

Secure Execution of Computations in Untrusted Hosts

S. H. K. Narayanan¹, M. T. Kandemir¹, and R. R. Brooks², I. Kolcu³

¹Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA

²Department of Electrical and Computer Engineering
Clemson University, Clemson, SC 29634, USA

³Computation Department, UMIST, Manchester, M60 1QD, UK

Abstract. Proliferation of distributed computing platforms, in both small and large scales, and mobile applications makes it important to protect remote hosts (servers) from mobile applications and mobile applications from remote hosts. This paper proposes and evaluates a solution to the latter problem for applications based on linear computations that involve scalar as well as array arithmetic. We demonstrate that, for certain classes of applications, it is possible to use an optimizing compiler to automatically transform code structure and data layout so that an application can safely be executed on an untrusted remote host without being reverse engineered.

©Springer-Verlag 2006. This is the author's version of the work. The publication is available at the Lecture Notes in Computer Science (LNCS) webpage at http://dx.doi.org/10.1007/11767077_9, or via the main LNCS webpage, <http://www.springer.de/comp/lncs/index.html>.

This work is supported in part by NSF Career Award 0093082 and a grant from the GSRC.

Secure Execution of Computations in Untrusted Hosts*

S.H.K. Narayanan¹, M.T. Kandemir¹, R.R. Brooks², and I. Kolcu³

¹ Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA
{snarayan, kandemir}@cse.psu.edu

² Department of Electrical and Computer Engineering
Clemson University, Clemson, SC 29634, USA
rrb@acm.org

³ Computation Department, UMIST, Manchester, M60 1QD, UK
ikolcu@umist.ac.uk

Abstract. Proliferation of distributed computing platforms, in both small and large scales, and mobile applications makes it important to protect remote hosts (servers) from mobile applications and mobile applications from remote hosts. This paper proposes and evaluates a solution to the latter problem for applications based on linear computations that involve scalar as well as array arithmetic. We demonstrate that, for certain classes of applications, it is possible to use an optimizing compiler to automatically transform code structure and data layout so that an application can safely be executed on an untrusted remote host without being reverse engineered.

1 Introduction

Mobile code technology allows programs to move from node to node on a network. Java is probably the best-known mobile code implementation. Applets can be downloaded and executed locally. Remote Method Invocation (RMI) allows applets registered with a service to be executed on a remote node. The use of a standardized language allows virtual machines running on different processors to execute the same intermediate code.

The software update and patch systems for both Microsoft and the Linux community are built on mobile code infrastructures. In current security research, the goal is to secure individual workstations and restrict program execution to a set of trusted programs. This paper looks at the largely ignored problem of protecting programs running on remote untrusted systems, and proposes automated compiler help to ensure secure execution of programs. Several important applications exist for this technology. A driving force for computer interoperability and sharing of software is business-to-business e-commerce. There are real needs to retrieve information from remote suppliers or clients. On the other hand, there

* This work is supported in part by NSF Career Award 0093082 and a grant from the GSRC.

are also real needs to guard corporate intellectual property. The approaches we present are an initial step towards allowing interoperability without risking reverse engineering and the loss of intellectual capital. Legal and accounting applications of the approach for auditing and monitoring systems is also foreseeable. Finally, remote sensor-based processing is another potential application domain because in many cases data remains with the sensors that collect them or our application needs data that is available only in a particular region covered by (untrusted) sensor nodes. Consequently, we need to be able to execute our application remotely and we do not know whether the sensor nodes are reliable. In all these scenarios, remote secure execution in an untrusted environment is a critical issue, and this paper proposes and evaluates a compiler-driven approach to this problem and presents experimental evidence demonstrating its applicability.

Mobile code security is not a new topic. Generally, the research in this area can be broken down into three parts. Protecting the host from malicious code, protecting the code from malicious hosts and finally, preventing man-in-the-middle type attacks which result in mobile code or generated results being leaked. In the mobile code protection domain, which is what this paper targets, prior works involve sending the original code and somehow ensuring that the right results are brought back, i.e. they concentrate on the results generated by mobile code. For example, [3] deals with spoofing and [14] addresses the problem of a malicious remote host not generating the right results. Further, [9] gives a technique to prevent one host from changing the results generated by another host.

