Sovereign Joins

Rakesh Agrawal[†] Dmitri Asonov[†] Murat Kantarcioglu[‡] Yaping Li[*]
[†] *IBM Almaden Research Center*
[‡] *The University of Texas at Dallas*
[*] *University of California, Berkeley*

Abstract

We present a secure network service for sovereign information sharing whose only trusted component is an off-the-shelf secure coprocessor. The participating data providers send encrypted relations to the service that sends the encrypted results to the recipients. The technical challenge in implementing such a service arises from the limited capability of the secure coprocessors: they have small memory, no attached disk, and no facility for communicating directly with other machines in the network. The internal state of an ongoing computation within the secure coprocessor cannot be seen from outside, but its interactions with the server can be exploited by an adversary.

We formulate the problem of computing join in this setting where the goal is to prevent information leakage through patterns in I/O while maximizing performance. We specify criteria for proving the security of a join algorithm and provide provably safe algorithms. These algorithms can be used to compute general joins involving arbitrary predicates and multiple sovereign databases. We thus enable a new class of applications requiring query processing across sovereign entities such that nothing apart from the result is revealed to the recipients.

1 Introduction

Conventional information integration approaches, as exemplified by centralized data warehouses and mediator-based data federations, assume that the data in each database can be revealed completely to the other databases. Consequently, information sharing across autonomous entities is inhibited due to confidentiality and privacy concerns. The goal of sovereign information sharing [2, 3, 8] is to enable such sharing by allowing queries to be computed across sovereign databases such that nothing apart from the result is revealed. The computation of join of sovereign databases in such a manner is referred to as sovereign join. We cite below two motivating applications of sovereign joins [3]:

- **Security** For national security, it might be necessary to check if any of the airline passengers is on the watch list of a federal agency [21]. Sovereign join may be used to find only those passengers who are on the list, without obtaining information about all the passengers from the airline or revealing the watch list.

- **Healthcare** In epidemiological research, it might be of interest to ascertain whether there is a correlation between a reaction to a drug and some DNA sequence, which may require joining DNA information from a gene bank with patient records from various hospitals. However, a hospital disclosing patient information could be in violation of privacy protection laws, and it may be desirable to access only the matching sequences from the gene bank.

1.1 Desiderata

A system offering sovereign join service has the following desirable attributes:

- The system should be able to handle general joins involving arbitrary predicates. The national security application cited above 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 [4].

1.2 Problem Addressed

We present a secure network service for sovereign information sharing whose only trusted component is a secure coprocessor [15, 26, 32]. IBM 4758 cryptographic coprocessor [17] is an example of a commercially available, tamper-responding secure coprocessor.

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 IBM 4758). 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, therefore, 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.

1.3 Related Work

In principle, sovereign information sharing can be implemented by using techniques for secure function evaluation (SFE) [13, 31]. Given two parties with inputs x and y respectively, SFE computes a function $f(x, y)$ such that the parties learn only the result. SFE techniques are considered to have mostly theoretic significance and have been rarely applied in practice, although some effort is afoot to change the situation [22].

To avoid the high cost of SFE, the approach taken in [3] was to develop specialized protocols for intersection, intersection size, equijoin, and equijoin size. Similar protocols for intersection have been proposed in [8, 16]. A new intersection protocol has been recently proposed in [10]. However, the protocols provided in [3] have the following shortcomings: (1) It is not clear how to extend them to operations involving general predicates as they are hash-based. (2) It is not obvious how to extend them to efficiently handle a large number of parties. (3) They leak information. For example, the equijoin size protocol leaks the distribution of duplicates; if no two values have the same number of duplicates, it can also leak the intersection.

Secure coprocessors have been earlier used in a variety of applications, including secure e-commerce [33], auditable digital time stamping [30], secure fine-grained access control [12], secure data mining [1], and private information retrieval [5, 28]. See [27] for a taxonomy of secure coprocessing applications. The techniques developed therein though are quite different. Note that the capabilities provided in the architectures such as Trusted Computing Group's trusted platform module [29], while complementary, do not solve our problem.

1.4 Paper Layout

The rest of the paper is organized as follows. In Section 2, we specify the adversarial model and give the simplifying assumptions and notations. In Section 3, we illustrate using classical nested loop some of the subtleties of the problem. This investigation enables us to distill the design principles underlying the proposed algorithms. We also define the correctness criteria for proving the safety of the join algorithms.

In Section 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 tradeoffs under different operating parameters.

Section 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 6, we analyze the performance characteristics of the proposed algorithms. We conclude with a summary and directions for future work in Section 7.

2 Preliminaries

This section specifies the adversarial model and our simplifying assumptions and notations.

2.1 Adversarial Model

Our computing model admits any number of data providers and result recipients. Without loss of generality, we will consider the case where two parties P_A and P_B that have private relations A and B are participating in the sovereign join operation and the result C is sent to the party P_C, which is not P_A or P_B. We assume that the join algorithms and the join predicates are known to the parties.

