Secure Outsourcing of Some Computations

Mikhail J. Atallah (Fellow), Konstantinos N. Pantazopoulos, Eugene H. Spafford (Senior Member)

COAST Laboratory

Department of Computer Sciences
Purdue University
1398 Department of Computer Sciences
West Lafayette, IN 47907–1398

{mja,kp,spaf}@cs.purdue.edu

Abstract

The rapid growth of the Internet facilitates the *outsourcing* of certain computations, in the following sense: A *customer* who needs these computations done on some data but lacks the computational resources (or programming expertise) to do so, can use an external *agent* to perform these computations. This currently arises in many practical situations, including the financial services and petroleum services industries. The outsourcing is *secure* if it is done without revealing to the agent either the actual data or the actual answer to the computation. In this paper we describe how representative operations matrix multiplication, matrix inversion, solution of a linear system of equations, convolution, and sorting can be securely outsourced in a practical sense.

The general idea is for the customer to do some carefully designed local preprocessing of the data before sending it to the agent, and also some local postprocessing of the answer returned by the agent to extract from it the true answer. The pre- and postprocessing should not take time more than proportional to the size of the input, which is unavoidable because the customer must at least read the input once. The purpose of the preprocessing step that the customer performs locally is to "hide" the real data with suitably chosen noise, sending to the agent the obfuscated data. The purpose of the postprocessing is to extract from the noisy answer returned by the agent the true answer that the customer seeks.

Index Terms — Computer security, data hiding, outsourcing, matrix computations, convolution, sorting

1 Introduction

Outsourcing is a general procedure employed in the business world when one entity, the customer C, chooses to farm out (*outsource*) a certain task to an external entity, the agent A. The reasons for the customer to outsource the task to the agent could be many, ranging from a lack of resources to perform the task locally to a deliberate choice made for financial reasons (it could be cheaper to outsource). Here we consider the outsourcing of certain kinds of computations, with the added twist that the data and the answers sought are to be hidden from the agent who is performing the computations on the customer's behalf. That is, the customer's information (both the data and the results obtained) is proprietary, and it is either the customer who does not wish to trust the agent with preserving the secrecy of that information, or it is the agent who insists on the secrecy so as to protect itself from liability because of accidental or malicious (e.g., by a bad employee) disclosure of the confidential data.

The current practice is that such outsourcing of sensitive and highly valuable proprietary data is commonly done "in the clear," that is, by revealing both data and results to the agent hired to perform the computation. One industry where this happens is the financial services industry, where the proprietary data includes the customer's projections of the likely future evolution of certain commodity prices, interest and inflation rates, economic statistics, portfolio holdings, etc. Another industry is the energy services industry, where the proprietary data is mostly seismic, and can be used to estimate the likelihood of finding oil or gas if one were to drill at the geographic spot in question. The seismic data is so massive that doing multiplication and inversion of such large matrices of data is beyond the computational resources of even the major oil service companies, which routinely outsource these computations to a number of supercomputing centers.

In this paper we propose various schemes for outsourcing to an outside agent a suitably-modified version of the input data, in a way that hides the data from the agent and yet has the property that the answers returned by the agent can easily be used to obtain the true answer – the one corresponding to the true input data. The local computations take time proportional to the size of the input, and the schemes we propose appear to work well experimentally, both from the point of view of data-hiding and from the point of view of numerical stability.

The framework of this paper differs from what is found in the cryptography literature concerning this kind of problem. Secure outsourcing in the sense of [2] follows an information-theoretic approach, leading to elegant negative results about the impossibility of securely outsourcing computationally intractable problems. In addition, the cryptographic protocols literature contains much that is reminescent of the framework of the present paper, with many elegant protocols for cooperatively computing functions without revealing information about the functions' arguments to the other party (cf. the many references in, for example, [22, 19]). Also reminescent of this work is the server-aided computation literature, but most papers there deal with modular exponentiations and not with numerical computing [15, 17, 20, 16, 13, 11, 3, 14, 10]). In this paper's framework, the encryption methods we use are very straightforward and are similar to "one time pad" schemes: For example, when we hide a number x by adding to it a random value r, then we do not re-use that same r to hide another number y (we generate another random number for that purpose). If we hide a sequence of such x's by adding to each a randomly genaretd r, then we have to be careful to use a suitable distribution for the r's: If we know the distribution of the x's, then we can use a distribution for the y's such that $x + y$ is uniformly distributed (and hence reveals nothing about the distribution of x, which might not be the case if r itself had been generated from a uniform distribution).

The random numbers we use for disguises are not shared with anyone: They are merely stored by the customer, and used locally to "undo" the effect of the disguise on the disguised answer received from the external agent. Randomness is not used only to hide a particular numerical value, but also to modify the nature of the disguise algorithm itself. For example, for any part of a numerical computation, we will typically have more than one alternative for performing a disguise (e.g., disguising problem size by shrinking it, or by expanding it, in either case by a random amount). Which method is used is also selected randomly. This implies that, if our outsourcing schemes are viewed as protocols, then they have the feature that one of the two parties in the protocol (the external agent) is ignorant of which protocol the other party (the customer) is performing. Our schemes are designed with the usual requirement that they should work even if their source code is in the public domain.

Our methods are geared towards the numerical problems we consider, all of which are solvable in polynomial time — but in our framework even "polynomial time" computation by the customer is too expensive if it is not *linear* in the size of the input. We thus require that the local computations done by the customer should be as light as possible, i.e., should take time that is proportional to the size of the input (which is unavoidable because the customer must at least read the input once). The time taken by the agent should not simply be polynomial: It should be proportional to the time it would have taken to solve the problem locally (i.e., without outsourcing). We believe that for the problems considered, and compared to the current practice, our proposed schemes are a substantial improvement. The experimental data from our "proof of concept" software implementation seems to confirm the practical viability of our methods.

Finally, our approach also differs from the *privacy homomorphism* approach that has been proposed in the past [18]. The framework of the latter assumes that the outsourcing agent is used as a permanent repository of the data, performing certain operations on it and maintaining certain predicates, whereas the customer needs only to decrypt the data from the external agent's repository to obtain from it the real data. Our framework is different in the following ways:

- The customer is not interested in keeping data permanently with the outsourcing agent; instead, the customer only wants to use temporarily its superior computational resources.
- The customer has some local computing power that is not limited to encryption and decryption. However, this local computing power is far less than that of the outsourcing agent. For example, if the problem domain has to do with $n \times n$ matrices, then we shall typically assume that the customer can afford to perform locally computations that take time proportional to n^2 but not n^3, whereas the outsourcing agent has the resources to perform n^3 operations (and thus can invert such matrices, multiply them, etc).

The matrix and vector operations we consider here should be viewed as base computations, ones that make possible the secure outsourcing of a wide variety of computations that can be decomposed into a sequence of base computations (there are too many such decomposable problems to enumerate here).

