SECURE OUTSOURCING OF SCIENTIFIC COMPUTATIONS

Mikhail J. Atallah, K.N. Pantazopoulos, John R. Rice and Eugene H. Spafford
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907, U.S.A.
email: {mja, jrr}@cs.purdue.edu

October 14, 1999

Abstract

We investigate the *outsourcing* of numerical and scientific computations using the following framework: A *customer* who needs computations done but lacks the computational resources (computing power, appropriate software, or programming expertise) to do these locally, would like to 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 external agent either the actual data or the actual answer to the computations. The general idea is for the customer to do some carefully designed local preprocessing (*disguising*) of the problem and/or data before sending it to the agent, and also some local postprocessing of the answer returned to extract the true answer. The disguise process should be as lightweight as possible, e.g., take time proportional to the size of the input and answer. The disguise preprocessing that the customer performs locally to "hide" the real computation can change the numerical properties of the computation so that numerical stability must be considered as well as security and computational performance. We present a framework for disguising scientific computations and discuss their costs, numerical properties, and levels of security. We show that no single disguise technique is suitable for a broad range of scientific computations but their is an array of disguise techniques available so that almost any scientific computation could be disguised at a reasonable cost and with very high levels of security. These disguise techniques can be embedded in a very high level, easy-to-use system (problem solving environment) that hides their complexity.

Contents

1 INTRODUCTION

1.1 Outsourcing and Disguise

Outsourcing is a general procedure employed in the business world when one entity, the *customer*, chooses to farm out (*outsource*) a certain task to an external entity, the *agent*. We believe that outsourcing will become the common way to do scientific computation [10, 11, 12, 23]. The reasons for the customer to outsource the task to the agent can be many, ranging from a lack of resources to perform the task locally to a deliberate choice made for financial or response time reasons. Here we consider the outsourcing of numerical and scientific computations, with the added twist that the problem data and the answers are to be hidden from the agent who is performing the computations on the customer's behalf. That is, 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 information.

The current outsourcing practice is to operate "in the clear", that is, by revealing both data and results to the agent performing 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 in a particular geographic area. The seismic data is so massive that doing matrix computations on such large data arrays is beyond the computational resources of even the major oil service companies, which routinely outsource these computations to supercomputing centers.

We consider many science and engineering computational problems and investigate various schemes for outsourcing to an outside agent a suitably disguised version of the computation in such a way that the customer's information is hidden from the agent, and yet the answers returned by the agent can be used to obtain easily the true answer. The local computations should be as minimal as possible and the disguise should not degrade the numerical stability of the computation. We note that there might be a class of computations where disguise is not possible, those that depend on the exact relationships among the data items. Examples that come to mind include (a) ordering a list of numbers, (b) typesetting a technical manuscript, or (c) visualizing a complex data set, (d) looking for a template in an image. However, example (a) can be disguised [3], and disguising (d) is actually a (quite nontrivial) contribution of this paper, so perhaps the others might be disguised also. To appreciate the difficulty of (d), consider the obvious choice for hiding the image, i.e., adding a random matrix to it: What does one do to the template so that the disguised version of the template occurs in the disguised version of the image? It looks like a chicken and egg problem: If we knew where the template occurs then we could add to it the corresponding portion of the images random matrix (so that the occurrence is preserved by the disguise), but of course we do

not know where it occurs – this is why we are outsourcing it in th first place.

1.2 Related Work in Cryptography

The techniques presented here differ from what is found in the cryptography literature concerning this kind of problem. Secure outsourcing in the sense of [1] follows an information-theoretic approach, leading to elegant negative results about the impossibility of securely outsourcing computationally intractable problems. In contrast, our methods are geared towards scientific computations that may be solvable in polynomial time, (e.g., solution of a linear system of equations) or where time complexity is undefined (e.g., the work to solve a partial differential equation is not related to the size of the text strings that define the problem). In addition, the cryptographic protocols literature contains much that is reminiscent of the outsourcing framework, 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, [30, 33]). The framework of the *privacy homomorphism* approach that has been proposed in the past [29] 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, the customer does not wish to do the computation locally, perhaps because of the lack of computing power or appropriate software or perhaps because of economics.

Our problem is also reminiscent of the *server-aided computation* work in cryptography, but there most papers deal with modular exponentiations and not with numerical computing [4, 17, 19, 22, 21, 23, 24, 25, 31].

Our problems and techniques afford us (as will be apparent below) the flexibility of using *one-time-pad* kinds of schemes for disguise. For example, when we disguise a number x by adding to it a random value r, then we do not re-use that same r to disguise another number y (we generate another random number for that purpose). If we hide a vector of such x's by adding to each a randomly generated r, then we have to be careful to use a suitable distribution for the r's (more on this later). The random numbers used for disguises are not shared with anyone: They are merely stored locally 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, in the following way. 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.

Note that the above 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 actually performing. This is the case even if the external agent has the source code, so long as the customer's seeds (of the generators for the randomness) are not known.

Throughout this paper when we use random numbers, random matrices, random permutations, random functions (e.g., polynomials, splines, etc., with random coefficients), etc.; it is assumed that each is generated independently of the others, and that quality random number generation is used (cf. [15, Chap. 23], [9, Chap. 12], 13, 20]).

The parameters, types, and seeds of these generators provide the keys to the disguises. We show how to use a single key to generate multiple keys which are "independent" and which simplify the mechanics of the disguise techniques. This key is analogous to the key in encryption but the techniques are different.

1.3 Other Differences Between Disguise and Encryption

The following simple example further illustrates the difference between encryption and disguise. Consider a string F of text characters that are each represented by an integer from 1 to 256 (i.e., these are byte string). Suppose that F_1 is an encryption of F with one of the usual encryption algorithms. Suppose that F_2 is a disguise of F that is created as follows: (1) Choose a seed (the disguise key) for a uniform random number generator and create a sequence G of random integers between 0 and 128, (2) Set $F_2 = F + G$. Assume now that F is a constant (the single value 69) string of length N and the agent wishes to discover the value of this constant. It is not possible to discover from F_1 the value 69 no matter how large N is. However, it is possible to discover 69 from F_2 if N is large enough. Since G is uniform, the mean of the values of G converge to 64 as N increases and thus, as N increases, the mean of F_2 converges to $133 = 64 + 69$ and the rate of convergence is order $1/\sqrt{N}$. Thus, when $1/\sqrt{N}$ is somewhat less than $1/2$, we know that the mean of F_2 is 133 and that the character is 69. An estimate of N is obtained by requiring that $128/\sqrt{N}$ be less than $1/2$ or N be more than about 60–70,000.

The point of this example is that the encryption cannot be broken in this case without knowing the encryption key — even if one knows the encryption method. However, the disguise can be broken without knowing the key provided the disguise method is known. Of course, it follows that one should not use a simplistic disguise and we provide disguise techniques for scientific computations with security comparable, one believes, to that of the most secure encryptions.

1.4 Four Simple Examples

The nature and breadth of the disguises possible are illustrated by the following:

1.4.1 Matrix multiplication

Consider the computation of the product of two $n \times n$ matrices M_1 and M_2. We use $\delta_{x,y}$ to denote the Kronecker delta function that equals 1 if $x = y$ and 0 if $x \neq y$. The disguise requires six steps:

1. Create (i) three random permutations π_1, π_2, and π_3 of the integers $\{1, 2, \ldots, n\}$, and (ii) three sets of non-zero random numbers $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, $\{\beta_1, \beta_2, \ldots, \beta_n\}$, and $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$.
2. Create 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 $P_3(i,j) = \gamma_i \delta_{\pi_3(i),j}$. These matrices are readily invertible, e.g., $P_1^{-1}(i,j) = (\alpha_j)^{-1} \delta_{\pi_1^{-1}(i),j}$.
3. Compute the matrix $X = P_1 M_1 P_2^{-1}$. We have $X(i,j) = (\alpha_i/\beta_j) M_1(\pi_1(i), \pi_2(j))$.
4. Compute $Y = P_2 M_2 P_3^{-1}$.
5. Send X and Y to the agent which computes the product $Z = XY = (P_1 M_1 P_2^{-1})(P_2 M_2 P_3^{-1})$ $= P_1 M_1 M_2 P_3^{-1}$ and sends Z back.
6. Compute locally, in $O(n^2)$ time, the matrix $P_1^{-1} Z P_3$, which equals $M_1 M_2$.

This disguise may be secure enough for many applications, as the agent 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 determine M_1 or M_2. This example is taken from [3] where the much more secure disguise of Section 3.1.1 is presented. Both of these disguises require $O(n^2)$ local computation, which is the minimum possible since the problem involve $O(n^2)$ data. The outsourced computations require $O(n^3)$ operations.

1.4.2 Quadrature

The objective is to estimate

$$\int_a^b f(x)dx$$

with accuracy *eps*. The disguise is as follows:

1. Choose $x_1 = a$, $x_7 = b$ and 5 ordered, random numbers x_i in $[a, b]$ and 7 values v_i with $\min |f(x)| \approx M_1 \leq M_2 \approx \max |f(x)|$. ($M_1$ and M_2 are only estimated roughly.)

2. Create the cubic spline $g(x)$ with knots x_i so that $g(x_i) = v_i$.

3. Integrate $g(x)$ exactly from a to b to obtain I_1.

4. Send $g(x)+f(x)$ and *eps* to the agent for numerical quadrature and receive the value I_2 back.

5. Compute $I_2 - I_1$ which is the answer.

All the computations made locally are simple, of fixed work, independent of *eps*, and depend weakly on $f(x)$. The random vectors and matrices of the previous example are replaced by a "random" smooth function. One has to determine 12 random numbers in order to break the disguise.

1.4.3 Edge detection

The objective is to determine the edges in a picture represented by an $n \times n$ array of pixel values $p(x, y)$ between 0 and 100,000 on the square $0 \leq x, y \leq 1$. This disguise is as follows:

1. Set $x_1, y_1 = 0$, $x_{10}, y_{10} = 1$, choose two sets of 8 ordered, random numbers with $0 < x_i, y_i < 1$, choose 100 random values $0 \leq v_{i,j} \leq 50,000$, and choose four pairs (a_i, b_i) of positive, random numbers with $a_1 = \min a_i$, $a_4 = \max a_i$, $b_1 = \min b_i$, $b_4 = \max b_i$.

2. Create the bi-cubic spline $s(x, y)$ so that $s(x_i, y_j) = v_{ij}$.

3. Determine the linear change of coordinates from (x, y) to (u, v) that maps the unit square into the rectangle with vertices (a_i, b_i).

4. Send $p(u(x, y), v(x, y)) + s(u(x, y), v(x, y))$ to the agent for edge detection and receive the image $e(u, v)$ back showing the edges.

5. Compute $e(x(u, v), y(u, v))$ to obtain the edges.

This disguise uses the fact that adding a smooth pixel function to the image and making a smooth change of coordinate does not add or delete any edges from the image. As in 1.4.1, the work of the disguise is proportional to the size of the data for the computing (plus a small constant amount). Here one must determine 124 random numbers in order to break the disguise.

1.4.4 Solution of a differential equation

The objective is to solve the two point boundary value problem

$$y'' + a_1(x)y' + a_2(x)y = f(x, y) \qquad y(a) = y_0, y(b) = y_1.$$

The disguise is as follows:

1. Choose a spline $g(x)$ as in 1.4.2 above.
2. Create the function
$$u(x) = g'' + a_1(x)g' + a_2(x)g.$$
3. Send the problem
$$y'' + a_1(x)y' + a_2(x)y = f(x,y) + u(x) \qquad y(a) = y_0 + u(a), y(b) = y_1 + u(b)$$
to the agent for solution and receive $z(x)$ back.
4. Compute $z(x) - g(x)$ which is the solution.

This disguise applies the problem's mathematical operator to a known function and then combines the real and the artificial problems. Here one must determine 12 random numbers to break the disguise.

This paper is organized as follows. Section 2 presents the general framework for disguises and most of the innovative approaches. These include:

- *One time random sequences* that are highly resistant to statistical attack.
- *Random mathematical objects* including matrices and vectors previously introduced in [3] plus random functions, operators and mappings introduced here.
- *Linear operator modification* to disguise computation involving operators.
- *Mathematical object modification* using random objects to create disguises that are highly resistant to approximation theoretical attacks.
- *Domain/dimensional/coordinate system modifications* to disguise computation via transformations.
- *Identities and partitions of unity* to create disguises highly resistant to symbolic code analysis attacks.
- *Master key/sub-key methodology* that allows for a single key from which a large number of independent sub-keys may be derived.
- *Disguise programs* which specify disguises concisely and retain the information exactly for disguise inversion.

Section 3 discusses the application of these approaches to the following:

- Linear algebra (matrix multiplication, matrix inversion, systems of equations, convolutions).

- Sorting.
- String pattern matching.
- Template matching for images with least squares or ℓ_1 norms in minimal order time.

Section 4 presents an analysis of the strengths of the disguises and identifies the three principal avenues of attack (statistical, approximation theoretic and symbolic code analysis). Section 5 presents a discussion of the cost of the disguise process.

2 GENERAL FRAMEWORK

In this section we show why multiple disguise techniques are necessary and then identify five broad classes of disguises. Within each class there may be several or many *atomic disguises*, the techniques used to create complete disguises. The necessity for multiple disguises illustrates again the different nature of disguises and encryption. Multiple disguises require multiple keys and thus we present a technique to use one *master key* from which many sub-keys may be generated automatically with the property that the discovery of one sub-key does not compromise the master key or any other sub-key. Finally, we present a notation of for *disguise programs* which use the atomic disguises to create complete disguises.