However, the prior works assume that the original code can be shared with all hosts, irrespective of whether they are trustworthy. Our approach allows the owner of the mobile code, to protect the code itself from being revealed and hence helps to preserve intellectual capital. The proposed mechanism allows the owner to send a code that is different from the original code and still get back the results that he/she wants. Hence, this approach ensures that if the owner does not want to share the code, the confidentiality of the original code is never lost. The proposed mechanism cannot individually solve all the issues involved in mobile code security (such as spoofing, correct code execution, obtaining untampered results), but when used in conjunction with existing techniques it ensures that the code as well as the generated results are secure.

This rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents a high-level view of the proposed approach, and its mathematical details are presented in Section 4. Section 5 explains how computation matrices are formed and how our approach handles affine computations. Section 6 discusses the selection of transformation matrices for correct secure execution. Section 7 discusses how our approach is extended when we have multiple servers. Section 8 provides an example and Section 9 discusses our experimental results. Section 10 concludes the paper.

2 Related Work

The work presented in this paper is related to many efforts in distributed computing, agent-based computing, remote procedure invocation, code security/safety,

and code obfuscation domains. In this section, we only focus on the secure execution of functions on untrusted hosts. This has been studied as a more general problem of confidentiality of execution in efforts such as [1, 4, 12, 13]. Most of these efforts focus on the circuit computation model which is not very well suited for general, large-scale mobile code. Sander and Tschudin [8, 10] defined a function hiding scheme, and focused on non-interactive protocols. In their framework, the privacy of a function is assured by an encrypting transformation on that function. Integrity of evaluation is the ability of the circuit owner to verify the correctness of the execution of his/her circuit. This problem has been widely studied in the view of reliability but not from the view of a malicious server. The proof-based techniques [14] suggested that the untrusted host has to forward a proof of correctness of execution together with the result.

Perhaps the most relevant prior work to the one presented in this paper is [6, 7] in which a function is encrypted using error coding and sent to the untrusted host which provides the clear-text input. The enciphered output generated by the host is then sent back to the original host, where it is decrypted and the result is verified. The authors advocate the employment of the tamper proof hardware (TPH) as a necessary mechanism to store and provide the control flow between the numerous functions that make up a program. Control flow is located on the TPH and is supplied to the untrusted host. The main difference between these studies and the work presented in this paper is that we target general scalar and array based computations not circuit-specific expressions. Consequently, the code and data transformations used by our approach are different from those employed in prior studies such as [6, 7], and are directed by an optimizing compiler. Further, our approach deals with the case with multiple untrusted hosts as well.

3 High-Level View

This section presents an overview of the proposed mechanism. First, the mechanism used for scalar codes is presented. Following this, array based codes are discussed. In both the cases, we use the term “client” to refer to the owner of the application code to be executed, and the term “server” to denote the remote untrusted host (node) that will execute this application.

3.1 Scalar Codes

The high-level view of our approach for linear scalar computations is illustrated in Figure 1(a). On the client side, we have a computation represented, in a compact form, by computation matrix C . We want to execute this computation using input data represented by vector \mathbf{I} , and generate an output, represented by vector \mathbf{O} . That is, the original computation that we want to perform (as the client) can be expressed in mathematical terms as:

$$\mathbf{O} = C\mathbf{I}. \tag{1}$$

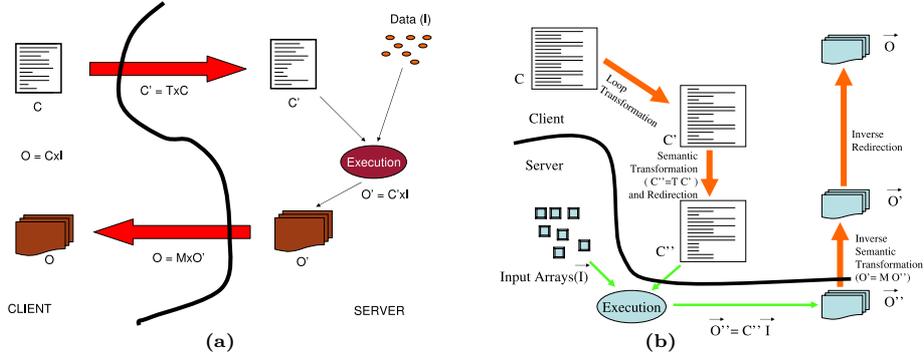


Fig. 1. High-level view of secure code execution in an untrusted server for (a) Scalar codes and (b) Array based codes. The thick curve represents the boundary between the client and the server. Both cases calculate $O = CI$.