Throughout what follows we use random numbers, random matrices, random permutations, etc.; it is always assumed that each is generated independently of the others, and that quality random number generation is used (cf. [7, Chap. 23], [21, [Chap. 12], [6, 12, 5]). It is *not* assumed that they are generated from a uniform distribution, or in fact from any particular fixed distribution. Indeed, for increased security, the exact form of the distribution used would itself be a variable, in the sense that the customer would have a catalog of distributions and would switch from using one to using another, to prevent the external agent A from knowing even the probabilistic characteristics of what C is sending. For example, when generating a large random vector S, the various entries of S should be generated from different distributions in C's catalog of available distributions. If all the n entries of S were generated uniformly in some interval centered at zero, then S would not do a good job of "hiding" another (secret) vector V that it is added to; to see this, simply observe that, in such a case, the sum of the n entries of $V + S$ would be very close to the sum of the n entries of V, thus partially compromising the composition of V.

In sections 2–6 we present our schemes for important computations that can be securely outsourced. These are matrix multiplication, matrix inversion, solution of a linear system of equations, convolution and sorting. We included sorting for theoretical rather than practical interest — we are not aware of anyone who outsources sorting. We do know that the major oil services companies outsource matrix operations and convolutions (it is somewhat surprising to see convolution there, because $O(n \log n)$ computation time is not that expensive, whereas the $O(n^3)$ computation time used by the matrix operations makes them exorbitantly expensive for large n). In each of the sections 2–6, we describe schemes of increasing complexity, starting each section with schemes that make no attempt at hiding the problem's dimension n, and ending it with a description of the modifications needed to hide n.

We assume that the reader is familiar with the basic mathematical objects mentioned below. For a review of the definitions of matrix product, matrix inversion, and their properties, we refer the reader to [9] (which contains many other references). For a review of convolution and its properties, we refer the reader to [1].

2 Matrix Multiplication

Assume that C wants to outsource the computation of the product of two $n \times n$ matrices M_1 and M_2. (At the end of this section we explain how essentially the same method works for non-square matrices.)

Notation 1 *We use $\delta_{x,y}$ to denote the function that equals 1 if $x = y$ and 0 if $x \neq y$ (the so-called "Kronecker delta" function).*

2.1 A Preliminary Solution

The following is a preliminary algorithm for performing matrix multiplication using an external agent. It satisfies the requirement that all local processing by C should take time proportional to the size of the input, in this case $O(n^2)$.

1. C creates (i) three random permutations π_1, π_2, and π_3 of the integers $\{1, 2, \cdots, n\}$, and (ii) three sets of non-zero random numbers $\{\alpha_1, \alpha_2, \cdots, \alpha_n\}$, $\{\beta_1, \beta_2, \cdots, \beta_n\}$, and $\{\gamma_1, \gamma_2, \cdots, \gamma_n\}$.

2. C creates matrices P_1, P_2, and P_3 where $P_1(i,j) = \alpha_i \delta_{\pi_1(i),j}$, $P_2(i,j) = \beta_i \delta_{\pi_2(i),j}$, and $P_3(i,j) = \gamma_i \delta_{\pi_3(i),j}$.

Observe that the inverse of P_1, P_1^{-1}, satisfies

$$P_1^{-1}(i,j) = (\alpha_j)^{-1}\delta_{\pi_1^{-1}(i),j}. \tag{1}$$

Similar relations hold for the inverse of P_2 and of P_3, hence, any entry of the matrices P_1^{-1}, P_2^{-1}, and P_3^{-1}, is available to C in constant time.

3. C computes locally matrix

$$X = P_1 M_1 P_2^{-1}. \tag{2}$$

Observe that left-multiplying a matrix by P_1 takes $O(n^2)$ time and amounts to permuting its rows according to π_1 and then multiplying each i-th resulting row by α_i. Also observe that right-multiplying a matrix by P_2^{-1} also takes $O(n^2)$ time and amounts to permuting its columns according to π_2 and then multiplying each j-th resulting column by $(\beta_j)^{-1}$. Thus

$$X(i,j) = (\alpha_i/\beta_j)M_1(\pi_1(i), \pi_2(j)). \tag{3}$$

4. C computes locally, in $O(n^2)$ time, the matrix

$$Y = P_2 M_2 P_3^{-1}. \tag{4}$$

5. C sends X and Y to A. A computes the product XY, which is

$$Z = XY = (P_1 M_1 P_2^{-1})(P_2 M_2 P_3^{-1}) = P_1 M_1 M_2 P_3^{-1} \tag{5}$$

and sends Z to C.

6. C computes locally, in $O(n^2)$ time, the matrix $P_1^{-1} Z P_3$, which equals $M_1 M_2$.
This completes the algorithm.

The above method may be secure enough for many applications, as A would have to guess two permutations (from the $(n!)^2$ possible such choices) and $3n$ numbers (the α_i, β_i, γ_i) before it can pin down M_1 or M_2.

2.2 An Improved Solution

The following scheme is more elaborate and gives somewhat better security because, in addition to left- and right-multiplying a matrix to be hidden by the sparse random matrices P_i or their inverse, the resulting matrix is further hidden by adding a dense random matrix to it. Of course the above-mentioned multiplication by a P_i matrix or its inverse needs to be done in $O(n^2)$ time, i.e., in time proportional to the size of the input matrices. The details follow.

1. C locally computes matrices $X = P_1 M_1 P_2^{-1}$ and $Y = P_2 M_2 P_3^{-1}$ as was done in the previous, preliminary scheme.

2. C selects two random $n \times n$ matrices S_1 and S_2 (that is, matrices whose entries are random). C also generates four random numbers $\beta, \gamma, \beta', \gamma'$ such that

$$(\beta + \gamma)(\beta' + \gamma')(\gamma'\beta - \gamma\beta') \neq 0.$$

If the above is violated then we discard the four random numbers chosen and we repeat the random experiment of choosing a new set of numbers; observe, however, that there is zero probability that a random choice results in a violation of the above condition, hence the random choice need not be repeated more than $O(1)$ times (in practice, once is usually enough).

3. C computes locally the six matrices $X + S_1$, $Y + S_2$, $\beta X - \gamma S_1$, $\beta Y - \gamma S_2$, $\beta' X - \gamma' S_1$, $\beta' Y - \gamma' S_2$. Then C sends these six matrices to agent A.

4. Agent A uses the six matrices it received in Step 3 to compute

$$\begin{aligned} W &= (X + S_1)(Y + S_2) && (6) \\ U &= (\beta X - \gamma S_1)(\beta Y - \gamma S_2) && (7) \\ U' &= (\beta' X - \gamma' S_1)(\beta' Y - \gamma' S_2) && (8) \end{aligned}$$

and sends the resulting matrices W, U, U' to C.