2.1 Need for Multiple Disguises

We believe that no single disguise technique is sufficient for the broad range of scientific computations. The analogy with ordinary disguises is appropriate: one does completely different things to disguise an airplane hanger than one does to disguise a person. In a personal disguise one changes their hair, the face, the clothes, etc., using several different techniques. The same is true for scientific computation.

If we consider just five standard scientific computations, quadrature, ordinary differential equations, optimization, and matrix multiplication, we see that none of the atomic disguises applies to all five. We are unable to find a single mathematical disguise technique that is applicable to all five.

2.2 Atomic Disguises

A disguise has three important properties:

- *Invertibility*: After the disguise is applied and the disguised computation made, one must be able to recover the result of the original computation.

- *Security*: Once the disguise is applied, someone (the agent) without the key of the disguise should not be able to discover either the original computation or its result. It must be assumed the agent has all the information about the disguised computation. One can use multiple agents to strengthen the security of a disguise but, ultimately, one must be concerned that these agents might collaborate in an attempt to break the disguise.

- *Cost*: There is a cost to apply the disguise and a cost to invert it. The costs to outsource the computation and to carry it out might be increased by the disguise. Thus, there are four potential sources of cost in using disguises.

The ideal disguise is, of course, invertible, highly secure and cheap. We present disguises for scientific computations that are invertible, quite secure and of reasonable cost. The cost of disguise is often related to the size of the computation in some direct way. Not unexpectedly, we see that increasing security involves increasing the cost.

2.2.1 Random objects

The first class of atomic disguise techniques is to create random objects: numbers, vectors, matrices, functions, parameters, etc. These objects are "mixed into" the computation in some way to disguise it. These objects are created in some way from random numbers which, in turn, use random number generators. If the numbers are truly random, then they must be saved for use in the disguise and inversion process. If they come from a pseudo random number generator, then it is sufficient to save the seed and parameters of the generator.

We strongly advocate that the "identity" of the generator be hidden, not just the seed, if substantial sequences from the generator are used. This can be accomplished by taking a few standard generators (uniform, normal, etc.) and creating *one time random sequences*. To illustrate, assume we have G1 = a uniform generator (with two parameters = upper/lower range), G2 = normal generator (with two parameter = mean and standard deviation), G3 = exponential generator (with two parameters = mean and exponent), and G4 = gamma generator (with two parameters = mean and shape). Choose 12 random numbers; the first 8 are the parameters of the four generators and the other four, α_1, α_2, α_3 and α_4 are used to create the one time random sequence

$$\alpha_1 G1 + \alpha_2 G2 + \alpha_3 G3 + \alpha_4 G4.$$

These random numbers must be scaled appropriately for the computation to be disguised. This is rather straight forward and discussed in a more general context later. Note that in creating this one set of random numbers, we use 16 sub-keys, the 8 parameters, the 4 coefficients, and the 4 seeds of the generators.

Once one has random numbers then it is straight forward to create random vectors, matrices and arrays for use in disguises. Objects with integer (or discrete) values such as permutations also can be created from random numbers in a straight forward manner.

Functions play a central role in scientific computations and we need to be able to choose random sets of them also. The technique to do this is as follows. Choose a basis of 10 or 30 functions for a high dimensional space F of functions. Then choose a random point in F to obtain a random function. The basis must be chosen with some care for this process to be useful. The functions must have high linear independence (otherwise the inversion process might be unstable) and their domains and ranges must be scaled compatibly with the computation to be disguised. The scaling is straight forward (but tedious). We propose to make F a *one time random space* as illustrated by the following example. Enclose the domain of the computation in a box (interval, rectangle, box, ..., depending on the dimension). Choose a random rectangular grid in the box with 10 lines in each dimension and assuring a minimum separation (say 3%). Create K sets of random function values at all the grid points (including the boundaries), one set for each basis function desired. These values are to be in the desired range. Interpolate these values by cubic splines to create K basis functions. These functions are smooth (they have two continuous derivatives). Add to this set of K basis functions a basis for the quadratic polynomials.

The approach illustrated above can be modified to make many kinds of one time random spaces of functions. If functions with local support are needed, just replace the cubic spline by Hermite quintics. If the functions are to vary more in one part of the domain than the other, then refine the grid in that part. If the functions are to be more or less smooth, then adjust the degree and smoothness of the splines appropriately. The computational techniques for creating all these splines (or piecewise polynomials) are given in the book by deBoor [9].

2.2.2 Linear operator modification

Many scientific computations involve linear operators, e.g.,

linear equations:	Solve $Ax = b$
differential equations:	Solve $y'' + \cos(x)y' + x^2y = 1 - xe^{-x}$

These operator equations are of the form $Lu = b$ and can be disguised by changing the objects in them (this is discussed later) or by exploiting their linearity. Linearity is exploited by randomly choosing v like u, i.e., v is the same type of object, and then solving $L(u + v) = b + Lv$ where Lv is evaluated to be the same type as b. This disguises the solution buy still reveals the operator. It will be seen later that, for equations involving mathematical functions, it is very advantageous to choose v to be a combination of a random function and functions that appear in the equation. Thus one could choose $v(x)$ in the above differential equation to be $v_{Ran}(x) + 4.2\cos(x) - 2.034xe^{-x}$. This helps with disguises of the operator.

2.2.3 Object modification

Scientific computations involve the objects discussed above which can often be manipulated by addition or multiplication to disguise the computation. The related techniques of substituting

equivalent objects is discussed later. We illustrate these by examples.

1. *Addition to integrals.* To disguise the evaluation of

$$\int_0^1 \sqrt{x}\cos(x+3)dx$$

add a random function to $\sqrt{x}\cos(x+3)$. Note that the basis of the space F chosen in Section 2.2.1 makes it trivial to evaluate the integral of one of these functions.

2. *Multiply by matrices.* To disguise the solution of $Ax = b$ choose two random diagonal matrices, D_1 and D_2, compute $B = D_1AD_2$ and outsource the problem

$$By = D_1b.$$

The solution x is then obtained from $x = D_2y$.

2.2.4 Domain and dimension modification

Many problems allow one to modify the domain or dimensions (matrix/vector problems) by expansion, restriction, splitting or rearrangement. Each of these is illustrated.

1. *Expansions.* The evaluation of

$$\int_0^1 \sqrt{x}\cos(x+3)dx$$

or solution of a problem of related type on [3,5]:

$$y'(x) = (x+y)e^{-xy}, \quad y(3) = 1$$

can be modified by expanding [0,1] to [0,2] or [3,5] to [2,5]. In the first case one selects a random function $u(x)$ from F with $u(1) = \cos(4)$, integrates it on [1,2] and extends $\sqrt{x}\cos(x+3)$ to [0,2] using $u(x)$. In the second case one chooses a random function $u(x)$ from F with $u(3) = 1$, $u'(3) = 4e^{-3}$. One computes its derivative $u'(x)$, its value $u(2)$ and solves the problem

$$\begin{array}{rcll} y'(x) & = & (x+y)e^{-xy} & \text{on } [3,5] \\ & = & u'(x) & \text{on } [2,3] \end{array}$$

with the initial condition $y'(2) = u'(2)$.

2. *Restriction.* One can decrease the dimension of a linear algebra computation or problem by having the customer perform a tiny piece of it and sending the rest to the agent. For example, in solving $Ax = b$, one chooses an unknown at random, eliminates it by Gauss elimination and then sends the remaining computation to the agent. This changes the order of the matrix by 1 and, further, it modifies all the remaining elements of A and b. This does not disguise the solution except by hiding one unknown.

3. *Splitting.* Many scientific problems can be partitioned into equivalent subproblems simply by splitting the domain. This is trivial in the case of quadrature and the linear algebra problem $Ax = b$ can split by partitioning

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

and creating two linear equations

$$A_{11}x_1 = b_1 - A_{12}x_2$$

$$(A_{22} - A_{21}A_{11}^{-1}A_{12})x_2 = b_2 - A_{11}^{-1}b_1.$$

(See Section 3.1 for more on disguising the $Ax = b$ problem.)

The differential equation problem

$$y'(x) = (x + y)e^{-xy} \quad y'(3) = 1 \quad \text{on} \quad [3,5]$$

can be split into

$$\begin{array}{llll} y'(x) = (x + y)e^{-xy} & y'(3) = 1, & \text{on} & [3,4] \\ y'(x) = (x + y)e^{-xy} & y'(u) = \text{ as computed}, & \text{on} & [4,5] \end{array}$$

An important use of splitting problems is to be able to disguise the different parts in different ways so as to strengthen the overall security of the disguise.

2.2.5 Coordinate system changes

Changes of coordinate systems are effective disguises for many scientific computations. They must be chosen carefully, however; if they are too simple they do not hide much, if they are too complex they are hard to invert. For discrete problems (e.g., linear algebra) permutations of the indices play a similar role. Random permutation matrices are easy to generate and simple to apply/invert. Disguises based on coordinate system changes are one of the few effective disguises for optimization and solutions of nonlinear systems.

We illustrate the range of possibilities by considering a particular problem, disguising the two-dimensional partial differential equation (PDE) problem

$$\begin{array}{rll} \bigtriangledown^2 f(x,y) + (6.2 + 12\sin(x+y))f = g_1(x,y) & (x,y) \text{ in } R \\ f(x,y) = b_1(x,y) & (x,y) \text{ in } R_1 \\ f(x,y) = b_2(x,y) & (x,y) \text{ in } R_2 \\ \frac{\partial f(x,y)}{\partial x} + g_2(x,y)f(x,y) = b_3(x,y) & (x,y) \text{ in } R_3 \end{array}$$

Here R_1, R_2 and R_3 comprise the boundary of R. To implement the change of coordinates, $u = u(x,y)$, $v = v(x,y)$ one must be able to

(1) invert the change, find the functions $x = x(u,v)$, $y = y(u,v)$

(2) compute derivatives needed in the PDE, e.g.,

$$\frac{\partial^2 f}{\partial x^2} = \frac{\partial^2 f}{\partial u^2}\left(\frac{\partial^2 u}{\partial x^2}\right)^2 + \frac{\partial f}{\partial u}\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 f}{\partial v^2}\left(\frac{\partial v}{\partial x}\right)^2 + \frac{\partial f}{\partial v}\frac{\partial^2 u}{\partial x^2}$$

The change of coordinates produces an equivalent PDE problem on some domain S in (u,v) space of the form

$$\begin{array}{rl} \sum_{i,j=0}^{2} a_{ij}(u,v)\frac{\partial^i \partial^i}{\partial u^i \partial u^j} f(u,v) = h_1(u,v) & (u,v) \in S \\ f(u,v) = c_1(u,v) & (u,v) \in S_1 \\ f(u,v) = c_2(u,v) & (u,v) \in S_2 \\ d_1(u,v)\frac{\partial f(u,v)}{\partial u} + d_2(u,v)\frac{\partial f(u,v)}{\partial v} + \ d_3(u,v)f(u,v) = c_3(u,v) & (u,v) \in S_3 \end{array}$$

Here S_1, S_2 and S_3 are the images of R_1, R_2, R_3 and the functions $a_{ij}(u,v)$, $h_1(u,v)$, $c_i(u,v)$, $d_i(u,v)$ are obtained from substituting in the changes of variables and collecting terms.

A completely general coordinate change is excellent in disguising the PDE problem. It changes the domain and all its problem coefficient functions (usually in complex ways). There are a number of coordinate changes where the inverse is known explicitly, but these are few enough that restricting oneself to them might weaken the security of disguise. For other coordinate changes the inverse must be determined numerically. This process has been studied in some depth by Ribbens [26, 27] and reliable, efficient procedures developed which can be used to create *one time coordinate changes* using parameterized mappings with randomly chosen parameters.

An intermediate variation here is to make coordinate changes in the variables independently. That is, we have $u = u(x)$ and $v = v(y)$. This approach substantially reduces the cost and complexity of the change and yet allows for randomly parameterized one time coordinate changes.

2.2.6 Identities and partitions of unity

One class of attacks on disguises is to make a symbolic analysis of the computational codes in order to separate the "random objects" from the original objects (see Section 4.3.3). Considerable protection against such attacks can be achieved by using mathematical identities to disguise the mathematical models involved in the computations. Examples of such identities include

$$a^2 - ax + x^2 = (a^3 + x^3)/(a + x)$$
$$\log(xy) = \log x + \log y$$
$$1 + x = (1 - x^2)/(1 - x)$$
$$\sin(x + y) = \sin x \cos y + \cos x \sin y$$
$$\cos^2 x = \sin^2 y + \cos(x + y)\cos(x - y)$$
$$p\cos x + q\sin(y) = \sqrt{p^2 + q^2}\cos(x - \cos^{-1}(p/\sqrt{p^2 + q^2}))$$
$$\sin(3(x + y)) = 3\sin(x + y) - 4\sin^3(x + y)$$

Thus, if any component of these identities appears symbolically in a computation, the equivalent expression can be substituted to disguise the problem. A general source of useful identities for disguise comes from the basic tools for manipulating mathematics, e.g., changes of representation of polynomials (power form, factored form, Newton form, Lagrange form, orthogonal polynomial basis, etc.), partial fraction expansions or series expansions. The above classical identities are purposely simple, disguises need complex identities, and these may be created by involving lessor known functions and compositions. For example, the simple relations involving the Gamma, Psi and Struve functions [2]

$$, (x + 1) = x, (x), \psi(x + 1) = \psi(x) + 1/x, H_{1/2}(x) = (2/\pi x)^{1/2}(1 - \cos x)$$

can be combined with the above to produce

$$\begin{aligned} \sin(x) &= [\sin(\psi(1 + 1/x) + x) - \sin(\psi(1/x))\cos x]/\cos(\psi(1/x)) \\ \log(x) &= \log, (x) + \log(, (x + 1)H_{1/2}(x)) - \log(1 - \cos x) + 1/2\log(\pi x/2) \end{aligned}$$

Identities which equal 1 are called *partitions of unity* and they can be used anywhere in a computation. Examples include

$$\sin^2 x + \cos^2 x = 1$$
$$\sec^2(x + y) - \tan^2(x + y) = 1$$
$$(\tan x + \tan y)/\tan(x + y) + \tan x \tan y = 1$$
$$b_1(r, x) + b_2(r, s, x) + b_3(r, s, x) + b_4(s, x) = 1$$

where the b_i are "hat" functions defined by

$$\begin{aligned} b_1(r, x) &= \max(1 - x/r, 0) \\ b_2(r, s, x) &= \max(0, \min(x/r, (s - x)/(s - r))) \\ b_3(r, s, x) &= \max(0, \min((x - r)/(s - r), (1 - x)/(1 - s))) \\ b_4(x, s) &= \max(0, (x - s)/(1 - s)) \end{aligned}$$

each of which is a piecewise linear function with breakpoints at 0, r, s and/or 1. Generalizations of this partition of unity are known for an arbitrary number of functions, arbitrary polynomial degree,

arbitrary breakpoints, and arbitrary smoothness (less than the polynomial degree). Partitions of unity allow one to introduce unfamiliar and unrelated functions into symbolic expression. Thus the second partition above becomes

$$\sec^2(y_7(x) + u(1.07296, x)) - \tan^2(y_7(x) + u(1.07296, x)) = 1$$

where $y_7(x)$ is the Bessel function of fractional order and $u(1.07296, x)$ is the parabolic cylinder function. The Guide to Available Mathematical Software [5] lists 48 *classes* of functions for which library software is available.

