Privacy Preserving Joins

Yaping Li
Minghua Chen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-137
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-137.html

November 22, 2007

Privacy Preserving Joins

Yaping Li and Minghua Chen
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA 94720
{yaping, minghua}@eecs.berkeley.edu

Abstract

In this paper, we design a system to perform privacy preserving joins of data from mutually distrustful organizations, leveraging the power of a secure coprocessor. The only trusted component is the secure coprocessor.

Under this setting, we critique a questionable assumption in a previous privacy definition [2] that leads to unnecessary information leakage. We then remove the assumption and propose a new justifiable definition. Based on this new definition, we propose three provable 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. We evaluate the performance of our algorithms by numerical examples and show the performance superiority of our approach over that of the secure multi-party computation.

1 Introduction

Information integration across databases owned by mutually distrustful entities is important in many applications. We consider the problem of how entities compute arbitrary joins function using their data in a *privacy preserving* way such that no party learns more than what can be deduced from its input and output alone. Two motivating applications of privacy preserving joins in airport security and national healthcare are presented in [2].

One straight forward solution to perform privacy preserving join is to rely on a Trusted Third Party (TTP) to whom all parties submit their inputs. This TTP then computes the desired function and distributes the results. This approach is in general easy to implement and efficient. Yet the level of trust placed on a TTP is typically too enormous to be ubiquitously accepted by all parties. In addition, a TTP may be compromised for being an attractive target of valuable information.

Another approach is along the lines of secure multi-party computation problem where parties collectively perform a computation over their data [11]. Generic protocols were designed to show the plausibility of such approach [11] and solutions to various constrained cases of the more general challenge of multi-party computation problem were proposed [9, 17]. This approach assumes a low level of trust among the parties; however, the computation and communication complexities of this approach are normally too high for them to be practical.

A natural question to ask is whether there exist solutions that strike a balance between the level of required trust and performance.

In this paper, we explore answers to this question by presenting a system that functions as a TTP, with a secure coprocessor being the only trusted component. A secure coprocessor is a programmable general purpose computing environment that withstands physical and logical attacks. Examples of such commercially available devices are IBM 4758 secure coprocessors [14], which has a 4MB internal memory, and its latter generation IBM 4764 cryptographic coprocessors [15], whose internal memory size is 64MB.

We argue in this paper 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 secure multi-party computation approach as in [18, 20].

Developing secure applications on a secure coprocessor faces two main challenges. First, the limited memory capacities of secure coprocessors preclude the trivial solution of performing a computation inside a secure coprocessor with ever growing database size inputs and outputs. Second, the access pattern between a secure coprocessor and its host machine may serve as a covert channel to convey useful information to an observer sitting at the host machine. Hiding the the access pattern has always been a challenging task [12, 16].

We list our contributions in this paper as follows:

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

1.1 Related Work

The problem of privately computing various set related operations generated significant interests in the research community. Two party protocols for intersection, intersection size, equijoin, and equijoin size were introduced in [3] for honest-but-curious adversarial model. Some of the proposed protocols leak information [2]. Similar protocols for set intersection have been proposed in [7, 13]. Efficient two party protocols for the private matching problem which are both secure in the malicious and honest-but-curious models were introduced in [9]. Efficient private and threshold set intersection protocols were proposed in [17]. Most of these protocols are equality based while we study algorithms that compute arbitrary join predicates.

Various applications have used a secure coprocessor in their design. They include auditable digital time stamping [22], secure e-commerce [23], secure fine-grained access control [10], secure data mining [1], and private information retrieval [4, 21]. Tiny trusted devices were used for secure function evaluation in [16].

The most relevant to our work is [2] in which Agrawal et al. attempted a privacy definition for join algorithms designed to run on a secure coprocessor and proposed several algorithms provably secure with respect to their definition. As pointed out in Section 2.3, the proposed algorithms leak certain information unnecessarily, achieving lower level of security as compared to our proposed algorithms. As such, we do not compare the performance of our proposed algorithms with that of the algorithms in [2].