5. C computes locally matrices V and V' where

$$\begin{aligned} V &= (\beta + \gamma)^{-1}(U + \beta\gamma W) && (9) \\ V' &= (\beta' + \gamma')^{-1}(U' + \beta'\gamma' W). && (10) \end{aligned}$$

Observe that $V = \beta XY + \gamma S_1 S_2$, and $V' = \beta' XY + \gamma' S_1 S_2$.

6. C computes locally the matrix

$$(\gamma'\beta - \gamma\beta')^{-1}(\gamma' V - \gamma V')$$

which happens to equal XY (as can be easily verified – we leave the details to the reader).

7. C computes $M_1 M_2$ from XY by computing

$$P_1^{-1} XY P_3 \;=\; P_1^{-1}(P_1 M_1 P_2^{-1})(P_2 M_2 P_3^{-1}) P_3 \;=\; M_1 M_2.$$

This completes the algorithm.

2.3 Non-square Matrices

We now turn our attention to the case when M_1 and M_2 are not square, i.e., when M_1 is $l \times m$ and M_2 is $m \times n$ and hence $M_1 M_2$ is $l \times n$. Essentially the same method as above works in that case, except that we have to carefully choose the sizes of the P_i and S_i matrices. For the S_i this is straightforward: S_1 must be $l \times m$ and S_2 must be $m \times n$, because each of them is added to matrices having such dimensions. But for the P_i we have a potential source of conflicting requirements: (i) A P_i must be a square matrix (because we need to use its inverse – non-square matrices have no inverse), (ii) the size of a P_i must be compatible with the number of rows of the matrices that it (or its inverse) is left-multiplying, and (iii) the size of a P_i must be compatible with the number of columns of the matrices that it (or its inverse) is right-multiplying. For example, as P_2 is used for left-multiplying M_2, and M_2 has m rows, there is a requirement that P_2 should be $m \times m$. Luckily, the requirement stemming from the fact that P_2^{-1} right-multiplies M_1 is compatible with the previous one, because M_1 has m columns. This is not an accident, and it is easy to verify that there are no conflicting requirements on the size of any of the P_i matrices that are used in the algorithm, $1 \leq i \leq 3$.

2.4 Hiding the Matrices' Dimensions

We briefly sketch how to hide the dimensions of the matrices to be multiplied. Let M_1 be an $a \times b$ matrix and M_2 be a $b \times c$ matrix. The problem of multiplying these matrices is replaced by one (or a small number) of multiplications of matrices whose dimensions a', b', c' are different from a, b, c (the new matrices are handled by using the methods already developed in the previous subsections). Hiding the dimensions can be done by either *enlarging* or *shrinking* one (or a combination of) the three relevant dimensions: We say that we have "enlarged" a if $a' > a$, that we have "shrunk" a if $a' < a$ (similarly for b' and c'). Although for convenience we shall explain how to enlarge/shrink a separately from how to enlarge/shrink b and c, it should be understood that these operations can be done in many possible combinations (we give some examples below).

2.4.1 Enlarging the dimensions

Enlarging a (so that it becomes $a' > a$) is done by appending $a' - a$ additional rows, having random entries, to the first matrix. Of course this causes the matrix product to have $a' - a$ additional rows, but these can be ignored.

Similarly, enlarging c (so that it becomes $c' > c$) is done by appending $c' - c$ additional columns, having random entries, to the second matrix. Of course this causes the matrix product to have

$c' - c$ additional columns, but these can be ignored.

On the other hand, enlarging b involves changes to both matrices, by appending $b' - b$ extra columns to the first matrix and $b' - b$ extra rows to the second matrix. Furthermore, these additional rows and columns cannot have completely random entries because they would then interact to corrupt the output: The output matrix has same dimensions after enlarging b as before — we need to make sure the output matrix is not changed by the enlargement of b. This is achieved as follows: Number the $b' - b$ extra columns $1, 2, \cdots, b' - b$, and similarly number the extra rows $1, 2, \cdots, b' - b$. Choose the entries of the odd-numbered extra columns (respectively, rows) to be random (respectively, zero), and choose the entries of the even-numbered extra columns (respectively, rows) to be zero (respectively, random). Verify that enlarging b in this way causes no change in the matrix product.

Of course the above three operations can be done in conjunction with each other: We would then first apply the enlargement of b, then the enlargements of a and c.

2.4.2 Shrinking the dimensions

Shrinking a is done by partitioning the first matrix M_1 into two matrices: One M_1' consisting of the first $a - a'$ rows, another M_1'' consisting of the last a' rows. The second matrix stays the same, but to get the $a \times c$ matrix we seek we now have to perform both $M_1' M_2$ and $M_1'' M_2$.

Similarly, shrinking c is done by partitioning the second matrix M_2 into two matrices: One M_2' consisting of the first $c - c'$ columns, another M_2'' consisting of the last c' columns. The first matrix stays the same, but to get the $a \times c$ matrix we seek we now have to perform both $M_1 M_2'$ and $M_1 M_2''$.

Shrinking b is done by partitioning both matrices into two matrices. The first matrix M_1 is partitioned into an M_1' consisting of the first $b - b'$ columns, another M_1'' consisting of the last b' columns. The second matrix M_2 is partitioned into an M_2' consisting of the first $b - b'$ rows, another M_2'' consisting of the last b' rows. The $a \times c$ matrix we seek is then $M_1' M_2' + M_1'' M_2''$.

Doing all of the above three shrinking operations simultaneously results in a partition of each of M_1 and M_2 into four matrices. If we denote by $M_1([i : j], [k : l])$ the submatrix of M_1 whose rows are in the interval $[i, j]$ and whose columns are in the interval $[k, l]$, then computing $M_1 M_2$ requires the following four computations:

$M_1([1 : a - a'], [1 : b - b'])M_2([1 : b - b'], [1 : c - c']) \ +$

$$M_1([1 : a - a'], [b - b' + 1 : b])M_2([b - b' + 1 : b], [1 : c - c']),$$

$M_1([1 : a - a'], [1 : b - b'])M_2([1 : b - b'], [c - c' + 1 : c]) \ +$

$$M_1([1:a-a'],[b-b'+1:b])M_2([b-b'+1:b],[c-c'+1:c]),$$
$$M_1([a-a'+1:a],[1:b-b'])M_2([1:b-b'],[1:c-c']) + M_1([a-a'+1:a],[b-b'+1:b])M_2([b-b'+1:b],[1:c-c']),$$
$$M_1([a-a'+1:a],[1:b-b'])M_2([1:b-b'],[c-c'+1:c]) + M_1([a-a'+1:a],[b-b'+1:b])M_2([b-b'+1:b],[c-c'+1:c]).$$

3 Matrix Inversion

Assume that C wants to outsource the inversion of the $n \times n$ matrix M. The scheme we describe next uses secure matrix multiplication as a subroutine. It satisfies the requirement that all local processing by C takes time proportional to the size of the input, in this case $O(n^2)$ time. We first give, in the next subsection, a scheme that does not concern itself with hiding n.