Finally, we note that using functions and constants that appear in the original expression strengthens disguises significantly. Thus, if 2.70532, x^2, $\cos(x)$ and $\log(x)$ initially appear in an ordinary differential equation, one should use identities that involve these objects or closely related ones, e.g., 1.70532, $2x^2 - 1$, $\cos(2x)$ or $\log(x + 1)$. Recall in elementary trigonometry courses that the establishment of identities is one of the most difficult topics (even given the knowledge that the two expressions are identical), so it is plausible that using several identities in a mathematical model provides very high security.

One can also create *one time identities* as follows. There exists well known, reliable library programs that compute the best piecewise polynomial approximation to a given function $f(x)$ with either specified or variable breakpoints [8, 9]. The number of breakpoints and/or polynomial degrees can be increased to provide arbitrary precision in these approximations. Thus given that

$$f(x) = \sin(2.715x + 0.12346)/(1.2097 + x^{1.07654})$$

or that $f(x)$ is computed by a 1000 line code, one can use these library routines to replace $f(x)$ by a code that merely evaluates a piecewise polynomial with "appropriate" coefficients and breakpoints. One time identities may also use the classical mathematical special functions that have parameters, e.g., incomplete gamma and beta functions, Bessel function, Mathieu functions, spheroidal wave functions, parabolic cylinder functions [2].

2.3 Key Processing

The atomic disguises usually involve a substantial number of keys (parameters of random numbers generators, random numbers, etc.) that partly define the disguise process. It is thus desirable to avoid keeping and labeling these keys individually and we present a technique that uses one *master key* to create an arbitrary number of *derived sub-keys*. Let K be the master key and k_i, $i = 1, 2, \ldots, N$ be the sub-keys. The sub-keys k_i are derived from K by a procedure P such as the following. We assume K is represented as a long bit string (a 16 character key K generates 128 bits) and we have a random number generator G. For each $i = 1, 2, \ldots, N$ we generate randomly a bit string of length 128 and use it as a mask on the representation of K (i.e., we select those bits of K where the random bit string is 1). For each i this produces a bit string of about 64 (in our

example) bits or a full word for a 64 bit computer. Thus, with a single key K and a fixed random number generator G, we can create, say, many thousands of sub-keys so that

- Each k_i is easily derived from K.
- Knowing even a substantial set of the k_i gives no information about K even if the generation procedure P is known.

Recall that many of the sub-keys are, in turn, seeds or parameters for a variety of random number generators so that massive sets of random numbers can be used without fear of revealing the master key or other sub-keys even if a statistical attack on this aspect of the disguise is successful (which is very unlikely, see Section 4.2.1).

Implicit in this discussion is that we envisage a substantial problem solving environment, call it *Atri*, for outstanding disguises which manages the disguise process and its inversion for the user. Of course, one can do all this by hand and, if super-security is essential, this might be advisable as one must assume a resourceful and determined attacker would have complete knowledge of Atri.

2.4 Disguise Programs

It should now be clear that outsourcing disguises involve several steps and that one must carefully record the steps of the disguise process, as well as record the master key. At this point we ignore cost issues, e.g., is it better to save random numbers used or to recreate them when needed? We focus entirely on the questions of

- Specifying the disguise in reasonably efficient ways.
- Retaining the information so the disguise can be inverted.

The second question is a subset of the first, in that it is often possible to invert a disguise without knowing everything about a disguise. This is illustrated in the earlier simple examples (Section 1.3) where,

- (1.3.1) one does not need to know P_2,
- (1..3.2) one does not need to know $g(x)$, only I_1 is needed.

It is clear that Atri should be built using, at least internally, a formal language for specifying disguises and the process for inverting them. We do not attempt to define such a language here, but rather illustrate how such a language might appear to a user. First we observe that Atri must

- Provide direct access to the objects (variables) in the computations. These might be single elements (numbers, arrays, functions, ...) or compositions (expressions, equations, problems, coordinate systems, ...).

- Provide a set of procedures to generate random objects as in Section 2.2.1 with specified attributes, e.g., size, dimension, smoothness, domain, random number generator type (seed/parameters, ...), etc.
- Allow old/new objects to be combined with appropriate (type dependent) operators (e.g., add, multiply, transform, substitute, insert, apply, ...).
- Provide simple sequencing control of the disguise, outsourcing, retrieval and disguise inversion actions.

To initially illustrate the nature of these programs, we express the simple examples of Section 1.3 as disguise programs. The informal language used is somewhat verbose in order to avoid defining many details explicitly, it is intended to be self-explanatory.

Matrix Multiplication (1.3.1): M1 * M2

Matrix Key K = AmY+JennA-Tygar-BeaU, Sub Key $k(i)$
m = Dimension (M1)
Permutations (1:m), $\pi 1$, $\pi 2$ $\pi 3$ using G_{UNIF} (k(1))
Vector (1:m), α, β, γ using G_{UNIF} (k(2), [1,2])
Matrix (1:m,1:m), $P1$, $P2$, $P3$

```
      P1(i,j)  =  α(i)  if π1(i) = j
               =  0       otherwise
      P2(i,j)  =  β(i)  if π2(i) = j
               =  0       otherwise
      P3(i,j)  =  γ(i)  if π3(i) = j
               =  0       otherwise
```

Matrix (1:m,1:m), X, Y, Z, ANS
X = P1 * M1 * $P2^{-1}$
X = P2 * M2 * $P3^{-1}$
Outsource [Product, X, Y]
Return Z
ANS = $P1^{-1}$ * Z * P3

Quadrature (1.3.2): Integrate f(x) on [a,b]

```
Master Key K = Quadrature_of_f(x)_1.3.2_a,b
Function f(x), h(x); Interval [a,b]
maxf = max(f,a,b); minf = min(f,a,b)
Vector (2:6), P using G_UNIF (k(1), a, b)
   P(1) = a;     P(7) = b
Vector (1:7), V using G_UNIF (k(2), minf, 2 * maxf)
Cubic spline g(x) with g(P(i)) = V(i) for i = 1:7
I1 = Integrate (g,a,b, exact)
h(x) = f(x) + g(x)
Outsource [Integrate, h, a, b, eps]
Return I2
ANS = I2 - I1
```

Edges (1.3.3): Find edges in photo Amalfi.10.29.97

```
Master Key K = Edges_in_Amalfi.10.29.97+ElmaGarMid
Photo Amalfi.10.29.97
(n,m) = Dimension (Amalfi.10.29.97)
r = Max (Amalfi.10.29.97)
(XCOR(1:4), YCOR(1:4)) = Corners (Amalfi.10.29.97)
Vector (1:10), X,Y
   X(1) = XCOR(1); X(10) = XCOR(4); Y(1) = YCOR(1); Y(10) = YCOR(4)
   X(2:9) = G_UNIF (k(1), X(1), X(10))
   Y(2:9) = G_UNIF (k(2), Y(1), Y(10))
Array (1:10,1:10), V using G_UNIF (k(3), 0, r)
Cubic spline S(x,y) with S(X(i), Y(j)) = V(i,j) for i,j = 1:10
H1 = (XCOR(4) - XCOR(1))/(n-1); H2 = (YCOR(4) - YCOR(1))/(m-1)
Array (1:n,1:m) SV
SV(i,j) = S(XCOR(1) + (i-1)*H1, YCOR(1) + (j-1)*H2) for i = 1:n, j = 1:m
Vector (1:4) A using G_UNIF (k(1), 2, 4)
NUXCOR(1) = A(1); NUXCOR(4) = A(1) + A(2)
NUYCOR(1) = A(3); NUYCOR(4) = A(3) + 1/A(4)
NUXCOR(2) = NUXCOR(4); NUXCOR(3) = NUXCOR(1)
NUYCOR(2) = NUYCOR(4); NUYCOR(3) = NUYCOR(1)
S1 = A(2)/(XCOR(1) - XCOR(4))
S2 = 1/(A(4) * (YCOR(1)-YCOR(4))
I1 = A(1) - S1 * XCOR(1)
I2 = A(3) - S1 * YCOR(1)
MAPS: U = I1 + S1*X; V = I2 + S2*Y
Photo Sphoto = (SV, NUXCOR, NUYCOR, n, m)
Photo Amalfi2 = (array(Amalfi.10.27.97), NUXCOR, NUYCOR, n, m)
Photo Out = Sphoto + Amalfi2
Photo Outedges
Outsource [edges, Out]
Return Outedges
Photo Amalfi.edges = (Array (outedges), XCOR, YCOR, n, m)
```

```
Differential Equations (1.3.4):  Solve 2 point BVP

Function Y(x), a1(x), a2(x), f(x), u(x)
Operator (L): L = y'' + a1(x) * y' + a2(x) * y
Boundary Conditions:  BC = (a, b, Y1, Y2)
ODE: Ly = f, BC
Master Key K = Addis+aBABA+Ethiopia.1948
maxf = max(f,a,b); minf = min(f,a,b)
Vector (2:6), P using G_UNIF (k(1), a, b)
   P(1) = a;     P(7) = b
Vector (1:7), V using G_UNIF (k(2), minf, 2*maxf)
Cubic spline g(x) with g(P(i)) = V(i) for i = 1:7
Boundary Conditions:  BC2 (a, b, Y1 + u(a), Y2 + u(b))
ODE2:  Ly = f + u, BC2
Outsource [Solve-ODE, ODE2, solution = z]
Return z(x)
ANS = z(x) - g(x)
```

We observe the following from these four simple disguises.

- Even simple disguises are rather complex.
- The disguise procedure must be recorded carefully.
- Minor changes in the disguise programs (even if syntactically and semantically correct) change the outsourced problem significantly or, equivalently for a fixed outsourced problem, change the original problem significantly.
- The disguise acquires security both from the random numbers used and from the complexity of the disguise process.
- The Atri environment should provide much higher level ways to specify disguises than these examples use. It should, however, allow one to tailor a disguise in detail as these examples illustrate.

3 APPLICATIONS

3.1 Linear Algebra

These disguise algorithms are taken directly from [3] except that the disguise of solving a linear system has been modified to enhance the numerical stability of the outsourcing process.

3.1.1 Matrix multiplications

A fairly satisfactory disguise is given in section 1.4.1. The following method provides even greater security by adding in a dense random matrix. The following scheme hides a matrix by the sparse random matrices P_i or their inverse, the resulting matrix is further hidden by adding a dense random matrix to it. The details follow.

1. Compute matrices $X = P_1 M_1 P_2^{-1}$ and $Y = P_2 M_2 P_3^{-1}$ as in Section 1.4.1.

2. Select two random $n \times n$ matrices S_1 and S_2 and generate four random numbers β, γ, β', γ' such that
$$(\beta + \gamma)(\beta' + \gamma')(\gamma'\beta - \gamma\beta') \neq 0$$

3. Compute the six matrices $X + S_1$, $Y + S_2$, $\beta X - \gamma S_1$, $\beta Y - \gamma S_2$, $\beta' X - \gamma' S_1$, $\beta' Y - \gamma' S_2$. Outsource to the agent the three matrix multiplications
$$\begin{aligned} W &= (X + S_1)(Y + S_2) && (1)\\ U &= (\beta X - \gamma S_1)(\beta Y - \gamma S_2) && (2)\\ U' &= (\beta' X - \gamma' S_1)(\beta' Y - \gamma' S_2) && (3) \end{aligned}$$
which are returned.

4. Compute the matrices
$$\begin{aligned} V &= (\beta + \gamma)^{-1}(U + \beta\gamma W) && (4)\\ V' &= (\beta' + \gamma')^{-1}(U' + \beta'\gamma' W) && (5) \end{aligned}$$
Observe that $V = \beta XY + \gamma S_1 S_2$, and $V' = \beta' XY + \gamma' S_1 S_2$.

5. Outsource the computation
$$(\gamma'\beta - \gamma\beta')^{-1}(\gamma' V - \gamma V')$$
which equals XY (as can be easily verified - we leave the details to the reader).

6. Compute $M_1 M_2$ from XY by
$$P_1^{-1} XY P_3 = P_1^{-1}(P_1 M_1 P_2^{-1})(P_2 M_2 P_3^{-1})P_3 = M_1 M_2.$$

3.1.2 Matrix inversion

The scheme we describe to invert the $n \times n$ matrix M uses secure matrix multiplication as a subroutine.