Also using secure coprocessors in [6], light weight data mining algorithms were developed for privacy preserving collaborative data mining and analysis. Our work is complementary to this approach. We provide solutions to join various databases in a privacy preserving way to produce results which are processed into the input to the light weight data mining algorithms.

2 Problem Formulation

This section gives an overview of the system, introduces the thread model, and defines privacy preserving joins.

2.1 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 multiple service requestors. A service provider includes a secure coprocessor $\mathcal{T}$ and a host $\mathcal{H}$ to which the secure coprocessor 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 paper. 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 [8].

2.2 Threat Model

We distinguish two types of standard adversary models in this paper: honest-but-curious and malicious adversaries [11]. We then reduce a malicious adversary to an honest-but-curious adversary.

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. We explain how to detect such deviation in our problem setting next.

2.2.1 Detecting Malicious Behavior

Reducing a malicious adversary to an honest-but-curious one is done by using cryptographic tools on $\mathcal{T}$. 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. As proposed in [2], we use authenticated encryption to detect memory tampering. Upon detection of such tampering, $\mathcal{T}$ terminates the program execution immediately.

2.2.2 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 preventing timing attacks. One example of a timing attack is that an adversary may tell whether two tuples match or not if it observes that $\mathcal{T}$ takes different amount of time when comparing two tuples that match and those 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 [10]. To keep the algorithm descriptions simple, we will not show the steps that burn CPU cycles.

The second issue is hiding from an adversary $\mathcal{H}$'s memory content. This is done by symmetric key encryption [19].

The third issue is preventing 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.

2.3 Definition of Privacy Preserving

We describe the *safety definition* formalized in [2] and point out two privacy related and one performance problems 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.

2.3.1 Problems in Previous Definition

The authors in [2] stated that for an algorithm to be safe, 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. The authors formalized the intuition into the following definition which we quote for the sake of completeness:

Definition 1. *[Safety of a Join Algorithm] Assume we have database relations A, B, C and D, where $|A| = |C|$, $|B| = |D|$, A and C have identical schema, as do B and D. For any given N (representing the maximum number of tuples in B (resp. D) that match a tuple in A (resp. C)), let J_{AC} (respectively, J_{CD}) be the ordered list of server locations read and written by the secure coprocessor during the join of A (resp. C) and B (resp. D). The join algorithm is safe if J_{AC} and J_{CD} are identically distributed.*

Recall that 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 safe join algorithm is correct, we point out two problems in Definition 1 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 [2] 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. The authors in [2] did not provide justifications for revealing N for this privacy critical problem. We further 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 1 does not explicitly prescribe the join result which allows the algorithms in [2] to produce a superset of the real join result and leak information. The algorithms in [2] are nested loop join based. For a tuple $a \in A$, if a matches $N' < N$ tuples in B, the algorithms in [2] generate 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 [2], 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 [2] outputs a fixed amount of $N|A|$ join tuples regardless of the actual join result size. This is not ideal for most of the applications. In 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 [2] 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.

2.3.2 Proposed Definition

We define privacy preserving joins with respect to honest-but-curious adversaries. We distinguish our definition from Definition 1 in three aspects: a) removal of the assumption

of N, b) an explicit requirement of a join algorithm to compute exact join results with no additional padding, and c) extension to the multi-party scenario.

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

3 Preliminaries

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

3.1 Assumptions and Notations

Assume J participating databases $D_1, \ldots, D_J$. Let $\mathcal{D} = D_1 \times \cdots \times D_J$, $L = |\mathcal{D}|$, $I = \{1, \ldots, L\}$, and `iTuple` be an element in $\mathcal{D}$. Let R denote the set of the join results with size S. For 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 `iTuples`, 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 `oTuples`. In our discussion of communication cost, we state the cost in terms of tuples.

3.2 Oblivious Sort

We use oblivious sorting as a cryptographic building block in our proposed algorithms. Our application of oblivious sort is different than that in [2].