3.1 Inversion Scheme

1. C selects a random $n \times n$ matrix S. The probability that S is non-invertible is small, but if that is the case then Step 4 below will send us back to Step 1 and we will have to start over with another random matrix S. This need only be repeated $O(1)$ times before S is invertible (in practice, once is usually enough).

2. C outsources the computation of
$$\hat{M} = MS \tag{11}$$
using secure matrix multiplication. As before, we use A to denote the agent used. Of course after this step A knows neither M, nor S, nor $\hat{M}$.

3. C generates matrices P_1, P_2, P_3, P_4, P_5 using the same method as for the P_1 matrix in Steps 1 and 2 of the preliminary solution to matrix multiplication. That is, $P_1(i,j) = a_i\delta_{\pi_1(i),j}$, $P_2(i,j) = b_i\delta_{\pi_2(i),j}$, $P_3(i,j) = c_i\delta_{\pi_3(i),j}$, $P_4(i,j) = d_i\delta_{\pi_4(i),j}$, and $P_5(i,j) = e_i\delta_{\pi_5(i),j}$ where π_1, π_2, π_3, π_4, π_5 are random permutations, and where the a_i, b_i, c_i, d_i, e_i are random numbers. Then C computes locally, in $O(n^2)$ time, the matrices
$$Q = P_1\hat{M}P_2^{-1} = P_1MSP_2^{-1} \tag{12}$$
$$R = P_3SP_4^{-1}. \tag{13}$$

4. C sends Q to agent A, who tries to compute Q^{-1} and, if it succeeds, sends Q^{-1} back to C. If it does not succeed then Q is not invertible, and hence at least one of S or M (possibly

both) is non-invertible. When A detects that Q is non-invertible then it lets C know, and C then does the following:

(a) C tests whether S is invertible by first computing $\hat{S} = S_1 S S_2$ where S_1 and S_2 are matrices known by C to be invertible, and then sending $\hat{S}$ to A for the purpose of inverting it.

Note: C is only interested in whether $\hat{S}$ is invertible or not, not in its actual inverse; in fact C will discard S whether $\hat{S}$ is invertible or not. The fact that C will discard S makes the choice of S_1 and S_2 less crucial than otherwise. Hence S_1 and S_2 can be generated so they belong to a class of matrices known to be invertible, such as the P_i we have been using (in such a case $\hat{S}$ can be computed by C locally, without outsourcing); there are many other classes of matrices known to be invertible (cf. [8, 9]). It is unwise to let S_1 and S_2 be the identity matrices, because by knowing S the agent A might learn how we are generating these random matrices.

(b) If A can invert $\hat{S}$ then C knows that S is invertible, hence that M is not invertible. If A informs C that $\hat{S}$ is not invertible, then C knows that S is not invertible. In that case C goes back to Step 1, i.e., chooses another S, etc. The number of time C has to go back to Step 1 in this way is small (zero in practice) because of the high probability that a randomly chosen S matrix is invertible.

Observe that $Q^{-1} = P_2 S^{-1} M^{-1} P_1^{-1}$.

5. C computes locally, in $O(n^2)$ time, the matrix

$$T = P_4 P_2^{-1} Q^{-1} P_1 P_5^{-1}.$$

It is easily verified that T is equal to $P_4 S^{-1} M^{-1} P_5^{-1}$.

6. C outsources to agent A the computation of

$$Z = RT \tag{14}$$

using secure matrix multiplication. Of course the random permutations and numbers used within this secure matrix multiplication subroutine must be independently generated from those of the above Step 3 (using those of Step 3 would compromise security).

Observe that

$$Z = P_3 S P_4^{-1} P_4 S^{-1} M^{-1} P_5^{-1} = P_3 M^{-1} P_5^{-1}.$$

7. C computes locally in $O(n^2)$ time $P_3^{-1}ZP_5$, which equals M^{-1}.

The security of the above follows from

1. the fact that the calculations of $\hat{M}$ and Z are done using secure matrix multiplication, which reveals neither the operands nor the results to agent A, and

2. the judicious use of the matrices $P_1, \cdots, P_5$ for "isolating" from each other the three separate computations that we outsource to A; such isolation is a good design principle whenever repeated usage is made of the same agent, to make it difficult for that agent to correlate the various subproblems it is solving (in this case three). Of course less care needs to be taken if one is using more than one external agent (more on this later).

3.2 Hiding the Matrix Dimension

Hiding n is achieved by (i) using the dimension-hiding version of matrix multiplication in the scheme of the previous section, and (ii) in Step 4, performing the inversion of Q by inverting a small number of $n' \times n'$ matrices where n' differs from n.

If we wish to hide the dimension of Q in Step 4 by enlarging it (i.e., $n' > n$), then we need only modify Step 4 so that it inverts one $n' \times n'$ matrix Q' defined as follows, where O' (respectively, O'') is an $n \times (n'-n)$ (respectively, $(n'-n) \times n$) matrix all of whose entries are zero, and S' is an $(n'-n) \times (n'-n)$ random invertible matrix:

$$Q'([1:n],[1:n]) = Q,$$

$$Q'([1:n],[n+1:n']) = O',$$

$$Q'([n+1:n'],[1:n]) = O'',$$

$$Q'([n+1:n'],[n+1:n']) = S'.$$

Of course the inversion of Q' is *not* performed by sending it directly to the agent A as the zeroes in it would reveal n. Rather, the inversion of Q is done by using the scheme of the previous subsection (which does not worry about hiding dimensions — this is acceptable because the dimension of Q' is different from the n that we seek to hide).

The case of shrinking dimension is more subtle, and relies on the following fact [1]: If $X = Q([1:m],[1:m])$ is invertible ($m < n$), $Y = Q([m+1:n],[m+1:n])$, $V = Q([1:m],[m+1:n])$, $W = Q([m+1:n],[1:m])$, and $D = Y - WX^{-1}V$ is invertible, then

$$Q^{-1}([1:m],[1:m]) = X^{-1} + X^{-1}VD^{-1}WX^{-1},$$

$$Q^{-1}([1:m],[m+1:n]) = -X^{-1}VD^{-1},$$

$$Q^{-1}([m+1:n],[1:m]) = -D^{-1}WX^{-1},$$

$$Q^{-1}([m+1:n],[m+1:n]) = D^{-1}.$$

The above suggests that the modified Step 4 would partition Q into four matrices X, Y, V, W, then use the secure matrix multiplication scheme of the previous section (possibly with dimension-hiding) and the inversion scheme of the previous subsection (possibly with dimension-enlargement) to compute the four pieces of Q^{-1} described in the above equations.