1. Select 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 sends us back to Step 1.

2. Outsource the computation to the agent

$$\hat{M} = MS \tag{6}$$

   using secure matrix multiplication. Of course, after this step the agent knows neither M, nor S, nor $\hat{M}$.

3. Generate 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 in Section 1.4.1. 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 compute the matrices

$$Q = P_1\hat{M}P_2^{-1} = P_1MSP_2^{-1} \tag{7}$$
$$R = P_3SP_4^{-1} \tag{8}$$

4. Outsource the computation of Q^{-1} and, if it succeeds, return Q^{-1}. If Q is not invertible, then the agent returns this information. We know that at least one of S or M (possibly both) is non-invertible. and we do the following:

   (a) Test whether S is invertible by first computing $\hat{S} = S_1SS_2$ where S_1 and S_2 are matrices known to be invertible and outsource $\hat{S}$ to the agent for inverting.
   Note: The only interest is whether $\hat{S}$ is invertible or not, not in its actual inverse. The fact we 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 (there are such classes). It is unwise to let S_1 and S_2 be the identity matrices, because by knowing S the agent might learn how we generate these random matrices.

   (b) If the agent can invert $\hat{S}$ then we know S is invertible, and hence that M is not invertible. If the agent says that $\hat{S}$ is not invertible, then we know that S is not invertible. In that case we return to Step 1, i.e., choose another S, etc.

5. Observe that $Q^{-1} = P_2S^{-1}M^{-1}P_1^{-1}$ and compute the matrix

$$T = P_4P_2^{-1}Q^{-1}P_1P_5^{-1}.$$

   It is easily verified that T is equal to $P_4S^{-1}M^{-1}P_5^{-1}$.

6. Outsource the computation of
$$Z = RT$$
using secure matrix multiplication. Of course the random permutations and numbers used within this secure matrix multiplication subroutine are independently generated from those of the above Step 3 (using those of Step 3 could 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. Compute $P_3^{-1} Z P_5$ which equals M^{-1}.

The security of the above follows from:

1. 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 $P_1, \ldots, P_5$ "isolates" 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, so as 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.

3.1.3 Linear system of equations

Consider the system of linear equations $Mx = b$ where M is a square $n \times n$ matrix, b is an n-vector, and x the vector of n unknowns. The scheme we describe uses local processing which takes time $\mathcal{O}(n^2)$ proportional to the size of the input.

1. Select a random $n \times n$ matrix B and a random number $j \in \{1, 2, \ldots, n\}$. Replace the j-th row of B by b. i.e. $B = [B_1, \ldots, B_{j-1}, b, B_{j+1}, \ldots, B_n]$.

2. Generate matrices P_1, P_2, P_3 using the same method as for the P_1 matrix in Steps 1 and 2 in Section 1.4.1. That is, $P_1(i,j) = a_i \delta_{\pi_1(i),j}$, $P_2(i,j) = b_i \delta_{\pi_2(i),j}$, $P_3(i,j) = c_i \delta_{\pi_3(i),j}$, where π_1, π_2, π_3 are random permutations, and where the a_i, b_i, c_i are random numbers.

3. Compute the matrices
$$\hat{M} = P_1 M P_2^{-1} \quad (9)$$
$$\hat{B} = P_1 B P_3^{-1} \quad (10)$$

4. Outsource to the agent the solution of the linear system $\hat{M}x = \hat{B}$. If $\hat{M}$ is singular then the agent returns a message saying so, then we know M is singular. Otherwise the agent returns
$$\hat{X} = \hat{M}^{-1} \hat{B}.$$

5. Compute $X = P_2^{-1}\hat{X}P_3$ which equals $M^{-1}B$, since

$$P_2^{-1}\hat{X}P_3 = P_2^{-1}\hat{M}^{-1}\hat{B}P_3 = P_2^{-1}P_2M^{-1}P_1^{-1}P_1BP_3^{-1}P_3 = M^{-1}B.$$

6. The answer x is the j-th column of X, i.e., $x = X_j$.

The security of this process follows from the fact that b is hidden through the expansion to a matrix B, and then M and B are hidden through random scalings and permutations.

3.1.4 Convolution

Consider the convolution of two vectors M_1 and M_2 of size n, indexed from 0 to $n-1$. Recall that the convolution M of M_1 and M_2 is a new vector of size $2n-1$, denoted $M = M_1 \otimes M_2$, such that

$$M(i) = \sum_{k=0}^{\min(i,n-1)} M_1(k)M_2(i-k).$$

The scheme described below satisfies the requirement that all local computations take $\mathcal{O}(n)$ time.

1. Choose vectors S_1, S_2 of size n randomly. Also choose five positive numbers α, β, γ, β', γ' such that

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

2. Compute locally the six vectors $\alpha M_1 + S_1$, $\alpha M_2 + S_2$, $\beta M_1 - \gamma S_1$, $\beta M_2 - \gamma S_2$, $\beta' M_1 - \gamma' S_1$, $\beta' M_2 - \gamma' S_2$.

3. Outsource to the agent the three convolutions:

$$W = (\alpha M_1 + S_1) \otimes (\alpha M_2 + S_2) \quad (11)$$
$$U = (\beta M_1 - \gamma S_1) \otimes (\beta M_2 - \gamma S_2) \quad (12)$$
$$U' = (\beta' M_1 - \gamma' S_1) \otimes (\beta' M_2 - \gamma' S_2) \quad (13)$$

   which are returned.

4. Compute locally the vectors

$$V = (\beta + \alpha\gamma)^{-1}(\alpha U + \beta\gamma W) \quad (14)$$
$$V' = (\beta' + \alpha\gamma')^{-1}(\alpha U' + \beta'\gamma' W) \quad (15)$$

   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. Compute

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

   which equals $M_1 \otimes M_2$.

3.1.5 Hiding Dimensions in Linear Algebra Problems

. The report [3] systematically describes in detail how to hide the dimensions of linear algebra problems by either increasing or decreasing them. Here we describe their techniques in a less detailed form. The basic ideas are seen from the multiplication of M_1M_2 where M_1 and M_2 are of dimension $a \times b$ and $b \times c$, respectively. First note that one can add k random rows to M_1 and/or k random columns to M_2 and then just ignore the extra rows and/or columns generated in the product. One may enlarge b by adding k columns to M_1 and k rows to M_2 which are related. The simple relationship proposed in [3] is to take the odd-numbered extra columns of M_1 and the even-numbered rows of M_2 to be identically zero. The other additional elements in M_1 and M_2 could be chosen randomly. The result is that the product of the augmented matrices is the same as M_1M_2.

To reduce the dimensions a or c, one can merely partition M_1 by rows or M_2 by columns and perform two smaller matrix multiplications. To reduce b one can partition both M_1 and M_2 of compatible dimensions and then compute M_1M_2 by eight smaller matrix multiplications. The total arithmetic work is unchanged.

To enlarge the dimension n in inverting the $n \times n$ matrix M one uses the same scheme as for matrix multiplication and at step 4 the matrix Q is augmented by a $k \times k$ random matrix S so that the matrix $\begin{pmatrix} Q & 0 \\ 0 & S \end{pmatrix}$ is reducible. The inversion of this larger matrix is done using the matrix inversion algorithm which hides the special structure of the enlarged matrix. To reduce the dimension n partition Q into

$$\begin{pmatrix} X & Y \\ V & W \end{pmatrix}.$$

If X and $Y - WX^{-1}V$ are invertible (and such a partition can be made) then the inverse of Q is the partitioned matrix

$$\begin{pmatrix} X^{-1} + X^{-1}VD^{-1}WX^{-1} & -X^{-1}VD^{-1} \\ -D^{-1}WX^{-1} & D^{-1} \end{pmatrix}.$$

To decrease the dimension n in the linear system $Mx = b$ one can use the scheme used for matrix inversion. To enlarge n we create the system

$$\begin{pmatrix} M & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ Sy \end{pmatrix}$$

where S is a $k \times k$ invertible matrix (say, random) and y is random vector. One then applies the previous algorithm which hides the special structure of the enlarged matrix. The zero block matrices above can be replaced by random matrices with a minor change in the right side.

To increase the dimension n of the problem $M_1 \otimes M_2$ one can merely pad the vectors M_1 and M_2 by adding k zeros. To decrease the dimension we first note that $M_1 \otimes M_2$ can be replaced by three convolutions of size $n/2$

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

$$(M_1^{(even)} - M_1^{(odd)}) \otimes (M_2^{(even)} - M_2^{(odd)})$$

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

where $M^{(odd)}$, $M^{(even)}$ mean the vector of odd and even indexed elements of M, respectively. From these convolutions one can easily find the products on the right side of the relationships

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

where $Shift(x)$ shifts the vector x by one position.

3.2 Sorting

Consider the problem of sorting a sequence of numbers $E = \{e_1, \ldots, e_n\}$. This can be outsourced securely by selecting a strictly increasing function $f : E \to \mathbb{R}$, such as

$$f(x) = \alpha + \beta(x + \gamma)^3$$

where $\beta > 0$. The scheme we describe below assumes this particular $f(x)$.

1. Choose α, β, and γ so that $\beta > 0$.

2. Choose a random sorted sequence $\Lambda = \{\lambda_1, \ldots, \lambda_l\}$ of l numbers by randomly "walking" on the real line from MIN to MAX where MIN is smaller than the smallest number in E and MAX is larger than the largest number in E. Let $\Delta = (MAX - MIN)/n$ and the random "walking" is implemented as follows.

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

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

   (c) Continue in the same way until you go past MAX. The total number of elements generated is l.

   Observe that Λ is sorted by the construction such that the expected value for the increment is Δ, therefore the expected value for l is $(MAX - MIN)/\Delta = n$.

3. Compute the sequences

$$E' = f(E) \quad (16)$$
$$\Lambda' = f(\Lambda), \quad (17)$$

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

4. Concatenate the sequence Λ' to E', obtaining $W = E' \cup \Lambda'$. Randomly permute W before outsourcing it to the agent, who returns the sorted result W'.

5. Remove Λ' from W' to produce the sorted sequence E'. This can be done in a linear time since both W' and Λ' are sorted.

6. Compute $E = f^{-1}(E')$.

The above scheme reveals n since the number of items sent to the agent has expected value $2n$. To modify n, we can let $\Delta = (MAX - MIN)/m$ in Step 2, where m is a number independent of n. The argument at the end of Step 2 shows that the expected value for the size of Λ is m, therefore the size of the sequence the agent received is $m + n$, and we can hide the size of problem by expanding the size this way.

3.3 Template Matching in Image Analysis

Given an $N \times N$ image I and a smaller $n \times n$ image P, consider the computation of an $(N - n + 1) \times (N - n + 1)$ score matrix $C_{I,P}$ is of the form

$$C_{I,P}(i,j) = \sum_{k=0}^{n-1} \sum_{k'=0}^{n-1} f(I(i+k, j+k'), P((k,k')), \qquad 0 \le i, j \le N - n,$$

for some function f. Score matrices are often used in image analysis, specifically in *template matching*, when one is trying to determine whether (and where) an object occurs in an image. A small $C_{I,P}(i,j)$ indicates an approximate occurrence of the object P in the image I (a zero indicates an exact occurrence). Frequent choices for the function f are $f(x,y) = (x-y)^2$ and $f(x,y) = |x-y|$ [16, 18]. We consider how to securely outsource the computation of C for these two functions.

3.3.1 The case $f(x,y) = (x-y)^2$

In the scheme described all local processing takes time proportional to the size $O(N^2)$ of the input.

1. Select a random $N \times N$ matrix S_1, and a random $n \times n$ matrix S_2. Generate five positive random numbers α, β, γ, β', γ' such that

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

If the above is violated then we discard the numbers chosen and repeat the choice of five numbers. Observe 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. Compute locally the six matrices of $\alpha I + S_1$, $\alpha P + S_2$, $\beta I - \gamma S_1$, $\beta P - \gamma S_2$, $\beta' I - \gamma' S_1$, $\beta' P - \gamma' S_2$.

3. Outsource to the agent the computation of three score matrices $C_{X,Y}$, one for each pair X, Y of matrices received:

$$\begin{aligned} W &= C_{(\alpha I+S_1),(\alpha P+S_2)} && (1)\\ U &= C_{(\beta I-\gamma S_1),(\beta P-\gamma S_2)} && (2)\\ U' &= C_{(\beta' I-\gamma' S_1),(\beta' P-\gamma' S_2)} && (3) \end{aligned}$$

which are returned.

4. Compute locally the matrices

$$\begin{aligned} V &= (\beta + \alpha\gamma)^{-1}(\alpha U + \beta\gamma W) && (4)\\ V' &= (\beta' + \alpha\gamma')^{-1}(\alpha U' + \beta'\gamma' W). && (5) \end{aligned}$$

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

The third matrix equals $C_{I,P}$ (as is easily verified).