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 ω elements using the Bitonic sort algorithm proceeds in stages [5]. Assuming ω is a power of 2, at each stage, the ω 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 possibly swapped. There are a total of approximately $\frac{1}{2}(\log_2 \omega)^2$ stages and $\frac{1}{2}\omega$ comparisons at each stage. Therefore, the cost of oblivious Bitonic sort is $\frac{1}{4}\omega(\log_2 \omega)^2$ comparisons and $\omega(\log_2 \omega)^2$ element transfers between the secure coprocessor and the host.

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 in, decrypts it, and outputs nothing if it is a decoy and re-encrypts it and outputs the encrypted tuple otherwise. An adversary 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 μ target elements in a list of length ω 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 Δ is the size of an 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 μ positions in the buffer. Since at most μ elements are kept, the bottom Δ elements in the swap area can be overwritten. We copy another Δ elements

from the source list, and overwrite the bottom Δ elements. We obliviously sort the buffer again to keep all the wanted elements in the top positions.

This process is continued until all elements in the source list is processed. The top μ elements in the buffer are 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 ω and μ, it is possible to minimize the total number of element transfers between the secure coprocessor and the host, by carefully selecting Δ. The optimal Δ, denoted as Δ^*, can be found by solving the following optimization problem:

$$\Delta^* = \arg\min_{\Delta>0} 4\mathcal{C}_{(\omega,\mu)}(\Delta). \quad (1)$$

Since Δ^* also minimizes $\log[\mathcal{C}_{(\omega,\mu)}(\Delta)]$, Δ^* 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, Δ^* is the first quadrant intersection point of two curves $\frac{\Delta}{\mu}$ and $\frac{\log_2(\mu+\Delta)}{2}$, and does not depend on ω.

3.3 Generate Random Order

Algorithm **A3** in Section 4 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.

4 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 3.

4.1 Algorithm A1 for Secure Coprocessors with Small Memory

Intuitively, if $\mathcal{T}$ always outputs a tuple regardless of whether there is a real join or not, then the communication patterns between $\mathcal{T}$ and $\mathcal{H}$ are independent of the contents of the participating databases. Consequently, an adversary does not learn any information on the contents of the participating databases, by observing the traffic between the $\mathcal{T}$ and $\mathcal{H}$. We turn the intuition into Algorithm **A1**.

Algorithm 1 : For Secure Coprocessors with Small Memory

```
1: for each iTuple do
2:    if satisfy(iTuple) then
3:       write join(iTuple) to H
4:    else
5:       write a decoy to H
6:    end if
7: end for
8: optimally oblivious sort outputs giving priority to decoys
9: remove decoys and output S results
```

For participating databases $D_1,\ldots,D_J$, $\mathcal{T}$ sequentially reads `iTuples` 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 **A1** always outputs L `oTuples`, regardless there are only S real results.

After reading all tuples, $\mathcal{T}$ has output all S real results and $L-S$ decoys. Next, $\mathcal{T}$ 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 **A1** does not take advantage of a large memory, resulting in significant communication cost when S is much smaller comparing to L, which is typically in practice.

4.1.1 Proof of Security

Proof. From the description of Algorithm **A1**, 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 $D_1, \ldots, D_J$. Algorithm **A1** satisfies Definition 2 and is privacy preserving. □

4.1.2 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 Δ^* is the optimal swap size given by Eqn. 1 with $\omega = L$ and $\mu = S$. We summarize the total communication cost in the following:

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

4.2 Algorithm A2 for Secure Coprocessors with Large Memory

We observe that a significant portion of the communication cost of Algorithm **A1** 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 $D_1, \ldots, D_J$, $\mathcal{T}$ sequentially reads `iTuples` 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 `iTuples` between two flushes, then an adversary who observes the communication patterns knows that there are M results in these N `iTuples`. 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 `iTuples`. $\mathcal{T}$ keeps repeating this process until it outputs all S join results.