4 Linear System of Equations

One of the most common uses of matrix inversion is in the solution of a system of linear equations $Mx = b$ where M is a known square $n \times n$ matrix, b is a known column vector of size n, and x is a column vector of n unknowns. However, a more numerically stable method of solving such a system is Gaussian Elimination [8], which takes M and b as input and produces x as output if M is nonsingular (otherwise it returns a message that M is singular). Therefore we need to consider the situation where C needs to outsource the solution of the linear system of equations $Mx = b$, that is, obtain x without having to reveal to A either M or b.

The scheme we describe below satisfies the requirement that all local processing by C take time proportional to the size of the input, in this case $O(n^2)$ time.

4.1 Outsourced Linear System Solution

1. C selects a random column vector V of size n and a random nonsingular matrix S of size $n \times n$.

2. C generates matrix P using the same method as for the P_1 matrix in Steps 1 and 2 of the preliminary solution to matrix multiplication. That is, $P(i,j) = a_i \delta_{\pi(i),j}$, where π is a random permutation, and where the a_i are random numbers.

3. C computes the following

$$\hat{M} = SMP^{-1}, \tag{15}$$

$$\hat{b} = SMP^{-1}V + Sb, \tag{16}$$

where the matrix multiplication involving S is securely outsourced, and the other operations are done locally (they take $O(n^2)$ time).

4. C outsources to agent A the solution of the linear system $\hat{M}\hat{x} = \hat{b}$. That is, C sends to A both $\hat{M}$ and $\hat{b}$. If $\hat{M}$ is singular then C gets a message from A saying so, and C can conclude that M itself is singular. Otherwise C gets back from A the column vector $\hat{x}$ where

$$\hat{x} = \hat{M}^{-1}\hat{b}. \tag{17}$$

5. C computes locally

$$P^{-1}\hat{x} - P^{-1}V \tag{18}$$

which is the answer x, because

$$\begin{aligned}
M(P^{-1}\hat{x} - P^{-1}V) &= MP^{-1}\hat{x} - MP^{-1}V \\
&= MP^{-1}(\hat{M}^{-1}b') - MP^{-1}V \\
&= MP^{-1}(PM^{-1}S^{-1})(SMP^{-1}V + Sb) \\
&\quad -MP^{-1}V \\
&= MP^{-1}V + b - MP^{-1}V \\
&= b.
\end{aligned}$$

This completes the algorithm.

The security of the above follows from the fact that M is hidden through permutation and scaling by right-multiplication by P^{-1}, and left-multiplication by the random matrix S. Also, the actual solution is hidden with the addition of an additive random component $(SMP^{-1}V)$ to the right hand side of the system of equations.

4.2 Hiding the Dimension

We only describe how to hide n by embedding the problem $Mx = b$ into a larger problem $M'x' = b'$ of size $n' > n$; shrinking the dimension can be done by using something akin to the equation at the end of the section on matrix inversion.

Notation 2 *In what follows, if X is an $r \times c$ matrix and Y is an $r' \times c$ matrix ($r < r'$), the notation "$Y = X(*, [1 : c])$" means that Y consists of as many copies of X as needed to fill the r' rows of Y; the last copy could be partial, if r does not divide r'.*

For example, if in the above $r' = 2.5r$ then the notation would mean that $Y([1 : r], [1 : c]) = Y([r + 1 : 2r], [1 : c]) = X$, and $Y([2r + 1 : 2.5r], [1 : c]) = X([1 : 0.5r], [1 : c])$.

The larger problem $M'x' = b'$ of size $n' > n$ is defined as follows. The matrix M' and vector b' are defined as follows, where O' (respectively, O'') is an $n \times (n'-n)$ (respectively, $(n'-n) \times n$) matrix all of whose entries are zero, S' is an $(n'-n) \times (n'-n)$ random invertible matrix, and y is a random vector of length $n'-n$:

$$M'([1:n],[1:n]) = M,$$

$$M'([1:n],[n+1:n']) = O',$$

$$M'([n+1:n'],[1:n]) = O'',$$

$$M'([n+1:n'],[n+1:n']) = S',$$

$$b'([1:n]) = b,$$

$$b'([n+1:n']) = S'y.$$

Then the solution x' to the system $M'x' = b'$ is $x'([1:n]) = x$ and $x'([n+1,n']) = y$. Note that he zero entries of O' and O'' do not betray n because Step 3 of the scheme of the previous subsection hides these zeroes when it computes $\hat{M} = SMP^{-1}$. We can even avoid having O' and O'' be zeroes if, in the above, we make

1. O' a random matrix (rather than a matrix of zeroes),

2. $O'' = M(*,[1:n])$,

3. $S' = O'(*,[n+1:n'])$,

4. $b' = (b + O'y)(*)$.

If the random choices made for y and O' result in a noninvertible M', then we repeat until we get an invertible M'. Assuming M' is invertible, the solution x' to the system $M'x' = b'$ is still $x'([1:n]) = x$ and $x'([n+1,n']) = y$, because

$$Mx + O'y = b'([1:n]) = b + O'y$$

and hence $Mx = b$.

5 Convolution

Assume that C needs to outsource the computation of the convolution of two vectors M_1 and M_2 of size n each, indexed from 0 to $n-1$. The convolution M of M_1 and M_2 is a new vector of size $2n$, denoted $M = M_1 \otimes M_2$, such that

$$M(i) = \sum_{k=0}^{i} M_1(k) M_2(i-k). \tag{19}$$

Convolution takes $O(n^2)$ time if done naively, $O(n \log n)$ time if the Fast Fourier Transform (FFT) is used [1].

The scheme we describe below satisfies the requirement that all local processing by C take $O(n)$ time.

5.1 Convolution Scheme

1. C selects two random vectors S_1 and S_2, of size n each (that is, vectors whose entries are random). C also generates five positive random numbers $\alpha, \beta, \gamma, \beta', \gamma'$ such that

$$(\beta + \alpha\gamma)(\beta' + \alpha\gamma')(\gamma'\beta - \gamma\beta') \neq 0.$$

   If the above is violated then we discard the five random numbers chosen and we repeat the random experiment of choosing a new set of numbers; observe, however, that there is zero probability that a random choice results in a violation of the above condition, hence the random choice need not be repeated more than $O(1)$ times (in practice, once is usually enough).

2. C computes locally the six vectors $\alpha M_1 + S_1$, $\alpha M_2 + S_2$, $\beta M_1 - \gamma S_1$, $\beta M_2 - \gamma S_2$, $\beta' M_1 - \gamma' S_1$, $\beta' M_2 - \gamma' S_2$. Then C sends these six vectors, in the above order, to agent A.