The server $\mathcal{S}$, offering sovereign information sharing, is a general purpose computer. A secure coprocessor $\mathcal{T}$ is attached to $\mathcal{S}$. The only trusted component is the secure coprocessor. All other components, including $\mathcal{S}$, are untrusted. We assume that no party (including$\mathcal{S}$) can observe the state of the computation inside $\mathcal{T}$ or tamper with the code loaded into it.

Communication between $\mathcal{T}$ and P_A, P_B, or P_C is encrypted. Similarly, any temporary value output by $\mathcal{T}$ to $\mathcal{S}$ is also encrypted.

2.2 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 remote attestation mechanism 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 [12].

We assume that P_A and P_B have signed a digital contract [12] 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 [11, 19, 24]. 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 ⟨Nonce, Ciphertext, Authentication Tag⟩ triple (authenticity). The indistinguishability from random strings implies that OCB is semantically secure [24], which ensures with high probability that duplicate tuples will be encrypted differently.

Encryption under OCB [24] 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 $C[m] = T[m] \oplus Y[m]$[first$|T[m]|$bits] where $Y[m] = E_k(\text{len}(T[m]) \oplus g(E_k(0^n)) \oplus Z[m])$, 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 $T = E_k(\text{Checksum} \oplus Z[m])$[first τ bits] where $C[m]0^*$ represents padding the last cipher block to the block size. The first τ 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 $Y[m] = E_k(\text{len}(C[m]) \oplus g(E_k(0^n)) \oplus \bar{Z}[m])$. $P[m] = C[m] \oplus Y[m]$[first $|C[m]|$ bit]. Checksum = $P[1] \oplus \cdots \oplus P[m-1] \oplus C[m]0^* \oplus Y[m]$. Let $T' = E_k(\text{Checksum} \oplus Z[m])$[first τ 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 $\mathcal{S}$ and $\mathcal{T}$. Our algorithms are designed to thwart the adversary from learning anything by observing this interaction.

2.3 Assumptions and Notations

To simplify exposition, we will assume that the tuples of A, B, and C are of the same size and that 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 $\mathcal{S}$, P_A, P_B, and P_C. Assume that P_A and P_B have sent their encrypted relations A and B respectively to $\mathcal{S}$, who has stored them on its local disk. Similarly, $\mathcal{T}$ writes the encrypted join result to $\mathcal{S}$'s disk (invoking the server process running on $\mathcal{S}$), which $\mathcal{S}$ 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 $\mathcal{S}$ by prepending the operation with the keyword `put`; the keyword `get` will indicate a transfer from $\mathcal{S}$ 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 [2] can be used for this purpose.

3 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.

3.1 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 $\mathcal{S}$ 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 $\mathcal{S}$ 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., $\mathcal{S}$ 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.

3.2 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.

3.3 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.

3.4 Correctness Criteria

We discussed earlier 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 [14], we formalize this intuition as follows:

Definition 1 (Safety of a Join Algorithm) *Assume we have database relations A, B, C and D, where $|A| = |C|$, $|B| = |D|$, A and C have identical schema, as do B and D. For any given N (Section 2.3), 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.*

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.

3.5 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 [12]. 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 [24].

Setting N In some applications, N might be known apriori. 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 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 in the inner relation [7].

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 $\mathcal{S}'$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, 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 [6]. 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 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.

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
      putscratch[(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 S to write first N of scratch[] to disk;
end for
```

Encryption Since both A and B are accessed sequentially, they can be encrypted using the procedure described in Section 2.2. 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.

4.1.1 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, always the first N locations of $scratch[]$ are accessed for writing the result tuples to disk. Thus, Definition 1 is satisfied.

4.1.2 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:

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

4.2 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 $\mathcal{S}$. 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 α to be $N/|B|$. Clearly, Algorithm 1 outperforms this variant for small values of α; we do not discuss it further.

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 δ 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 γ times to find all the matches for a. Conceptually, imagine partitioning the tuples from B that match a into γ groups of $\lceil N/\gamma \rceil$ tuples each. During pass i over B, $\mathcal{T}$ computes the ith group of the matched tuples and outputs them to $\mathcal{S}$ 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 2.2.

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 S;
    Request S to write joined[] to disk;
  end for
end for
```

4.3.1 Correctness (Proof Sketch)

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

4.3.2 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:

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

4.3.3 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 $\mathcal{S}$. Define $F = M + 1 - \delta$ where δ 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 4.3.4. So, we keep only one tuple of A in memory and our problem reduces to one of 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 K to be the largest integer such that $K(1+N) \leq F$, i.e., $\mathcal{T}$ can hold K tuples in A and all of their up to KN matching tuples. Then the optimal way to partition the memory is $F_a = K$, $F_b = F - K(1+N)$, and $F_j = KN$.

4.3.4 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 ith group of the PN' matching tuples for each element in L and outputs to $\mathcal{S}$ 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 $\mathcal{S}$, 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 Parallelism

Consider a server in which more than one secure coprocessor is attached to the server. 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.