Consequently, M results are written out to the host machine every L tuples; the 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 **A2** are shown below.

Algorithm 2 : For Secure Coprocessors with Large Memory

```
1:  // lindex: largest index of iTuple that leads to a join
2:  // pindex: index of iTuple of previous join
3:  lindex := 0; pindex :=-1
4:  while pindex < lindex do
5:     prepare to read one round of iTuple in a fixed order
6:     for each iTuple do
7:        if satisfy(iTuple) && current index > pindex
          then
8:           store join(iTuple) in memory
9:           if current index > lindex then
10:             lindex := current index
11:          end if
12:       end if
13:       if memory is full then
14:          write M results to H
15:          pindex := current index
16:       end if
17:    end for
18: end while
19: output S results
```

4.2.1 Proof of Security

Proof. The secure coprocessor reads L `iTuples` sequentially a total of $\lceil S/M \rceil$ times. After each but the last scan of the L `iTuples`, 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 $D_1, \ldots, D_J$. Algorithm **A2** satisfies Definition 2 and is privacy preserving. □

4.2.2 Communication Cost

The write cost S of Algorithm **A2** is clearly minimum. 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 `iTuples` in each cycle. Hence, the total communication cost is then given by

$$S + \lceil \frac{S}{M} \rceil L. \qquad (3)$$

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 1. The communication cost approaches the minimum one $S + L$, as M approaches S.

4.3 Algorithm A3 for Trading Privacy Preserving Level with Efficiency

In Algorithm **A2**, during each write cycle, $\mathcal{T}$ has to read all `iTuples` before it outputs the M results stored in its memory. When $M \ll S$, $\mathcal{T}$ spends a large number of write cycles to output all S join results, resulting in a high read cost. One way to improve the efficiency is to shorten the write cycle.

We assume $\mathcal{T}$ knows L and S, and is able to randomly read every `iTuples` 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 3.3.

To achieve better write efficiency, $\mathcal{T}$ can first partitions L randomly ordered `iTuples` 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 `oTuples` 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 **A2**.

However, in the case where $K > M$, $\mathcal{T}$ will not able to output all the join results in one pass and will need to access this segment again to output the missing results. Alternatively, $\mathcal{T}$ can use Algorithm **A2** 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 blemish case. Let $x(n)$ be a random variable denoting the number of join results in n randomly selected `iTuples`. Denote the event of having k results in these n `iTuples`. 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

$$P[x(n) = k] = \frac{1}{\binom{L}{n}} \binom{L-S}{n-k} \binom{S}{k}, \tag{4}$$

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

$$P[x(n) \leq M] = \frac{1}{\binom{L}{n}} \sum_{k=1}^{M} \binom{L-S}{n-k} \binom{S}{k} \tag{5}$$

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 `oTuples`, which in turn reducing 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:

$$n^* = \arg\min_{n>0} n \text{ subject to } P_M(n) < \varepsilon. \tag{6}$$

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

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

Algorithm 3 : For Trading Privacy Preserving Level with Efficiency

```
1: screen all iTuples to get L and S
2: compute n* from Eqn. 6
3: use PRNG to generate a random order for reading iTuples
4: p1 :=0; p2 :=0
5: for each iTuple do
6:    increment p2
7:    if satisfy(iTuple) then
8:       record join(iTuple) in memory
9:    end if
10:   if p2 − p1 == n* || p2 == L then
11:      output max(S, M) real results and decoys to H
12:      p1 := p2
13:   end if
14: end for
15: optimally oblivious sort output giving priority to decoys
16: remove decoys and output S results
```

4.3.1 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 $D_1, \ldots, D_J$.

Algorithm **A3** 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 event to happen is bounded by ε.