3. Agent A uses the six vectors received from C to compute three convolutions, one for each pair of vectors received:

$$W = (\alpha M_1 + S_1) \otimes (\alpha M_2 + S_2) \tag{20}$$

$$U = (\beta M_1 - \gamma S_1) \otimes (\beta M_2 - \gamma S_2) \tag{21}$$

$$U' = (\beta' M_1 - \gamma' S_1) \otimes (\beta' M_2 - \gamma' S_2) \tag{22}$$

   A then sends W, U, U' to C.

4. C computes locally the vectors V and V' where

$$V = (\beta + \alpha\gamma)^{-1}(\alpha U + \beta\gamma W) \tag{23}$$

$$V' = (\beta' + \alpha\gamma')^{-1}(\alpha U' + \beta'\gamma' W). \tag{24}$$

   Observe that $V = \alpha\beta M_1 \otimes M_2 + \gamma S_1 \otimes S_2$, and $V' = \alpha\beta' M_1 \otimes M_2 + \gamma' S_1 \otimes S_2$.

5. C computes locally the vector

$$\alpha^{-1}(\gamma'\beta - \gamma\beta')^{-1}(\gamma' V - \gamma V'),$$

   which happens to equal $M_1 \otimes M_2$ (as is easily verified). This completes the algorithm.

The security of the above scheme is based on the fact that the six vectors received by A do not enable it to discover M_1 or M_2, as A does not know the numbers $\alpha, \beta, \gamma, \beta', \gamma'$ and the vectors S_1, S_2.

5.2 Hiding the Dimension

Hiding the dimension by expanding the problem size is straightfowrad by "padding" the two input vectors with zeroes (the details are easy and are omitted). The zeroes do not betray the value of n because Step 2 hides these zeroes by adding random numbers to them.

Hiding the dimension by shrinking the problem size is done in two steps: (i) Replacing the convolution of size n by three convolutions of size $n/2$ each, and then (ii) recursively hiding (by shrinking or by expanding) the sizes of these three convolutions. If suffices for the depth of the recursion in (ii) to be $O(1)$. That (i) is possible is seen as follows. For an n-vector M, let $M^{(even)}$ (respectively, $M^{(odd)}$ denote the $(n/2)$-vector consisting of the even (respectively, odd) numbered entries of M. It is easy to verify that

$$(M_1 \otimes M_2)^{(odd)} = M_1^{(even)} \otimes M_2^{(odd)} + M_1^{(odd)} \otimes M_2^{(even)},$$

$$(M_1 \otimes M_2)^{(even)} = M_1^{(even)} \otimes M_2^{(even)} + Shift(M_1^{(odd)} \otimes M_2^{(odd)}),$$

where $Shift(x)$ shifts the vector x by one position. This implies that the following three convolutions, involving vectors of size $n/2$ each, are enough to obtain $M_1 \otimes M_2$:

1. $(M_1^{(even)} + M_1^{(odd)}) \otimes (M_2^{(even)} + M_2^{(odd)})$
2. $(M_1^{(even)} - M_1^{(odd)}) \otimes (M_2^{(even)} - M_2^{(odd)})$

3. $M_1^{(odd)} \otimes M_2^{(odd)}$

Adding and subtracting the results of the above convolutions (1) and (2) enables us to obtain $M_1^{(even)} \otimes M_2^{(odd)} + M_1^{(odd)} \otimes M_2^{(even)}$ and $M_1^{(even)} \otimes M_2^{(even)} + M_1^{(odd)} \otimes M_2^{(odd)}$: The former is recognized as $(M_1 \otimes M_2)^{(odd)}$, and the latter allows us to obtain (in conjunction with the result of convolution (3)) $(M_1 \otimes M_2)^{(even)}$.

6 Sorting

Asssume that C needs to outsource the sorting of a sequence of numbers $E = \{e_1, \cdots, e_n\}$ with the e_i taken from a set equipped with a total ordering relationship (without loss of generality let us assume that the e_i are real: $e_i \in R$, $i = 1, \cdots, n$). E is not to be revealed to the outsourcing agent. This can be done as follows.

C selects a strictly increasing function $f : E \mapsto R$, such as

$$f(e_i) = \alpha + \beta(e_i + \gamma)^2 \tag{25}$$

where α, β, γ are known to C but not to A. In fact, even the nature of f could be hidden from A if C selects the function f from a large catalog of functions; the rest of this section assumes the above quadratic form for f. Observe that for the above f to be stricly increasing, γ must be chosen such $e_i + \gamma \geq 0$ for all i.

The scheme we describe below satisfies the requirement that all local processing by C take $O(n)$ time.

1. C chooses α, β, and γ locally, thus defining the function f (as explained above).

2. C chooses locally a random sorted sequence $\Lambda = (\lambda_1, \cdots, \lambda_l)$ of l numbers. This is done by randomely "walking" on the real line from MIN to MAX where MIN is smaller than the smallest number being sorted and MAX is larger than the largest number being sorted. This random "walking" is implemented as follows. Let $\Delta = (MAX - MIN)/n$. C generates the sorted sequence Λ as follows:

   (a) Randomely generate λ_1 from a uniform distribution in $[MIN, MIN + 2\Delta]$.

   (b) Randomely generate λ_2 from a uniform distribution $[\lambda_1, \lambda_1 + 2\Delta]$.

   (c) Continue in the same way until you go past MAX, at which time you stop. The total number of elements generated is l for some integer l.

Observe that Λ is sorted by construction, that each random increment has expected value Δ, and that the expected value of l is $(MAX - MIN)/\Delta = n$.

3. C produces locally the sequences

$$E' = f(E) \tag{26}$$

$$\Lambda' = f(\Lambda), \tag{27}$$

where $f(E)$ is the sequence obtained from E by replacing every element e_i of E by $f(e_i)$.

4. C concatenates a copy of the sequence Λ' to E', obtaining $E' \cup \Lambda'$. Then C generates a randomly permuted version (call it W) of $E' \cup \Lambda'$.

5. C sends W to agent A, who sorts it and sends back a sorted version of W, call it W'.

6. C receives W' and removes from it the sequence Λ'. C can do this in $O(n)$ time because W' and the saved copy of Λ' are already sorted. This produces a sequence $\hat{E}$, which is a sorted version of $E' = f(E)$.

7. C computes $f^{-1}(\hat{E})$, which is equal to a sorted version of E. This completes the algorithm.

The above scheme reveals n because the number of items we send to A for sorting has expected value $2n$. To change this from $2n$ to $m + n$ where m is unrelated to n, we would have to modify Step 2 so that $\Delta = (MAX - MIN)/m$ where m differs from n (hence the expected value of l in Step 2 becomes m). This hides problem size by expanding it. Hiding it by shrinking is done by partitioning the problem into a constant (small) number of problems, each of which is then recursivey sorted, i.e., by outsourcing with size-hiding (using shrinking or expansion). The sorted pieces are then merged locally by C, in linear time.

