents a tuple in Table $A$.

$B$ Party $P_B$ owns a private relation $B$.

$b$ Tuple $b$ represents a tuple in Table $B$.

$C$ Table $C$ represents the result of joining tables $A$ and $B$.

$c$ Tuple $c$ represents a tuple in Table $C$.

$\mathcal{D}$ Assume $J$ input databases $X_1, \ldots, X_J$. $\mathcal{D} = X_1 \times \cdots \times X_J$.

$D$ Table $D$ is a private relation and an input to a privacy preserving join operation.

$F$ $F = M + 1 - \delta$ represents $\mathcal{T}$'s free memory.

$\mathcal{H}$ The host $\mathcal{H}$ is a general purpose computer to which a secure coprocessor is attached and provides additional memory and disk space for the secure coprocessor.

$I$ The index $I = \{1, \ldots, L\}$ is associated with the input table $\mathcal{D}$.

`iTuple` Tuple `iTuple` represents an element in the input table $\mathcal{D}$.

$L$ Assume $J$ input databases $X_1, \ldots, X_J$. Let $\mathcal{D} = X_1 \times \cdots \times X_J$, $L = |\mathcal{D}|$.

$M$ To simplify exposition, we will assume that the tuples of tables $A$, $B$, and $C$ are of the same size and that the free memory of the secure processor can hold at most $M + 2$ such tuples.

$N$ $N$ is the maximum number of tuples from Table $B$ that match a tuple from Table $A$.

`oTuple` Tuple `oTuple` represents an output tuple and can be either a real join result or a decoy.

$P_A$ Party $P_A$ has a private relation $A$ which is an input to a privacy preserving join.

$P_B$ Party $P_B$ has a private relation $B$ which is an input to a privacy preserving join.

$P_C$ Party $P_C$ represents the recipient of a privacy perserving join operation over private relations $A$ and $B$ and is not $P_A$ nor $P_B$.

$Q$ Define $Q$ to be the largest integer such that $Q(1+N) \leq F$, i.e., $\mathcal{T}$ can hold $Q$ tuples in $A$ and all of their matching tuples in $B$ up to $QN$ matching tuples.

$R$ The set of the join tuples is $R$.

$S$ $S = |R|$ represents the number of the join tuples.