Based on the above argument, Algorithm **A3** is privacy preserving with probability at least $1 - \varepsilon$. □

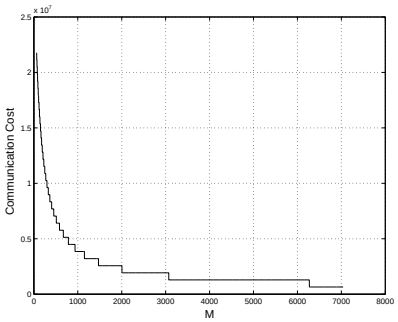

Figure 1. Communication cost of Algorithm A2 as a function of memory size M, under setting $L = 640,000$ and $S = 6,400$.

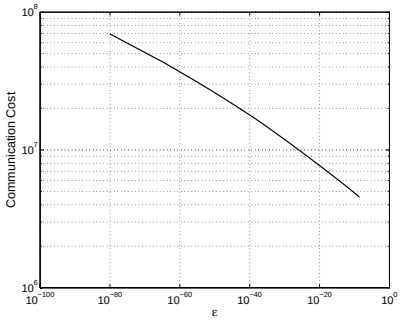

Figure 2. Communication cost of Algorithm A3 as a function of ε, under setting $L = 640,000$, $S = 6,400$, and $M = 64$.

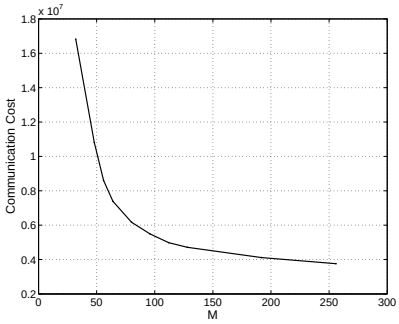

Figure 3. Communication cost of Algorithm A3 as a function of M, under setting $L = 640,000, S = 6,400$ and $\varepsilon = 10^{-20}$.

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

The read cost of Algorithm **A3** 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

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

where Δ^* is the size of the swap area that minimizes the cost for obliviously filtering $\lceil \frac{L}{n^*} \rceil M - S$ decoys. We compute Δ^* by solving the problem in Eqn. 1, with $\omega = \lceil \frac{L}{n^*} \rceil M$ and $\mu = S$. The total communication cost of Algorithm **A3** is then given by

$$2L + \lceil \frac{L}{n^*} \rceil M + \frac{\lceil \frac{L}{n^*} \rceil M - S}{\Delta^*}(S + \Delta^*)[\log_2(S + \Delta^*)] \quad (7)$$

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

As ε increases, the privacy preserving level drops. Algorithm **A3** can increase the segment size n^* while still keeping the chance of encountering blemish cases being less than ε. Consequently, the communication cost of Algorithm **A3** deceases monotonically as ε increases, as shown in Figure 2.

Based on the same argument, the communication cost of Algorithm **A3** increases as ε 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 **A3** reduces to Algorithm **A1**.

From Figure 2 we observe that decreases in the communication cost of Algorithm **A3** is less significant as ε approaches 1. For example, the cost reduction is more than 1.3×10^7 as ε increases from 10^{-60} to 10^{-50}, while the reduction is only less than 10^7 as ε increases from 10^{-20} to 10^{-10}. This implies that it is more profitable to trade privacy preserving level with efficiency when ε is small than when it is large.

As the memory size M increases, Algorithm **A3** 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 **A3** 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 3.

Similar to the relationship between cost reduction and ε, 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.

4.4 Comparison of Algorithms A1, A2 and A3

We compare levels of privacy preserving and communication costs of Algorithms **A1**, **A2**, and **A3** in Table 1.

As seen from Table 1, Algorithm **A1** and **A2** guarantee 100% privacy preserving, while Algorithm **A3** guarantees $(1-\varepsilon)\times$ 100% privacy preserving, which is by design. However, as ε can be chosen to be arbitrarily small to meet practical needs, we believe Algorithm **A3** is practically as secure as Algorithm **A1** and **A2**. For example, if $\varepsilon = 10^{-10}$, then on average, Algorithm **A3** performs a "salvage" action once every 10^{10} trials.