The security of the above scheme is based on the fact that the six matrices received by the agent do not enable it to discover I or P, as the agent does not know the numbers α, β, γ, β', γ' and the matrices S_1, S_2. The disguise program based on this algorithm is as follows:

```
Master Key K = Score.one.for.Amira-please, Sub Key k(i)
Matrix (1:N,1:N), S_1 using G_UNIF (k(1), [0,L])
Matrix (1:n,1:n), S_2 using G_UNIF (k(2), [0,L])
Until (β + αγ)(β' + αγ')(γ'β − γβ') ≠ 0 do
Real, α, β, γ, β'γ' using G_UNIF (k(3), [-1,1])
End do
Matrix (1:N,1:N), I_1=αI+S_1, I_2=βI -γS_1, I_3=β'I -γ'S_1
Matrix (1:n,1:n), P_1=αP+S_2, P_2=βP -γS_2, P_3=β'P -γ'S_2
Outsource [Compute score, C(I_1,P_1)]
Outsource [Compute score, C(I_2,P_2)]
Outsource [Compute score, C(I_3,P_3)]
Return W_1, W_2, W_3
Matrix (1:N-n,1:N-n) V = (β + αγ)^{-1} (α W_2 + βγ W_1),
                       V' = (β' + αγ')^{-1} (α W_3 + β'γ' W_1)
ANS = α^{-1}(γ'β − γβ')^{-1}(γ'V - γV')
```

3.3.2 The case $f(x, y) = |x - y|$

The algorithm uses as a subroutine a two-dimensional version of the securely outsourced convolution technique in Section 3.1.4. Let A be the alphabet, i.e., the set of symbols that appear in I or P. For every symbol $x \in A$ we do the following:

1. We replace, in I, every symbol other than x by 0 (every x in I stays the same). Let I_x be the resulting image.

2. We replace every symbol that is $\leq x$ by 1 in P, and replace every other symbol by 0. Let P_x be the resulting image. Augment P_x into an $N \times N$ matrix Π_x by padding it with zeroes.

3. Outsource the computation of the score matrix

$$D_x(i,j) = \sum_{k=0}^{n-1} \sum_{k'}^{n-1} I_x(i+k, j+k')\Pi_x(k,k'), \qquad 0 \leq i,j \leq N-n.$$

   This is essentially a 2-dimensional convolution, and it can be securely outsourced using a method similar to the one given in Section 3.1.4.

4. We replace, in P, every symbol other than x by 0 (every x in P stays the same). Let P'_x be the resulting image. Augment P'_x into an $N \times N$ matrix Π'_x by padding it with zeroes.

5. We replace every symbol that is $< x$ by 1 in I, and every other symbol by 0. Let I'_x be the resulting image.

6. Outsource the computation of the score matrix

$$D'_x(i,j) = \sum_{k=0}^{n-1} \sum_{k'}^{n-1} I'_x(i+k, j+k')\Pi_x(k,k'), \qquad 0 \le i,j \le N-n.$$

   This is done securely as above.

7. Compute locally

$$C_{I,P} = \sum_{x \in A} (D_x + D'_x).$$

Thus the computation of $C_{I,P}$ for the case $f(x,y) = |x-y|$ can be done by means of $O(|A|)$ two-dimensional convolutions (which can be securely outsourced). This is reasonable for small-size alphabets (binary, etc.). However, for large alphabets, the above solution has a considerable extra cost compared to computing $C_{I,P}$ directly. The number of convolutions can be reduced by using convolutions only for symbols with many occurrences in I and P, and "brute force" (locally) for the other symbols. However, this still has the disadvantage of having a considerable local computational burden. Thus our solution for the case $f(x,y) = |x-y|$ is less satisfactory, for large $|A|$, than our solution for the case $f(x,y) = (x-y)^2$.

This algorithm illustrates that "programming scientific disguises" requires a relatively complete programming language which must be supported by the Atri system. Of course,as a problem solving environment, Atri would have high level natural constructs to invoke such a common procedure as this one. But, for novel situations, Atri also needs to provide the capability to write "ordinary" procedural programs. The disguised program for this algorithm uses the external procedure called *2D-Convolution* and the outsourced computations are in it. The disguise program is as follows:

```
Master Key = Sixteen-Twelve-Eight9403, Sub Key k(i)
Matrix (1:N,1:N), I
Matrix (1:n,1:n), P
For ℓ = 0 to L do
Matrix (1:N,1:N), I_ℓ(i,j) = 0 if I(i,j) ≠ ℓ
                  else I_ℓ(i,j) = ℓ
Matrix (1:n,1:n), P_ℓ(i,j) = 1 if P(i,j) ≤ ℓ
                  else P_ℓ(i,j) = 0
Matrix (1:N,1:N), T_ℓ(i,j) = P_ℓ (i,j) if 1 ≤ i,j ≤ n
                  else T_ℓ(i,j) = 0
Comment:  The first outsourcing occurs here
Invoke procedure [2D-Convolution, I_ℓ, T_ℓ]
Return Matrix (1:N-n,1:N-n) D_ℓ
End do
For ℓ = 0 L do
Matrix (1:N,1:N), I_ℓ(i,j) = 1 if I(i,j) < ℓ
                  else I_ℓ(i,j) = 0
Matrix (1:n,1:n), P_ℓ(i,j) = 0 if P(i,j) ≠ ℓ
                  else P_ℓ(i,j) = P(i,j)
Matrix (1:N,1:N), T_ℓ(i,j) = P_ℓ (i,j) if 1 ≤ i,j ≤ n
                  else T_ℓ(i,j) = 0
Comment:  The second outsourcing occurs here
Invoke procedure [2D-Convolution, I_ℓ, T_ℓ]
Return Matrix (1:N-n,1:N-n) D'_ℓ
End do
ANS C_{I,P} = Sum [D_ℓ + D'_ℓ , ℓ = 0 to L]
```

Of course this disguise constitutes only a first step in the general secure outsourcing of image analysis; the literature contains many other measures for image comparison and interesting new ones continue to be proposed (for example, see [6] and the papers it references).

3.4 String Pattern Matching

Let T be a text string of length N, P be a pattern of length $n(n \leq N)$, both over alphabet A. We seek a score vector $C_{T,P}$ such that $C_{T,P}(i)$ is the number of positions at which the pattern symbols equal their corresponding text symbols when the pattern is positioned under the substring of T that begins at position i of T, i.e., it is $\sum_{k=0}^{n-1} \delta_{T(k+i),P(i)}$ where $\delta_{x,y}$ equals one if $x = y$ and zero otherwise.

For every symbol $x \in A$ we do the following:

1. Replace, in both T and P, every symbol other than x by 0, and every x by 1. Let T_x and P_x be the resulting text and pattern, respectively. Augment P_x into an length N string Π_x by padding it with zeros.

2. Outsource the computation of

$$D_x(i) = \sum_{k=0}^{n-1} I_x(i+k)\Pi_x(k), \qquad 0 \le i \le N-n.$$

   This is essentially a convolution, and it can be securely outsourced using the method given in Section 3.1.4. It is easily seen that $C_{T,P}$ equals $\sum_{x \in A} D_x$.

4 SECURITY ANALYSIS

4.1 Breaking Disguises

The nature of disguises is that they may be broken completely (i.e., the disguise program is discovered) or, more likely, they are broken *approximately*. That is, one has ascertained with some level of uncertainty some or all of the objects in the original computation. For the example program Quadrature (1.3.2) of Section 2l4, one might have ascertained.

Object	*Certainty*
Computation = Integration	100%
Interval = $[a, b]$	100%
$A1(x)$ with $\lvert f - A1\rvert \le 0.25$	60%
$A2(x)$ with $\lvert f - A2\rvert \le 0.05$	8%
$I3$ with $\lvert I3 - ANS\rvert \le 0.8$	47%
$I4$ with $\lvert I4 - ANS\rvert \le 0.1$	4%
$I5$ with $\lvert I5 - ANS\rvert \le 0.02$	0.03%

Here $A1(x)$ and $A2(x)$ are functions that have been determined somehow. Thus we see that there is a continuum of certainty in breaking a disguise which varies from 100% to none (no information at all). Indeed, an attacker might not even be able to identify all the objects in the original computation. On the other hand, an attacker might obtain all the essential information about the original computation without learning any part of the disguise program exactly. The probabilistic nature of breaking disguises comes both from the use of random numbers and the uncertainty in the behavior of the disguise program itself (as mentioned earlier). Of course, an attacker could only guess at the levels of certainty about the object information obtained.

This situation again illustrates the difference between disguise and encryption; in the latter one usually goes quickly (even instantly) from no information to a compete break. Thus one cannot expect to have precise measurements of the degree to which a disguise is broken or of the strength of a disguise. Indeed, the strength is very problem dependent and even attacker dependent. For one computation an attacker may be completely satisfied with $\pm$ 15% accuracy in knowledge about the original problem while in another computation even $\pm$ 0.01% knowledge is useless.

4.2 Attack Strategies and Defenses

Three rather unrelated attack strategies are discussed here. We exclude non-analytical strategies which, for example, incorporate knowledge about the customer (e.g., the company does oil/gas exploration and the customer's net address is department 13 in Mobile, Alabama), or which attempt to penetrate (physically or electronically) the customer's premises. History suggests that the non-analytical strategies are the most likely to succeed when strong analytical security techniques are used.

4.2.1 Statistical attacks

Knowledge of the Atri environment or casual examination immediately leads the attacker to attempt to derive information about the random number generators used. A determined attacker can check all the numbers in the outsourced computation against all the numbers particular random generators produce. This is an *exhaustive match attack*. While the cycle lengths of the generators are very long, one cannot be complacent about the risk of this approach as we see teraflops or petaflops computers coming into use. There are three defenses against this attack:

1. Use random number generators with (real and random) parameters. With 32 bit reals for two parameters, this increases the cost of an exhaustive match of numbers by a factor of about 10^{14}. This should be ample to defeat this attack. Note also that an exact match no longer breaks the random number generator as sub-sequences from random sequences with different parameters will "cross" from time to time. On the other hand, sometimes one is constrained in the choice of parameters, e.g., it might be necessary that the numbers generated "fill" the interval [0,1] and not go outside it.

2. Restart the random number generators from time to time with new sub-keys. Similarly, one can change the random number generator used from time to time. Thus using a new seed for every 1,000 or 10,000 numbers in a sequence of 10,000,000 numbers greatly reduces the value of having identified one seed or parameter set.

3. Use combinations of random number sequences as mentioned in Section 2.2.1. If certain simple constraints are required, e.g., all values lie in [0,1], one can use a rejection technique to impose such constraints.

In summary, we conclude that even modest care will prevent an exhaustive match attack from succeeding.

An alternate attack is to attempt to determine the parameters of the probability distribution used to generate a sequence of random numbers. We call this a *parameter statistics attack* and it is illustrated by the simple example in Section 1.2. In general, one can estimate the moments of the probability distribution by computing the moments of the sample sequence of generated random numbers. The mean of the sample of size N converges to the mean of the distribution with an error that behaves like $O(1\sqrt{N})$. This same rate of convergence holds for almost any moment and any distribution likely to be used in a disguise. This rate of convergence is slow by not impossibly so, a sample of 10,000,000 should provide estimates of moments with accuracy of about 0.03%. There are three defenses against this attack:

1. Use random number generators with complex probability distribution functions. This forces an attacker to estimate many different moments; and also to estimate which moments (and how many) are used. If problem constraints limit the parameters of a distribution, one can often apply the constraints after the numbers are generated via rejection techniques, etc.

2. Restart the random number generator from time to time with new sub-keys. Restricting sequences to only 10,000 per sub-key limits parameter estimation to about 1% accuracy; shorter sequences limit parameter estimation accuracy even more.

3. Use random number generators whose probability distribution function contains multiple random parameters, e.g., a cubic spline with five random breakpoints. Then, even the accurate knowledge of several (many?) moments of the distribution function provides low accuracy knowledge about the distribution function itself. This aspect of defense is related to approximation theoretic attacks discussed next.

4.2.2 Approximation theoretic attacks

The disguise functions are chosen from spaces described in Section 2.2.1. Let F be the space of these functions, $u(x)$ be an original function, $f(x)$ be a disguise function so that $g(x) = u(x) + f(x)$ is observable by the agent. The agent may evaluate $g(x)$ arbitrarily and, in particular, the agent might (if F were known) determine the best approximation $g^*(x)$ to $g(x)$ from F. Then the difference $g^*(x) - g(x)$ equals $u^*(x) - u(x)$ where $u^*(x)$ is the best approximation to $u(x)$ from F. Thus $g^*(x) - g(x)$ is entirely due to $u(x)$ and gives some information about $u(x)$. There are three defenses against this attack:

1. Choose F to have very good approximating power so that the size of $g^*(x) - g(x)$ is small. For example, if $u(x)$ is an "ordinary" function, then including in F the cubic polynomials and the cubic splines with 5 or 10 breakpoints (in each variable) gives quite good approximation

power. One would not expect $\| g^*(x) - g(x) \| / \| u(x) \|$ to be more than a few percent. If $u(x)$ is not "ordinary" (e.g., is highly oscillatory, has boundary layers, has jumps or peaks) then care must be taken to include functions in F with similar features. Otherwise the agent could discover a great deal of information about $u(x)$ from $g(x)$.

2. Choose F to be a *one time random space* as described in Section 2.2.1. Since F itself is then unknown, the approximation $g^*(x)$ cannot be computed accurately and any estimates of it must have considerable uncertainty.

3. Approximate the function object $u(x)$ by a high accuracy, variable breakpoint piecewise polynomial. It is known [8] that this can be done efficiently (using a moderate number of breakpoints) and software exists to do this in low dimensions [9]. Then, one adds disguise functions with the same breakpoints and different values to the outsourced computation.

Underlying all these defenses is the fact that if F has good approximation power and moderate dimension, then it is very hard to obtain any accurate information from the disguised functions. This same effect is present in the defense against statistical attacks where the function to be hidden is the probability density function of the random number generator.

4.2.3 Symbolic code analysis

Many scientific computations involve a substantial amount of symbolic input, either mathematical expressions or high level programming language (Fortran, C, etc.) code. It is natural to pass this code along to the agent in the outsourcing and this can compromise the security. An expression COS(ANGLE2 * x - SHIFT) + BSPLINE(A, x) is very likely to be the original function (COS $\cdots$) plus the disguise (BSPLINE $\cdots$) and they can be distinguished no matter how much the BSPLINE function values behave like COS as a function. The symbolic information may be pure mathematics or machine language or anything in between. Outsourcing machine language is usually impractical and, in any case, provides minimal security. Fortran decompilers are able to reconstruct well over 90% of the original Fortran from machine language. Thus we must squarely address how to protect against symbolic code analysis attacks on high level language or mathematical expression of the computation. There are four general defenses against symbolic code analysis attacks:

1. Neuter the name information in the code. This means to delete all comments and to remove all information from variable names (e.g., name them A followed by their order (a number) of appearance in the code). This is an obvious, easy but important part of the defense.