7 Experimental Results and Practical Observations

The purpose of the experimental work is not only to have "proof of concept" software, but also to shed some light on the numerical properties of the schemes proposed, namely, the difference between the answer we obtain and the answer that would have been obtained if all of the computations had been done locally (i.e., without using our outsourcing schemes). If computers had infinite precision (or if we use sophisticated software that simulates such precision) then the difference is, of course, zero. We report a series of experimental results for the secure outsourcing algorithms presented in the previous sections. The algorithms have been implemented in `ANSI C++`, compiled with the `GNU` `g++` compiler and executed in `double` precision on a `SUN SparcStation 20` running the `Solaris 5.4` operating system. In the results that are reported the following metrics are used.

Vector Metrics For a vector V of size n we use the following norms

- $||V||_1 = \sum_{i=1}^{n} |V(i)|$
- $||V||_2 = \sqrt{\sum_{i=1}^{n} |V(i)|^2}$
- $||V||_\infty = \max_{1 \leq i \leq n} |V(i)|$

For two vectors of size n, V and $\hat{V}$ we use the following errors

- *Absolute Error*: $\epsilon_{abs} = ||\hat{V} - V||$
- *Relative Error*: $\epsilon_{rel} = \frac{||\hat{V} - V||}{||V||}$

where the norms involved can be any of the three norms defined above.

Matrix Metrics For a matrix M of size $m \times n$ we use the following norms

- $||M||_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |M(i,j)|$
- $||M||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |M(i,j)|^2}$, where F stands for the Frobenius norm.
- $||M||_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |M(i,j)|$

For two matrices of size $m \times n$, M and $\hat{M}$ we use the following errors

- *Absolute Error*: $\epsilon_{abs} = ||\hat{M} - M||$
- *Relative Error*: $\epsilon_{rel} = \frac{||\hat{M} - M||}{||M||}$
- *Maximum Absolute Error*: $\epsilon_{max} = \max_{1 \leq (i,j) \leq (m,n)} |\hat{M}(i,j) - M(i,j)|$

where the norms involved can be any of the three norms defined above.

General Metrics For a sequence of length K of pairs of matrices $\{(\hat{M}_i, M_i), i = 1, \cdots, K\}$, or vectors $\{(\hat{V}_i, V_i), i = 1, \cdots, K\}$, we define the *average* absolute error as

$$\frac{1}{K} \sum_{i=1}^{K} \epsilon_{abs}(\hat{M}_i, M_i), \quad or, \quad \frac{1}{K} \sum_{i=1}^{K} \epsilon_{abs}(\hat{V}_i, V_i)$$

We define the *average* relative error similarly as

$$\frac{1}{K}\sum_{i=1}^{K}\epsilon_{rel}(\hat{M}_i, M_i), \quad or, \quad \frac{1}{K}\sum_{i=1}^{K}\epsilon_{rel}(\hat{V}_i, V_i)$$

The average ϵ_{max} is defined similarly. The *root-mean-square* error (RMS) is defined, for any of the three norms, as

$$\sqrt{\frac{1}{K}\sum_{i=1}^{K}(||\hat{M}_i - M_i||)^2}, \quad or, \quad \sqrt{\frac{1}{K}\sum_{i=1}^{K}(||\hat{V}_i - V_i||)^2},$$

The experimental results reported are based on a sequence of trial inputs that were randomly generated. The averages are taken over these sequences of inputs. The error results that are reported are based on the value obtained through secure outsourcing and the value that is computed by the normal local implementation of the relevant algorithm. The number of trials for convolution was $1,000$, while for the other three algorithms the number of trials was 100 because of the enormous size of the computations. Indicative numbers are reported for three different sizes of the input. For convolution, we give the error on vectors of size 10, 100, and $1,000$. For matrix multiplication we report results for products of square matrices 10×10, 50×50, and 100×100. The results for the solution of linear systems of equations are for 10, 50, and 100 unknowns. Finally the results for matrix inversion are for matrix sizes of 10×10, 50×50, and 100×100. The actual entries of the matrices and vectors in the above experiments were typically two to three digits long, i.e., between 10 and $1,000$, generated randomly.

The RMS for all the algorithms is reported in all three norms while the relative, absolute and maximum are reported only for the infinity norm. Observe that $\epsilon_{abs} \equiv \epsilon_{max}$ for vectors using the infinity norm, so that for the solution of linear systems and the convolution we only report the ϵ_{abs} in norm infinity.

Speaking in general terms, the absolute error in norm infinity is an indication of the number of decimal digits that are correct. For example, an error of 10^{-p} would imply that at least p decimal digits are correct.

We observe that in all four algorithms, the error is very small but tends to increase as we scale the size of the input. This is expected as the accumulation of round-off errors becomes larger for larger inputs. Let us mention that our implementation has adopted no special techniques for error control or higher accuracy. We have implemented the outsourcing algorithms described in a straightforward manner, using LU decomposition with implicit partial pivoting for matrix inversion and linear system solution and plain computer algebra for convolution and matrix multiplication.

In this sense, the results reported here should be considered as an upper bound on the error – smaller errors would result if we had used sophisticated numerical methods for error control.

100 trials	RMS, $\|\| \bullet \|\|_1$	RMS, $\|\| \bullet \|\|_F$	RMS, $\|\| \bullet \|\|_\infty$	ϵ_{abs}, $\|\| \bullet \|\|_\infty$	ϵ_{rel}, $\|\| \bullet \|\|_\infty$	ϵ_{max}
size: 10×10	$2.115e-10$	$9.947e-11$	$1.667e-10$	$2.268e-10$	$2.569e-16$	$6.446e-11$
size: 50×50	$1.342e-08$	$1.997e-09$	$4.231e-09$	$5.322e-08$	$2.643e-15$	$4.381e-09$
size: 100×100	$1.853e-07$	$1.169e-08$	$1.982e-08$	$3.079e-06$	$4.025e-14$	$1.489e-07$

Table 1: Error metrics for secure matrix multiplication

$1,000$ trials	RMS, $\|\| \bullet \|\|_1$	RMS, $\|\| \bullet \|\|_2$	RMS, $\|\| \bullet \|\|_\infty$	ϵ_{abs}, $\|\| \bullet \|\|_\infty$	ϵ_{rel}, $\|\| \bullet \|\|_\infty$
size: 10	$9.797e-08$	$2.476e-08$	$1.726e-08$	$2.055e-08$	$2.622e-16$
size: 100	$1.088e-05$	$7.853e-07$	$3.078e-07$	$5.680e-07$	$8.178e-16$
size: 1000	$1.652e-03$	$3.465e-05$	$8.629e-06$	$2.276e-05$	$3.491e-15$