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

Table 1. Level of privacy preserving vs. communication cost.

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

For large L and small M with respect to S, the write efficiency of an algorithm dominates the communication cost. Algorithm **A2** has a low write efficiency; hence, Algorithm **A3** outperforms Algorithm **A2** in terms of communication cost. This is illustrated in the next section.

In cases M is close to S, Algorithm **A2**'s write efficiency M/L is high with respect to the optimal value $S/\ L$. Consequently, Algorithm **A2** might has less communication cost than that of Algorithm **A3**. Algorithm **A2** might be attractive in some scenarios considering its preservation of privacy and ease of implementation for it does not require oblivious sort and random access to `iTuples`.

4.5 Parallelism

For Algorithm **A1**, parallelly processing `iTuples` and generating `oTuples` can be achieved by simply partitioning the `iTuples` into P sets and allocating one set to one coprocessor to process. Oblivious filtering out decoys in parallel requires a specific algorithm other than the sequential process in [5]. Before that, Algorithm **A1** is not be able to fully enjoy the power of multi-coprocessors. For Algorithm **A2**, one $\mathcal{T}$ serves as the coordinator of parallelism. It screens the `iTuples` and calculates the output size S. Without loss of generality, we assume that S divides P and 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 `iTuples` in the same predefined and fixed order. Algorithm **A2** enjoys a linear speed up in performance. For Algorithms **A3**, 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 `iTuples` with an index that falls in its respective range. Parallelism in processing `iTuples` and generating `oTuples` can be achieved. However, similarly to Algorithm **A1**, a parallel algorithm for oblivious filtering out decoys is needed in order for Algorithm **A3** to fully take advantage of the multi-coprocessors.

5 Numerical Results

In this section, we present numerical analysis of the proposed algorithms, as well as a general-purpose secure multiparty computation (SMC) algorithm, in the case of joining two equal-size databases D_1 and D_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 [2] 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 2. Different settings of L, S and M tested in the numerical experiments.

We explain the purposes of pairs of the settings in Table 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 [18, 20] as the reference SMC algorithm. It is most efficient to the best of our knowledge. As suggested in [18], its communication cost is given by

$$\xi_1\kappa_0 L G_e(\varpi) + 32\xi_1\kappa_1(\varpi\sqrt{L}) + 2\xi_2\xi_1\kappa_1(S\varpi), \quad (8)$$

where $\kappa_0 = 64, \kappa_1 = 100$ are two security parameters, $G_e(\varpi) = 2\varpi$, ϖ is the length of tuples in bits, ξ_1 and ξ_2 are the parameters to control the privacy preserving level. As the communication cost we compute here is in terms of tuples, ϖ 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. 8 with the above parameter settings.

Algorithm **A1**, **A2** and **A3** perform privacy preserving joins at different costs. The costs of Algorithm **A1** and **A2** are computed straightforwardly using the corresponding formulas in Table 1. The cost of Algorithm **A3** is computed as a function of the privacy preserving parameter ε.

For arbitrary $\varepsilon \in [0, 1]$, we compute n^* by solving the problem in Eqn. 6. Upon knowing n^*, we find the optimal swap size Δ^* for an oblivious sort by solving the optimization problem in Eqn. 1, and compute the cost of Algorithm **A3** using the corresponding formula in Table 1. Under the example settings, we show the minimum communication cost as a function of ε in logarithmic scale in Figure 4.

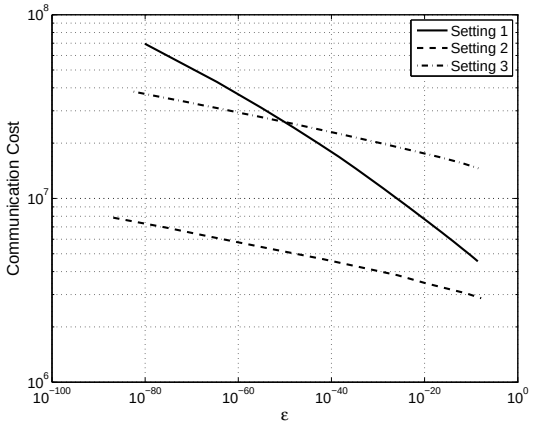

Figure 4. Computational cost of Algorithm A3 (logarithmic scale) as a function of privacy preserving parameter ε, under different settings of L, S and M.