2. Approximate the basic mathematical functions. The elementary built-in functions (sine, cosine, logarithm, absolute value, exponentiation, ...) of a language are implemented by library routines supplied by the compiler. There are many alternatives for these routines which can be

used (with neutered names or in-line code) in place of the standard names. One can also generate *one time elementary function approximations* for these functions using a combination of a few random parameters along with best piecewise polynomial, variable breakpoint approximations. In this way all the familiar elementary mathematical operators besides arithmetic can be eliminated from the code.

3. Apply symbolic transformations. Examples of such transformations are changes of coordinates, changes of basis functions or representations, and use of identities and expansions of unity. Changes of coordinates are very effective at disguise but can be expensive to implement. Consider the potential complications in changing coordinates in code with hundreds or thousands of lines. The other transformations are individually of moderate security value, but they can be used in almost unlimited combinations so that the combinatorial effects provide high security. For example, we transform the simple differential equation

$$y'' + x * \cos(x)y' + (x^2 + \log(x))y = 1 + x^2$$

using only a few simple symbolic transformations such as

$$\begin{gathered}
\cos^2 x - \sin^2 y = \cos(x+y)\cos(x-y) \\
\sec^2(x+y) - \tan^2(x+y) = 1 \\
(\tan x + \tan y)/\tan(x+y) + \tan x \tan y = 1 \\
1 + x = (1-x^2)/(x-x) \\
\sin(3(x+y)) = 3\sin(x+y) - 4\sin^3(x+y) \\
a^2 - ax + x^2 = (a^3 + x^3)/(a+x)
\end{gathered}$$

plus straight forward rearranging and renaming. The result is quite complicated (Greek letters are various constants that have been generated):

$$\begin{gathered}
(\beta\cos^2 x - \delta)y'' + x[\cos x/(\gamma\cos(x+1)) - \cos x \sin(x+1)\tan(x+1)]* \\
[\epsilon - \sin^2 x + \epsilon\sin(x+1) - \sin^2 x \sin(x+1)]y' \\
+[\beta(x\cos x)^2 - \eta(x + \log x) + \theta\cos x \log x^2]* \\
[\eta\sin x + \delta\tan x + [\chi\sin x + \mu\cos x + \nu)/\tan(x+2)]y \\
= (1+x^2)[\sin x + \eta\cos x]
\end{gathered}$$

If we further rename and/or re-implement some of the elementary functions and replace the variable names by the order in which the variable appears, this equation becomes

$$\begin{aligned}
& y''[x01 * x02(x) - x03] \\
+ \; & y'[x04 * x/(x05\cos(x+1) + \cos x * x06(x)\tan(x+1)]* \\
& [x07 - \sin^2 x - x08(x)\sin^2 x + x07\sin^2(x+1)] \\
+ \; & y[x01 * (x * x09(x))^2 - x10(x + \log x) + x11\cos x \log x^2]* \\
& [x12 * x13(x) + x14\tan x + (x15\sin x + x16\cos x + x17)] \\
= \; & \sin x + x18 * (1+x^2) * x09(x) + x19(x) + x10 * x^2\cos x
\end{aligned}$$

It is hard to say at this point how difficult it would be for a person to recover the original differential equation. It certainly would take a considerable effort, and this disguise uses rather elementary techniques.

4. Use reverse communication. This is a standard technique [28] to avoid passing source code into numerical computations and can be used to hide parts of the original computation. Instead of passing the code for $u(x)$ to the agent, one insists that the agent send x to the customer who evaluates $u(x)$ and returns the value to the agent; the agent never has access to the source code. If $u(x)$ is very simple, communication adds greatly to the cost of evaluating it. But it is more likely that $u(x)$ is complex (otherwise another disguise would be used) and that the extra communication cost is not important.

4.3 Disguise Strength Analysis

A systematic or general analysis of the security of disguises is not practical at this time for two reasons. First, the disguises are very problem dependent so each type of computation, perhaps even each individual computation, must be analyzed using different techniques. Second, disguises can be of essentially unlimited complexity. And this complexity is not of a linear nature (e.g., a function of the number of bits in the keys), but is a disorganized conglomeration of unrelated techniques. The strengths of disguises is derived from this complexity but it also tends to defeat the analysis of the resulting strength. Before one becomes uncomfortable with the strength of disguises, one must realize that a determined agent will be able to apply enormous resources in attacks. It may be feasible soon to analyze every code fragment to see if it computes a standard mathematical function and, if so, use this to simplify the disguise. It may be feasible to generate systematically disguise programs such as those in Section 2.3 (including choices for the numerical values of parameters) and see if they generate the outsourced computation. Further, it is possible that some disguise techniques that look strong now will become easy to defeat once they are studied in depth.

Nevertheless, we are optimistic that the complexity possible for disguises will allow this approach to withstand attacks for the foreseeable future. This optimism comes from the analysis of just three simple numerical computations given below. We note that it is harder to disguise simple computations than complicated ones. We also note that historically one of the biggest sources of weak security is that people become bored and lazy in applying security properly. Thus for the disguise of complex computations, it is imperative that tools such as the Atri system be available to minimize the effort of disguise and the risk of accidental holes in them.

To illustrate the strength possible for disguises and the analysis techniques one can use, we consider disguises of three simple, common scientific computations.

4.3.1 Matrix multiplication

We use the disguise program of Section 2.4 modified to (a) have controlled lengths of sequences from random number generators, and (b) to use better disguised random number generators. Let L be the maximum length for a sequence from a random number generator so that $M = \lceil m/L \rceil$ is the number of distinct generators needed. Let $G_{ONE}(A(i))$, $i = 1, 2, \ldots, M$ be one time random number generators as described in Section 2.2.1. Each has a vector $A(i)$ of 12 random parameters/seeds. The fourth line of the Section 2.2.1 program can be replaced by

```
Vector (1:m) α, β, γ
M = ⌈ M/L ⌉
For j=1 to M do
   Vector (1:12) A(j) using G_UNIF (k(j+1), [0,1])
   Vector (1:L) T1, T2, T3 using G_ONE A(j))
   α((j-1)L+i) = T1(i) for i = 1:L
   β((j-1)L+i) = T2(i) for i = 1:L
   γ((j-1)L+i) = T3(i) for i = 1:L
End do
```

This change in the program creates the non-zero vectors for the three matrices $P1, P2, P3$ used to disguise $M1$ and $M2$.

An agent attacking this disguise receives $X = P1 * M1 * P2^{-1}$ and $Y = P2 * M2 * P3^{-1}$. The only attack strategy possible is statistical. If the agent has no information about $M1$ and $M2$, there is no possibility to determine $M1$ and $M2$. Not even the average size of their entries can be estimated with any accuracy. Thus this disguise is completely secure. If it is possible that the agent has external information about $M1$ and $M2$, then one should take steps to disguise that type of information. For example, the class of problems one normally deals with might have some "typical" sparsity, sign or size patterns. These patterns can also be disguised, but we do not discuss this topic in this paper.

4.3.2 Numerical quadrature

We use the disguise program of Section 2.4 modified to have a second disguise function added. One potential weakness of the earlier disguise given is to an approximation theoretic attack. The $g(x)$ created is quite smooth and $f(x)$ might not be. So we replace the 7th line of the program as follows:

```
Cubic spline g1(x)=approximation (f,a,b, cubic spline, variable breaks, 1%)
Vector (1:L) x = breakpoints of g1(x)
Vector (1:L) R using G_UNIF (k(3), 0.8, 1.31)
Cubic spline g2(x) with g2(X(i)) = R(i)*g(X(i)) for i = 1:L
Cubic spline g3(x) with g3(P(i)) = V(i) for i = 1:7
Cubic spline g(x) = g3(x) + g2(x)
```

This change effectively modifies $f(x)$ randomly by about 25% and then adds another, smooth random function to it.

An agent attacking this disguise can try both approximation theoretic and code analysis strategies. The disguise so far only protects against approximation theoretic attacks. We see no way that the values of $h(x)$ sent to the agent can lead to accurate estimation of the result, say better than about 100% error. If we choose to protect against a code analysis attack using reverse communication, then the security is complete. If we choose to protect against a code analysis attack by replacing the $f(x)$ by a high accuracy approximation, then the security is again complete. To do this one just replaces 1% in the above code, by, say, *eps* and then defines

$$h(x) = g(x) + g2(x) + g3(x).$$

4.3.3 Differential equations

We assume that the differential equation is of the form

$$a_1(x)y'' + a_2(x)y' + a_3(x)y = a_4(x) \qquad y(a) = y_1, y(b) = y_2.$$

If reverse communication is used for the evaluation of the functions $a_i(x)$ then no symbolic information about them is available to the agent. There is no opportunity for statistical attack so the only feasible attack is approximation theoretic. Program (1.3.4) of Section 2.4 can be used to disguise the solution $y(x)$. If similar disguises are applied to the four functions $a_i(x)$, then neither the solution nor problem could be discovered by the agent and the disguise would be completely secure.

Since reverse communication can be expensive, we consider the alternative of using symbolic disguise such as in item 3 of Section 4.2.3. A symbolic attack is made "bottom up", that is, one attempts first to identify the elements (variables, constants, mathematical functions) of the symbolic expressions. There are two approaches here:

* *Program analysis.* For example, deep in a subroutine one might set $X137 = X09$ and later, in another subroutine set $X11 = X137$. This can be done either with constants or actual variables. Or, one can replace $\sin(X11)$ by $X12(X11)$ and implement $X12$ with machine code for sine; there are several standard alternatives for this implementation.

* *Value analysis.* For example, one can check to see if the value of $X9$, $1/X9$ or $(X9)^2$ is the same as any of the other constants in the program. Similarly, one can evaluate $X12(x)$ for a wide variety of x values and check to see if it is equal (or close) to $\sin(x)$, $\log(x)$, $\tan(x)$, x^3, etc.

The use of program substitution or modification can greatly increase the strength of a disguise. Indeed, it is clear that if the symbolic code is lengthy, then a modest amount of program disguise scattered about provides enormously strong security.

However, we focus our analysis on shorter symbolic forms such as the differential equation program which are completely defined by a single symbolic equation and "simple" functions. That is, how hard is it to break disguises based on introducing mathematical identities? The symbolic code attacker would first check:

1. *Which constants are related?*

2. *Which functions are standard mathematical functions?*

For question 1, suppose that there are N constants c_i in the equation and we ask if any c_i is $r(c_j)$ for a collection of relationship functions r. If the number of relationships r used is M, then this checking requires evaluating NM constants and then testing if any pairs are equal. The latter effort is work of order $NM \log M$ (using sorting) and values of $N = 20$, $M = 500$ are reasonable. Thus, the relationship of pairs of constants could be determined by sorting them or using about 10^5 operations. Such a computation is well within the capability of the agent, so we assume there is no security in binary constant disguises, i.e., if $X1 = 1.108$, $X2 = \sec(1.108)$ and $X3 = , (1.108)$ are present, then the agent will discover this fact.

Consider the next 3-way relationship such as

$$X03 = \alpha^2/\cos(2), \quad X15 = 1 + \alpha, \quad X12 = \alpha \tan(2)$$

which appear in the example at hand. In the notation used above, one considers if any c_i is $r_i(r_2(c_j)r_3(c_k))$. The number of constants computed is M^3N^2 (perhaps $5 \cdot 10^{10}$). The pairwise comparison requires work of order $M^3N^2 \log(MN)$ and the total comparison work is of the order of $M^3N^2 \log(MN)$. With $N = 20$ and $M = 500$, this is a computation of order about 10^{12} (including a factor of 10 or so for the function evaluations). This is a substantial, but not outrageous, computation today and will be less formidable with each passing year. We believe that disguises using 4-way relationships, e.g.,

$$X03 = \alpha^2\beta/\cos(2), X15 = 1 + \alpha\beta, X17 = \alpha \tan(2) H_{1/2}(\beta), X23 = \frac{\alpha - \beta}{\log 2}$$

will not be broken in the foreseeable future. We estimate the computational work to be the order of 10^{16} operations assuming $M = 500$, $N = 20$.

There is another consideration in using values to discover relationships among constants: there are only about 10^{10} different 32-bit numbers and only about 10^8 of them in any one broad range of sizes. Thus, the pairwise comparison approach must lead to many "accidental" matches, perhaps 100–1000 for 3-way relationships. Since each match is a starting point for a complex disguise attack, each of these accidental matches initiates a lengthy additional analysis for the attacker. This means that even 3-way relationships among constants provide a very high level of security.

Finally, another level of complexity can be obtained by splitting constants. Thus the 1.108 used above can be split three times by replacing $X1 = 1.108,\ X2 = \sec(1.108),\ X3 = ,\ (1.108)$ with

$$\begin{aligned} X1 &= .0634,\ X2 = 1.0446,\ X3 = X1 + X2 \\ X4 &= -.6925,\ X5 = 1.8005,\ X6 = \sec(X4 + X5) = 1/(\cos X4 \cos X5 - \sin X4 \sin X5) \\ X7 &= 1.000,\ X8 = 0.108,\ X9 = ,\ (X7 + X8) \end{aligned}$$

which makes all the constants unique. The task of checking all pairs of sums is not insurmountable but applying simple identities as follows makes the task of identifying the number 1.108 extremely complex

$$\begin{aligned} X6 &= 1/(\cos X4 \cos X5 - \sin X4 \sin X5) \\ X9 &= X8 *,\ (X8) \end{aligned}$$

For question 2, consider a function $F(x)$ and we can ask: Is F one of the standard mathematical functions? Is $F(x)$ related to another function $G(x)$? First, we note that $F(x)$ is parameterized by a random real number α, e.g., $\sin(\alpha x)$ or $u(\alpha, x)$ (the parabolic cylinder function), then it is hopeless to identify F by examining its values. It requires at least 100 times as much effort to compare two functions as it does two constants, and the parameter α increases the number of potential equalities by a factor of about 10^8. Thus, checking 2-way relationships between standard functions and parameterized functions is a computation of order 10^5 (what we had for constants) times 100 times 10^8 or about order 10^{15}.