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.

5.1 False Starts

We explore the adaptation of classical sort-merge join, grace hash join, and the idea of commutative encryption from [3, 8, 16].

5.1.1 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 is 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.

5.1.2 Hash-based Join (Unsafe)

We consider the family of grace hash join algorithms [9, 20]. They begin by partitioning A and B into disjoint subsets called buckets, which have the property that all tuples with the same hash of the join attribute value share the same bucket. 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[18]);
for each $a \in A$ **do**
  $a_{\mathcal{T}} = decrypt(\texttt{get}\ a)$;
  Place $encrypt(a)$ into the ith bucket, where $i = hash(a_{\mathcal{T}}.joinattr)$;
  if the ith bucket is full **then**
    Fill all other buckets with decoy tuples and output all the buckets to $\mathcal{S}$;
  end if
end for

5.1.3 Commutative Encryption (Unsafe)

We now consider an algorithm inspired by the idea of commutative encryption used in [3, 8, 16].

The first encryption is done by the data providers before sending their relations to $\mathcal{S}$. Now, $\mathcal{T}$ executes the algorithm below. The key point is that $\mathcal{T}$ employs symmetric encryption [25] using the same key for re-encrypting the two relations.

Obliviously shuffle A;
for each $a \in A$ **do**
  $a_{\mathcal{T}} = decrypt(\texttt{get}\ a)$;
  `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 (leaks the distribution of the duplicates).

5.2 Sovereign 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

[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.

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 S to write scratch[] to disk;
end for
```

obliviously 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 ith 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 tuple 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 2.2. However, B requires oblivious sorting. Hence, its encryption should use the strategy described for encrypting $scratch[]$ array in Section 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.

5.2.1 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.

5.2.2 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 $\mathcal{S}$ 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:

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

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 γ. 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 α and γ.

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

6.1 $\gamma = 1$

We find that Algorithm 2 dominates the other two algorithms. To see this, set α 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 γ 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.

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 α (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 α increases, γ also increases.

6.3 Equijoins, $\gamma > 1$

Both Algorithms 1 and 3 are insensitive to γ. For comparing them, let us substitute α in the last term in the cost formula for Algorithm 1 with $1/|B|$, the smallest value α 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 α 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 $|B| \geq 1$.

6.4 Performance Relationship Summary

Figure 1 summarizes the performance relationship among the three algorithms. An unlabeled arrow pointing from Algorithm X to Y denotes that X dominates Y. A labeled arrow from X to Y indicates the condition under which X outperforms Y; the performance relationship is reversed if the condition is not satisfied. For simplicity, we label the arrows with only the significant conditions.

6.5 Comparison with Secure Function Evaluation

We now compare the performance of the proposed algorithms to the technique for secure function evaluation

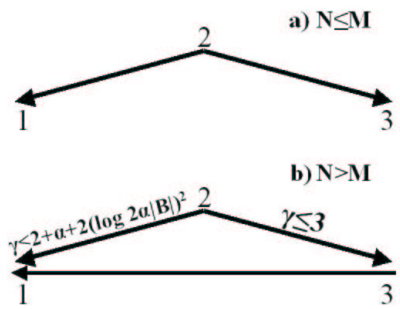

Figure 1. Performance Relationship

(SFE), based on secure circuit evaluation [13, 31]. 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 [22, 23].

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. Assume that the output has $|B|Nw$ bits. Assume 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, $4l|B|^2 G_e(w)$ 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 $4k_o B2G_e(w)$ 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

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

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 [22], and take $G_e(w) = 2w$. For

low values of α, it can be seen that SFE can be orders of magnitude slower.

7 Summary

We presented a secure information sharing service offering sovereign joins, built using off-the-shelf secure coprocessors. Our design satisfies the desiderata for such a service: we can do general joins involving arbitrary predicates across any number of sovereign 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. Our other contributions include:

- 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.

Directions for future work include developing algorithms for other database operations, particularly aggregation.

Acknowledgements We wish to thank Bishwaranjan Bhattacharjee, Chris Karlof, Bruce Lindsay, Guy Lohman, Arnie Rosenthal, and Dan Shiffman for insightful comments and discussions.

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 Nastional Science Foundation through ITR Award IIS-0205647. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of any funding sponsor or the US Government.

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, 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.

[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] 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.

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

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

[7] 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.

[8] 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.

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

[10] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Proc. Advances in Cryptology – EUROCRYPT 2004*, Interlaken, Switzerland, May 2004.

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

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

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

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

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

[16] 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.

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

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

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

[20] 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.

[21] T. Kontzer. Airlines and hotels face customer concerns arising from anti-terrorism efforts. *Information Week*, March 2004.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] S. W. Smith. Secure coprocessing applications and research issues. Los Alamos Unclassified Release RLA-UR-96-2805, Los Alamos National Laboratory, Los Alamos, NM, Aug. 1996.

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

[29] Trusted Computing Group. TCG specification architecture overview, 2004.

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

[31] 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.

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

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