As seen from Figure 4, for the same amount of increase in ε, the cost reduction of Algorithm **A3** in setting 1 with a smaller memory size M, is more significant than that in 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 ε in the range of 10^{-10} to 10^{-20} achieves a good level of privacy preserving. We choose ε to be 10^{-10} and 10^{-20} to show the performance of Algorithm **A3** under different privacy preserving levels. The results are shown in Table 3, together with the cost of Algorithm **A1**, **A2** and **A3**.

As seen from Table 3, regardless of the increase in memory size M, the cost of Algorithm **A1** is the highest and remains the same with fixed input and output sizes. The cost of Algorithm **A2** changes inversely proportionally with M. With a strict privacy preserving level of $\varepsilon = 10^{-20}$, Algorithm **A3** has the minimal cost which is orders of magnitudes less than those of the other two algorithms in the experiments. Notice even the most expensive algorithm **A1** 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.

	setting 1	setting 2	setting 3
SMC in [18]	1.1×10^{10}	1.1×10^{10}	4.5×10^{10}
A1	2.3×10^{8}	2.3×10^{8}	1.2×10^{9}
A2	6.4×10^{7}	1.6×10^{7}	2.6×10^{8}
A3 ($\varepsilon = 10^{-20}$)	7.4×10^{6}	3.4×10^{6}	1.8×10^{7}
A3 ($\varepsilon = 10^{-10}$)	4.6×10^{6}	2.8×10^{6}	1.5×10^{7}
Cost reduction: **A3** ($\varepsilon = 10^{-20}$) v.s. **A2**	88%	79%	93%

Table 3. Communication costs of Algorithm A1, A2 and A3, for different settings of L, S and M.

The last row in Table 3 represents the cost reduction of Algorithm **A3** with $\varepsilon = 10^{-20}$ against Algorithm **A2**. We observe that the advantage of Algorithm **A3** over Algorithm **A2** 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 4.4.

6 Conclusions

In this paper, We study a system that performs privacy preserving joins of data from multiple organizations, leveraging the power of a secure coprocessor. The only trusted component in the system is the secure coprocessor. On one hand, the required trust level for such system is much lower than that for a completely trusted third party. On the other hand, the complexity of computing an *arbitrary* function on multiple parties' data via a secure coprocessor is much lower than that of the general secure multi-party computation approach [18, 20].

Under this setting, we critique a questionable assumption in a previous privacy definition [2] that leads to unnecessary information leakage. We remove such unnecessary assumption, and propose a justifiable definition.

We then propose three algorithms to compute general joins of arbitrary predicates, and prove their correctness

[2] Careful readers might notice the cost of SMC is for communication between participating databases, while the cost 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.

and security under the new definition. Algorithm **A1** and **A2** guarantee 100% privacy preserving but have high computation cost. Algorithm **A3** is able to trade privacy preserving level $1 - \varepsilon$ with communication cost efficiently, where $\varepsilon \in [0, 1]$ is a design parameter. We give explicit expressions for their computation costs, evaluate their performances by numerical examples, and show their significant improvement as compared to that of general multiparty computation approach.

References

[1] 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.

[2] 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.

[3] 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.

[4] D. Asonov. *Querying Databases Privately*. PhD thesis, Springer Verlag, June 2004.

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

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

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

[16] 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.

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

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

[19] National Institute of Standards and Technology. Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. *Special Publication 800-67*, 2004.

[20] 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.

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

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

[23] B. S. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *The First USENIX Workshop on Electronic Commerce*, July 1995.