Consider then the problem of deciding if $F(x)$ is one of or related to one of the standard mathematical functions $G_i(x)$. There are perhaps 25–50 standard functions ($\sqrt{}$, sin, log, cosh,...) and perhaps 5 times as many in simple relationships ($1/\log x,\ 1 + \log x,\ 2 * \log x,\ \log(1 + x)$, $\log(2 * x), \ldots$). Then there is the uncertainty in the range of arguments, so one might try five ranges (e.g., [.1,.2], [1.5,1.8], [12,13], [210,215], [–1,1]). Thus, $F(x)$ should be compared to a known $G_i(x)$ about 1000 times. If one comparison requires 1000 operations and there are 20 functions $F(x)$, then the total computing effort is the order of $20 * 1000 * 1000 = 2 * 10^7$ operations. This is easily done and there is no security in simple, single function disguises; a diligent agent will identify every one of them.

Consider the next 2-way function disguises, e.g.,

$$\begin{aligned} X1(x) &= \cos(x)\sin(x + 1), X2(x) = \cos(x)/\tan(x + 2), \\ X3(x) &= (1 + x^2)\sin(x), X4(x) = \cos(\sin(x + 1)). \end{aligned}$$

There are perhaps 100 simple functions (as above), so the number of pairs is 10^4, the number of simple combinations is five times this. Thus the work of $2 * 10^7$ operations seen above increases to the order of 10^{11} or 10^{12}. Thus the use of 2-way function disguise provides strong security and 3-way function disguises, e.g.,

$$\begin{aligned} X1(x) &= \cos(x)\sin(x+1)e^x, X2(x) = \cos(x)/\tan(x+2), X3(x) = e^{-x}(1+x^2)\sin x, \\ X4(x) &= x^2\tan(x+2)/\sin(x) \end{aligned}$$

provides complete security.

Consider now the situation with no disguises of functions or constants are made or, equivalently, when the agent is successful in breaking all these disguises. Question 3 is then:

3. *How well can one identify the identities and/or partitions of unity used and remove them from the problem?*

The process needed here is essentially the same as simplifying mathematical expressions by using manipulations and identities. This task is addressed some in symbolic computer systems and is known to be a formidable challenge even for polynomials and the elementary mathematical functions. We are aware of no thorough complexity analysis of mathematical simplification, but results relating the process to pattern matching suggest that the complexity is very high. The original differential equation can be represented as a tree and the process of using identities and simple manipulations is an expansion and rearrangement of this tree. It is plausible the expansion process involves (a) some functions with random parameters, (b) some functions that already appear in the expression, and (c) some completely unrelated common functions. Let us assume that for each identity used there are 4 simple manipulations, e.g., combining terms, splitting terms, rearrangements. The number of choices for (a) is enormous, but these functions might be clues to simplification by using the assumption they are unlikely to be part of the original problem. The number of choices of identities for (b) is moderate but still substantial, perhaps 6–10 identities for each original function. The number of choices for (c) is substantial, perhaps 100 distinct partitions of unity exist involving the common functions. Many more can be obtained by simple manipulation.

Assume now that the differential equation is simple, involving, say 4 common functions. Assume its tree representation has 10 nodes (4 of which cannot be modified being the = and differential operators). Further assume the disguise process uses N_1 identities involving random functions, $N2$ identities for each original function, and N_3 identities involving unrelated functions. Finally, assume that each identity is represented by a tree with 6 nodes. We obtain a rough estimate of the number of possible trees for the disguise as follows. The tree has $10 + N_1 + 4N_2 + N_3$ "fat nodes", a fat node represents an identity used. The number of choices for the N_1 identities is, say, 4 per function. The number of choices for the N_2 identities is, say, 8. The number of choices for the N_3 identities is 4 per new function introduced. The final tree has $10 + 6(N_1 + 4N_2 + N_3)$ nodes. The possible number of disguise trees is enormous, perhaps the order of $4^{N_1} \cdot 8^{N_2} \cdot 4^{N_3}$. If

$N_1 = N_2 = N_3 = 3$, there are about 10^6 possible trees, increasing 3 to 4 in these values gives more than 10^8 trees. And this estimate does not include the effect of the simple manipulations. The fact that the number of possibilities is so huge does not automatically mean the simplification is impossible. It does, however, suggest that the simplification is a very formidable computation, one that provides very strong security.

4.3.4 Domains of functions

Another view of symbolic code disguises and analysis comes from identifying *domains* of symbolic functions. A domain is a collection of symbolic elements that are related naturally by having some simple operations which transform one element of a domain into another. Examples include:

- *Numbers*: integer, real, complex. The operations are arithmetic.

- *Polynomials*: In one variable examples are x, $x^3, \ldots, 17 + 3x + 9x^4, \ldots$. In several variables examples are x, xy, $xy^2, \ldots, 9 + 3xy + 6.042\ x^3y^5z^4, \ldots$. The operations are again arithmetic, both numerical and polynomial (e.g., $(x+2)^2 = x^2 + 4xy + 4y^2$).

- *Algebraic*: Examples in one variable include $x^{1/2}$, $x^{7/2}, \ldots, 4 + x + 1.03x^{3/2}, \ldots$
  $x^{.073}$, $x^{.146}$, $x^{1.073}$, $2 + 7x^{.073} + 5.43x^{.292} + 18x^{2.146}$.

  These domains are closed under multiplication by the "base element", i.e., by constants, by x, by x or y, by x or particular fractional powers of x, respectively.

- *Trigonometric*: Examples in one variable include $\sin x$, $\cos^2 x$, $17 + 3.1 \sin x + 14.2 \cos^2(3x)$

  This domain is closed under multiplication by "base elements" $\sin x$ and $\cos x$, by constants and by identities involving sines, cosines, tangents, cotangents, secants.

There are many other domains involving higher transcendental functions which have a natural operations (particular to each domain) under which they are closed. Examples include hyperbolic functions – exponentials, logarithms, Gamma, Bessel, etc. The *span* of an expression is the union of the domains of the various elements (terms) in the expression. The span of a problem, equation, operator,... is the union of the spans of the expressions composing the problem, equation, operator,... . The *dimension* of the span is the number of domains in the span.

The principal techniques for disguising an expression within a domain are:

#1 Use standard identities to eliminate all the original elements, e.g.

$$4 + 3.79x^2 + 8x^4 \Rightarrow 1.729 + 2.271 + 3.74x^2 + .05x^2 + 6.204x^4 + 1.796x^4$$
$$\Rightarrow 1.729 + 2.271 + .05(x+1)(x-1) + .05 + 6.204x^4 + 1.796(x^2+1)(x^2-1) + 1.796$$
$$\Rightarrow 5.846 + .05(x+1)(x-1) + 3.74x^2 + 6.204x^4 + 1.796(x^2+1)(x+1)(x-1)$$
$$\Rightarrow 5.846 + 1.801(x+1)(x-1) + 3.74x^2 + 6.204x^4 + 1.796(x^2+1)(x+1)(x-1)$$

#2 Use partitions of unity (e.g., $\cos^2 x + \sin^2 x = 1$) to introduce new terms into the expression. Thus

$$4 + 3.79\cos x + 8\sin^2 x \Rightarrow 1.729 + 2.271 + 3.79/\sec\ x + 4(1 - \cos 2x)$$
$$\Rightarrow 1.729 + 2.271(\sin^2 x + \cos^2 x) + 3.79/\sec\ x + 4 + 4\sin^2 x - 4/\sec^2 x$$

#3 Use partitions of unity to introduce new domains into the expression. Thus

$$4.0 + 3.79x^2 \Rightarrow 1.729 + 2.271(\sin^2 x + \cos^2 x) + 3.79(x-1)(x+1) + 3.79$$
$$\Rightarrow 5.519 + 2.271\sin^2 x + 2.271(\cos^2 x - 1/\sec^2 x) + 3.79(x-1)(x+1)$$

We note that: (i) For constants there are no partitions of unity or other useful disguise identities within the domain. (ii) For polynomials there are no partitions of unity as 1 is a polynomial. However, factoring provides disguise opportunities. (iii) Partial fractions provide a way to easily transform to rational functions. From $1/(x-2)+1/(x-4)$ we have $1 = \frac{(2x-2)}{x^2-2x-8} + \frac{x+1}{4-x}$. Using such a partition of unity we can expand the domain of polynomials to the domain of rational functions.

It is clear that one can expand the complexity of an expression without limit. It is well known that simplifying complex expressions is very difficult. It is well known that one can replace common terms (e.g., $\sin x$, $\sqrt{x}$, $\log x$) by polynomial or piecewise polynomial approximations that are highly accurate and that there are multiple choices for doing this.

The basic scientific questions to be answered are:

Q1: How does one measure the strength of such disguises, e.g., what is the complexity of the symbolic expression simplification problem?

We believe that the answer to this question is not known but that experience shows that this complexity grows very rapidly with number of terms and with the dimension of the expression. Simplification is an intrinsic problem for symbolic algebra systems and they provide a number of tools to aid people in the process. Clever people using these tools often have difficulty finding suitably "simple" forms of expressions even when no direct effort has been made to disguise the symbolic structure.

Q2: Given that one starts with a particular expression (k terms, dimension d) and wants a disguise with K terms, how strong can the disguise be and how is the strongest disguise determined?

It is clear that Q2 is much harder than Q1, but it focuses the process of disguise clearly: One wants to limit the size of the disguised expression while making the disguise as strong as possible.

Chieh-Hsien Tiao has developed the following scheme for automatically generating disguises within a domain by combining random selections of partitions of unity (and other identities if available) – including partitioning constants. This creates an algorithm for making symbolic disguises.

Disguise Algorithm. Assume we are given a function $f(x)$ and a collection C of m different sets of partitions of unity. All functions involved lie in a particular domain and each set involves at most n functions, i.e., the collection is

$$C = \{\{f(i,j)(x)\}_{j=1}^{n}\}_{i=1}^{m}$$

Let q be the disguise depth and $d(x)$ the current disguised function. Then set $d(x) = f(x)$ and

For $k = 1$ to q do

(a) Randomly permute the positions of the elements of $d(x)$.

(b) For each term in $d(x)$ apply a partition of unity chosen randomly from C.

Make a final random permutation of the elements of $d(x)$.

This algorithm expands the number of terms in $d(x)$ exponentially in q, i.e., the final number of terms in $d(x)$ is of the order of $t * n^q$ where t is the number of terms in the original $f(x)$. The collection C and the random objects in steps (a) and (b) are the keys to this disguise. We illustrate this algorithm with a simple example. Assume we want to disguise $f(x) = x$, and we have 3 partitions of unity involving 2 functions each. $\{\{cos^2(x), sin^2(x)\}, \{sec^2(x), -tan^2(x)\}, \{x, 1-x\}\}$. Choose $q = 3$ and set $d(x) = x$.

1. (a) With only one term $d(x) = x$, no permutation is needed here.
   (b) Choose a random number in between 1 and 3, say 2.
   (c) Apply the second partition of unity on x to obtain

$$d(x) = x * sec^2(x) - x * tan^2(x).$$

2. (a) Choose a random permutation of {1, 2}, say {2, 1}.
   (b) Choose a random number for each term, say {2, 3}.
   (c) Apply 2a and 2b to obtain

$$\begin{aligned} d(x) &= -x * tan^2(x) + x * sec^2(x) \\ &= -x * tan^2(x) * sec^2(x) + x * tan^4(x) + x^2 * sec^2(x) + x * (1-x) * sec^2(x). \end{aligned}$$

3. (a) Choose a permutation of {1, 2, 3, 4}, say {3, 4, 1, 2}.
   (b) Choose a random number for each term, say {3, 2, 3, 1}.

(c) Apply 3a and 3b to obtain

$$\begin{aligned} d(x) &= x^2 * sec^2(x) + x * (1-x) * sec^2(x) - x * tan^2(x) * sec^2(x) + x * tan^4(x) \\ &= x^3 * sec^2(x) + x^2 * (1-x) * sec^2(x) + x * (1-x) * sec^4(x) \\ &\quad -x * (1-x) * sec^2(x) * tan^2(x) - x^3 * tan^2(x) * sec^2(x) - x^2 * (1-x) * tan^2(x) * sec^2(x) \\ &\quad +x * tan^4(x) * cos^2(x) + x * tan^4(x) * sin^2(x). \end{aligned}$$

4. Choose a final random permutation, say {1, 3, 7, 6, 2, 5, 4, 8} to obtain

$$\begin{aligned} f(x) = x &= x^3 * sec^2(x) + x * (1-x) * sec^4(x) + \\ &\quad x * tan^4(x) * cos^2(x) - x^2 * (1-x) * tan^2(x) * sec^2(x) + \\ &\quad x^2 * (1-x) * sec^2(x) - x^3 * tan^2(x) * sec^2(x) - \\ &\quad x * (1-x) * sec^2(x) * tan^2(x) + x * tan^4(x) * sin^2(x) \\ &= d(x). \end{aligned}$$

5. The key to retrieve $f(x)$ from $d(x)$ is

   {3,{1},{2},{2,1},{2,3},{3,4,1,2},{3,2,3,1},{1,3,7,6,2,5,4,8}}.