Table 2: Error metrics for secure convolution

100 trials	RMS, $\|\| \bullet \|\|_1$	RMS, $\|\| \bullet \|\|_2$	RMS, $\|\| \bullet \|\|_\infty$	ϵ_{abs}, $\|\| \bullet \|\|_\infty$	ϵ_{rel}, $\|\| \bullet \|\|_\infty$
size: 10	$4.512e-10$	$1.627e-10$	$8.273e-11$	$2.053e-11$	$3.248e-12$
size: 50	$3.445e-07$	$5.812e-08$	$2.007e-08$	$9.463e-09$	$2.100e-09$
size: 100	$1.599e-04$	$1.979e-05$	$5.008e-06$	$1.742e-06$	$2.097e-07$

Table 3: Error metrics for secure solution of linear systems

100 trials	RMS, $\|\| \bullet \|\|_1$	RMS, $\|\| \bullet \|\|_F$	RMS, $\|\| \bullet \|\|_\infty$	ϵ_{abs}, $\|\| \bullet \|\|_\infty$	ϵ_{rel}, $\|\| \bullet \|\|_\infty$	ϵ_{max}
size: 10×10	$2.725e-13$	$7.504e-13$	$1.172e-12$	$2.652e-13$	$1.048e-13$	$7.292e-14$
size: 50×50	$3.810e-08$	$1.821e-08$	$3.142e-08$	$7.927e-08$	$9.248e-09$	$1.028e-08$
size: 100×100	$1.697e-06$	$5.857e-07$	$1.467e-06$	$1.731e-05$	$2.692e-06$	$1.205e-06$

Table 4: Error metrics for secure matrix inversion

8 Further Remarks

All of the schemes described in this paper assume the use of a single external agent. If more than one agent is available, then by randomly choosing from the available pool of agents we can afford to do less data hiding. This was pointed out earlier in the context of one particular scheme, but all of our schemes could be simplified if they were allowed to make use of more than one agent. (Intuition makes one expect a tradeoff between the number of available agents and the amount of hiding needed.) Future research in this area may well encounter problems for which secure outsourcing can be achieved only by using more than one external agent.

A multi-agent environment raises many interesting questions, including:

- Whether it is reasonable to assume that mutiple external agents will not conspire with each other against the customer, by sharing with each other the data that the customer sends them.
- If external agents are conspiring against the customer, how they can overcome the problem of "matching" the relevant subcomputations outsourced by the customer to each of them (from among the potentially huge number of computations outsourced to them by the customer). The customer can make this task difficult by
  - deliberately interleaving the temporal ordering of the jobs outsourced to achieve better security, and
  - deliberately outsourcing "fake" computations.

  The above two obfuscation techniques make sense even in a single-agent environment.
- How one goes about proving that the secure outsourcing of a particular problem inherently requires at least k external (non-conspiring) agents, $k > 1$.

We also note that the scheme proposed in this paper solves an interesting problem related to the distributed scheduling system described by Chapin and Spafford in [4]. The work described there provided an architecture to distribute large computations without disclosing information about the machines doing the computation, and without sacrificing control of those machines. The drawback to that scheme was that the users did not have a means of hiding their data and computation from the machine owners. The method described here addresses that concern, and enables outsourcing to take place in an environment that is not completely defined.

Acknowledgements

The authors gratefuly acknowledge support from the COAST Project at Purdue and its corporate sponsors. The support of the Purdue Special Initiative Fellowship and the support and advice of Dr. Elias Houstis are gratefully acknowledged by the second author.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] M. Abadi, J. Feigenbaum, J. Killian. On Hiding Information from an Oracle. J. of Computer and System Sciences, **39**, pages 21–50, 1989.

[3] P. Beguin, J-J. Quisquater. Fast Server-Aided RSA Signatures Secure Against Active Attacks. CRYPTO 1995, pp. 57–69.

[4] S. J. Chapin and E. H. Spafford. Support for security in distributed systems: Using MESSIAHS. In *proceedings of the* National Computer Security Conference, pages 339–447, October 1994.

[5] B. Dole, S. Lodin, and E. H. Spafford. Misplaced trust: Kerberos 4 session keys. In *Proceedings of 4th* Symposium on Network and Distributed System Security, pages 60–71, IEEE Press, February 1997.

[6] D. E. Eastlake, S. D. Crocker, and J. I. Schiller. *RFC–1750 Randomness Recommendations for Security.* Network Working Group, December 1994.

[7] S. Garfinkel and E. H. Spafford. *Practical UNIX & Internet Security.* O'Reilly & Associates, second edition, 1996.

[8] G. H. Golub and C. F. Van Loan. newblock *Matrix Computations.* Johns Hopkins University Press, third edition, 1997.

[9] R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, 1985.

[10] S-J. Hwang, C-C. Chang, W-P. Yang. Some Active Attacks on Fast Server-Aided Secret Computation Protocols for Modular Exponentiation. *Cryptography: Policy and Algorithms,* LNCS 1029, 1996, pp. 215–228.

[11] S-I. Kawamura, A. Shimbo. Fast Server-Aided Secret Computation Protocols for Modular Exponentiation. *Proc. of* IEEE J. on Selected Areas in Communications, 11(5), 1993, pp. 778–784.

[12] D. E. Knuth. *The Art of Computer Programming, Volume 2.* Addison Wesley, second edition, 1981.

[13] C-S. Laih, S-M. Yen. Secure Addition Sequence and its Application on the Server-Aided Secret Computation Protocols. AUSCRYPT 1992, pp. 219–230.

[14] C-H. Lim, P. J. Lee. Security and Performance of Server-Aided RSA Computation Protocols. CRYPTO 1995, pp. 70–83.

[15] T. Matsumoto, K. Kato, H. Imai. Speeding Up Secret Computations with Insecure Auxiliary Devices. CRYPTO 1988, pp. 497–506.

[16] B. Pfitzmann, M. Waidner. Attacks on Protocols for Server-Aided RSA Computation. EUROCRYPT 1992, pp. 153–162.

[17] J-J. Quisquater, M. de Soete. Speeding Up Smart Card RSA Computations With Insecure Coprocessors. Smart Card 2000, North Holland, 1991, pp. 191–197.

[18] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. In Richard A. DeMillo, editor, *Foundations of Secure Computation*, pages 169–177. Academic Press, 1978.

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

[20] A. Shimbo, S. Kawamura. Factorization Attacks on Certain Server-Aided Computation Protocols for the RSA Secret Transformation. Electronic Letters, 26(17), 1990, pp. 1387–1388.

[21] D. R. Stinson. *Cryptography: Theory and Practice.* CRC Press, Boca Raton, FL, 1995.

[22] G. J. Simmons, editor. *Contemporary Cryptology: The science of Information Integrity.* IEEE Press, 1992.