Questions that arise naturally include:

Q3: Under what circumstances is it advantageous to increase the span of the disguised expression?

Q4: Is the best choice (i.e., providing strongest disguise with a given number of terms) to reduce everything to piecewise polynomials (i.e., include logic and constants). Note that one can disguise the common routines for $\sin x$, x^a, a^x, $\log x$, etc., so their well-known constants are not used directly.

Q5: Can one analyze the simplest disguise problem, namely that for constants. Given C_1, C_2, $C_3, \ldots, C_k$ and C^*, is there a way to write $C^* = \alpha_1 C_1 + \alpha_2 C_2 + \ldots + \alpha_k C_k$ where the α_k are "small" integers, i.e., $-M \leq \alpha_i \leq M$. If the complexity of determining the α_k can be found and is relatively "difficult", then this would be good indication that disguises based on symbolic substitutions are hard to break. Exhaustive search requires $(2M)^k$ trials, so $M = 20$ and $k = 6$ requires 10^{10} checking operations.

4.3.5 Code disguises

There is another class of disguises which can further add to the security of symbolic disguises. These are not based on mathematical methods but rather on programming language (code) transformations. These techniques are called *code obfuscations* and are primarily intended to provide "watermarks" for codes, i.e., to change the code in such a way that it computes the same thing but is different from the original code. The changes made must be both unique to each version

and difficult to reverse engineer. In other words, the original code must be difficult to obtain from the obfuscated code. A review and taxonomy of these techniques are given in [7]. These obfuscations can be applied to the actual computer programs for the symbolic portions of an outsourced computation and further increase the security of the symbolic content.

In summary, our analysis of disguise strength shows that:

- Reverse communication provides complete security.
- Disguises of constants provide strong to complete security.
- Disguises of functions with random parameters provide complete security.
- The difficulty of mathematical simplification (removal of identities) provides very strong, probably complete, security.
- The use of code obfuscation techniques provides another level of security for symbolic disguises.

These conclusions depend, of course, on the effort put into the disguise. Only moderate effort is needed to achieve these conclusions, substantial efforts will provide complete security. The combined effect of these analyses suggests that even modest effort may provide complete security.

5 COST ANALYSIS

We have identified four components of the cost of disguise and these are analyzed here.

5.1 Computational Cost for the Customer

A review of the disguise techniques proposed shows that all of them are affordable in the sense that the computation required of the customer is proportional to the size of the problem data. For some computations, e.g., solving partial differential equations or optimization, the cost can be several, even 5 or 10, times the size of the problem data. However, these are computations where the solution cost is not at all or very weakly related to the problem data, i.e., very large computations are defined by small problem statements.

The principal cost of the customer is, in fact, not computational, but in dealing with the complex technology of making good disguises. The envisaged Atri problem solving environment is intended to minimize this cost and it is plausible that an average scientist or engineer can quickly learn to create disguises that provide complete security.

5.2 Computational Cost for the Agent

A review of the disguise techniques shows that some disguises might dramatically increase the computational cost for the agent. Since the customer is probably paying this cost, this is our concern. The effects are problem dependent and in many (most?) cases the disguise has a small effect on the agent's computational cost.

None of the disguise techniques proposed change the basic type of the computation. Thus, numerical quadrature is not reposed as an ordinary differential equations problem. Nor do we propose to disguise a linear system of equations as a nonlinear system. Such disguises might provide high security, but they are not necessary. But the nature (and cost) of a computation is sometimes affected greatly by fairly small perturbations in the problem.

5.2.1 Preservation of problem structure

We define *problem structure* to include all those aspects of a problem which affect the applicability of software or problem solving techniques or which strongly affect the cost of using a technique. Examples of structure that are important to preserve in disguises are:

1. *Sparsity.* In optimization and partial differential equation problems the sparsity pattern of an array can reflect important information about the problem. One can usually change the sparsity by applying some "solution preprocessing" such as augmenting or reducing the number of variables. However, these changes, especially augmentation, might have a substantial negative impact on the efficiency of solution algorithms. Just how much impact appears to be dependent on the specific problem and its disguise.

2. *Function smoothness and qualitative behavior.* Function smoothness and other qualitative behaviors (e.g., rapid oscillations or boundary layers) could provide important information about a problem. These behaviors also have an important impact on the efficiency of many algorithms, even rendering some of them ineffective. Thus the disguise should mimic these behaviors whenever possible (the third item in Section 4.2.2 presents a way to do this). Further the disguise should not introduce any such behavior if this could have a substantial negative impact on algorithm efficiency or applicability.

3. *Geometric simplicity.* Geometry is sometimes a very important part of the critical information about a computation so it is important to disguise it using domain augmentation, splitting or transformations. All of these can introduce problem features leading to increased computational cost. For example, introducing a reentrant corner into the domain of a partial differential equation problem can make the computation more expensive by orders of magnitude. Disguise techniques must be careful to avoid such changes in the geometry.

4. *Problem simplicity.* Some simple problems allow exceptionally efficient solution techniques, e.g.,

$$u_{xx} + u_{yy} + 2u = f(x, y) \qquad (x, y) \text{ in rectangle}$$

can be solved by FFT methods. One can disguise the solution or map the rectangle into a similar one and still use the FFT method; other disguise techniques are likely to make the FFT method unusable and to require a substantially more expensive computation for the solution.

Since problem structure can be a very important attribute of a computation from the security standpoint, care must be taken to disguise this structure as well. We see from the examples above that the disguise of structure can have a large negative impact on the cost of the computation. In most cases the structure can be disguised without this impact if care is taken. The case of extremely simple computations seems to present the hardest challenge to disguise well while maintaining efficient computation. In some of these cases it may be necessary to make the computation more expensive. Most of these computations are "cheap" due to their simplicity, so perhaps the extra cost for security will be tolerable.

5.2.2 Control of accuracy and stability

A computation is unstable if a small change in it makes a large change in its result. Care must be taken so that disguises do not introduce instability. There are two typical cases to consider:

1. *The original computation is stable.* This means that moderate perturbations can be made safely, but how does one measure "moderate"? There is no general technique to estimate quickly and cheaply the stability of computations. Since the disguises involve "random" or "unpredictable" changes, it is highly unlikely that a disguise introduces instability. But highly unlikely does not mean never. For some computations, stability can be estimated well, and with low cost, during the computations. Agents providing outsourcing service should provide stability estimates for such computations. For other computations the customer may need to estimate and control the stability of the computation.

2. *The original computation is not very stable.* This means that moderate, even minor, perturbations are unsafe if made in the "wrong direction". Even if the disguise perturbs the computation in a "safe direction" (making it more stable), the consequence is that the inversion process of the disguise becomes unstable. The agent would correctly state that the computation is stable. If the customer is not aware of the potential instability in the disguise inversion, a complete loss of accuracy could occur with no warning.

Of course, we should always hold the customer responsible for understanding the stability properties of the original problem. It is the "duty" (at least the goal) of the numerical algorithms to preserve whatever stability that exists in the computation presented to them. They should also report any large instability that they sense. Since the disguise and its inversion is the hands of the customer, the agent has no additional responsibility. One can envisage that the Atri system could provide the user with tools and aids to assist in the evaluation of the effects of disguises on stability.

5.3 Network Costs

The disguises can increase the network traffic costs by increasing (1) the number of data objects to be transmitted, and (2) the bulk of the data objects. A review of the disguises proposed in this paper shows that the number of objects is rarely changed much from the original computation. However, the bulk of the individual objects might change significantly. Coordinate changes and the use of identities can change functions from expressions with 5–10 characters to ones with many dozens of characters. This increase is unlikely to be important as in most computations with symbolic function data, the size of the data is very small compared to the size of the computation and thus the network cost of the input data is negligible. The disguise techniques used for integers at the end of Section 1.2 increased their length from 8 bits to 9 bits; similar schemes could increase their length from 1 byte to 2 bytes. This might double the network costs for some computations. The network cost in returning the result is less likely to be increased, but it can be for some computations.

In summary, we conclude that:

- Customer costs for disguise are reasonable and linear in the size of the computation data. The principal cost is in the "intellectual" effort needed to make good disguises.
- Agent costs have a minimal increase unless the customer changes the problem structure. The control of problem structure increases the intellectual effort of the customer. There might be especially simple computations where good disguise requires changing the problem structure.
- Network cost increases vary from none to modest, rarely will this cost be doubled.

References

1. M. Abadi, J. Feigenbaum, and J. Killian. On hiding information from an oracle, *J. of Computer and System Sciences*, **39**, (1989), 21–50.
2. M. Abramowitz and I.A. Stegun. *Handbook of Mathematical Functions*, Appl. Math. Series 55, Nat. Bur. Stnds., U.S. Govt. Printing Office, (1964).

3. M.J. Atallah, K.N. Pantazopoulos, and E.H. Spafford. Secure outsourcing of some computations, Department of Computer Sciences, CSD-TR-96-074, Purdue University, (1996).

4. P. Beguin and J-J. Quisquater. Fast server-aided RSA signatures secure against active attacks, *CRYPTO*, (1995), 57–69.

5. R.F. Boisvert, S.E. Howe, and D.K. Kahaner. GAMS: A framework for the management of scientific software, *ACM Trans. Math. Software*, **11**, (1995), 313–355. `http://gams.nist.gov/`

6. M. Boninsegna and M. Rossi. Similarity measures in computer vision, *Pattern Recognition Letters*, **15**, (1994), 1255–1260.

7. C. Collberg, C. Thomborson and D. Low. *A taxonomy of obfuscating transformations*, Tech. Rpt. 148, Department Computer Science, University of Auckland, (1988).

8. C. deBoor and J.R. Rice. An adaptive algorithm for multivariate approximation giving optimal convergence rates, *J. Approx. Theory*, **25**, (1979), 337–359.

9. C. deBoor. *A Practical Guide to Splines*, SIAM Publications, (1978).

10. B. Dole, S. Lodin, and S.E. Spafford. Misplaced trust: Kerberos 4 session keys. In *Proceedings of the 4th Symposium on Network and Distributed System Security*, IEEE Press, (1997), 60–71.

11. T. Drashansky, A. Joshi, and J.R. Rice. SciAgents – An agent based environment for distributed, cooperative scientific computing, *Proc. 7th Intl. Conf. Tools with Artificial Intel.*, IEEE Press, (1995), 452–459.

12. T. Drashansky, S. Weerawarana, A. Joshi, R. Weerasinghe, and E.N. Houstis. Software architecture of ubiquitous scientific computing environments for mobile platforms, Department of Computer Sciences, CSD-TR-95-032, (1995).

13. D.E. Eastlake, S.D. Crocker, and J.I. Schiller. *RFC-1750 Randomness Recommendations for Security*, Network Working Group, (1994).

14. E. Gallopoulos, E.N. Houstis, and J.R. Rice. Computer as thinker/doer: Problem solving environments for computational science, *IEEE Comp. Sci. Eng.*, **1**, (1994), 11–23.

15. S. Garfinkel and E.H. Spafford. *Practical UNIX & Internet Security*, O'Reilley & Associates, Second Edition, (1996).

16. R.C. Gonzalez and R.E. Woods. *Digital Image Processing*, Addison-Wesley, Reading, MA, (1992).

17. 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), 215–228.

18. A.K. Jain. *Fundamentals of Digital Image Processing.* Prentice Hall, Englewood Cliffs, N.J., (1989).

19. S-I. Kawamura and A. Shimbo. Fast server-aided secret computation protocols for modular exponentiation. In *Proceedings of IEEE J. on Selected Areas in Communications*, **11**, (1993), 778–784.

20. D.E. Knuth. *The Art of Computer Programming, Volume 2*, Addison Wesley, Second Edition, (1981).

21. C-S. Laih and S-M. Yen. Secure addition sequence and its application on the server-aided secret computation protocols, *AUSCRYPT*, (1992), 219–230.

22. C-H. Lim and P.J. Lee. Security and performance of server-aided RSA computation protocols, *CRYPTO*, (1995), 70–83.

23. T. Matsumoto, K. Kato, and H. Imai. Speeding up secret computations with insecure auxiliary devices, *CRYPTO*, (1988), 497–506.

24. B. Pfitzmann and M. Waidner. Attacks on protocols for server-aided RSA computation, *EUROCRYPT*, (1992), 153–162.

25. J-J. Quisquater and M. de Soete. Speeding up smart card RSA computations with insecure co-processors, *Smart Card 2000*, North Holland, (1991), 191–197.

26. C.J. Ribbens. A fast adaptive grid scheme for elliptic partial differential equations, *ACM Trans. Math. Softw.*, **15**, (1989), 179–197.

27. C.J. Ribbens. Parallelization of adaptive grid domain mappings. In *Parallel Processing for Scientific Computing*, (G. Rodrique, ed.), SIAM, Philadelphia, (1989), 196–200.

28. J.R. Rice. *Numerical Methods, Software, and Analysis*, Second Edition, Academic Press, (1993), Section 7.6.D.

29. R.L. Rivest, L. Adleman, and M.L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, (R.D. DeMillo, ed.), Academic Press, (1978), 169–177.

30. B. Schneider. *Applied Cryptography*, Wiley, Second Edition, (1996).

31. A. Shimbo and S. Kawamura. Factorization attacks on certain server-aided computation protocols for the RSA secret transformation, *Electronic Letters*, **26**, (1990), 1387–1388.

32. D.R. Stinson. *Cryptography: Theory and Practice*, CRC Press, Boca Raton, FL, (1995).

33. G.J. Simmons, ed., *Contemporary Cryptology: The Science of Information Integrity*, IEEE Press, (1992).