The problem is that the client does not have input I and this input data cannot be transmitted to the client.¹ Consequently, the computation must be performed at the server side. The client transforms C to C' , and sends this transformed code to the server. The server in turn executes this transformed code represented by matrix C' using input I , computes an output (O'), and sends it to the client (note that $O' \neq O$). Since only the client knows the relationship between C and C' , it also knows how to obtain the originally required output O from O' , and it uses an appropriate data transformation for this purpose.

3.2 Array-Based Codes

The high-level view of our approach for array based computations is illustrated in Figure 1(b). C is transformed by a loop transformation into a code C' , in which the order, by which the elements of an array are accessed, within the loop is changed. In the next step, C' undergoes a semantic transformation to form a new code in which the meaning of the code itself is changed. In order to prevent the untrusted host from gleaning the locations of the arrays to which computed values are written (on the left-hand-side of the expressions), the left-hand-side arrays are replaced by different array expressions. This step is referred to as redirection or data remapping. C'' is applied to the input I by the server to generate the output O'' . This output is sent back to the client, which obtains O' from it by applying the inverse of the semantic transformation used earlier. Following this, we use the inverse of the array redirection used earlier, which eventually gives us O , the desired output (i.e., $O = CI$).

¹ This can be due to two potential reasons: either the data is not physically movable as in the case in a remote sensor processing environment, or the server is not willing to share data, due to security concerns.

4 Mathematical Details

This section provides the mathematical details of our proposed method. For the purpose of clarity, the determination of computation and code/data transformation matrices is dealt with separately in Section 5 and Section 6, respectively.

4.1 Scalar Codes

The main restriction that we have regarding the computation to be performed is that it should be a *linear function* of \mathbf{I} , and as a result, can be represented by a matrix (C), as is well-known from the linear algebra theory. Note that in the execution scenario summarized above, the client performs two transformations:

- **Code Transformation:** This is performed to obtain C' from C . As both C' and C are linear and expressed using matrices, we can use a linear transformation matrix T to denote the transformation performed. Consequently :

$$C' = TC. \quad (2)$$

- **Data Transformation:** This is performed to obtaining \mathbf{O} from \mathbf{O}' , and can also be represented using a matrix (M):

$$\mathbf{O} = M\mathbf{O}'. \quad (3)$$

4.2 Array-Based Codes

The client performs the following series of transformations on the computation matrix C :

- **Loop Transformation:** In optimizing compiler theory, loop transformations are used to reorder the points in loop iteration spaces [11]. Here it is used to obtain C' from C . Each execution of a loop body is represented by an iteration vector \mathbf{i} . An array reference accessed in a nest is represented as:

$$L\mathbf{i} + \mathbf{o}, \quad (4)$$

where L is referred to as the *access matrix* and \mathbf{o} is referred to as the *offset vector*. A linear loop transformation can be represented using a transformation matrix T_L . Upon application of this transformation, the iteration vector \mathbf{i} is mapped to $\mathbf{i}' = T_L\mathbf{i}$. As a consequence, the new subscript function is given by the following expression:

$$LT_L^{-1}\mathbf{i}' + \mathbf{o}. \quad (5)$$

This means that the new (transformed) access matrix is $L' = LT_L^{-1}$. The loop transformation does not affect the offset vector \mathbf{o} . The loop bounds are, however, affected by this transformation. The loop bounds of the transformed iteration space can be computed – in the most general case – using techniques such as Fourier-Motzkin elimination [11].

- **Semantic Transformation:** This is performed to obtain C'' from C' . Since both C'' and C' are linear and expressed using matrices, a transformation matrix T is used to denote the transformation being applied.

$$C'' = TC'. \quad (6)$$

It needs to be emphasized that T is entirely *different* from T_L . While it is true that both of them are applied to the loop nest, T_L re-orders loop iterations, whereas T modifies the loop body. Another important difference is that while T_L is a semantic-preserving transformation, T changes the meaning of the computation performed within the loop body.

- **Redirection:** This is a data space transformation performed to hide the memory locations in the client to which the results of computation are being stored. This also makes manipulating the results of the computation easier as the dimensions of the result matrices will be the same. Let $L\mathbf{i} + \mathbf{o}$ be an array reference after the loop and semantic code transformations have been applied. Our goal is to apply a data (memory layout) transformation such that the access matrix and the offset vector are mapped to desired forms. While any data transformation that changes L and \mathbf{o} is acceptable, the one adopted in this work transforms the access matrix to the identity matrix and the offset vector to the zero vector if it is possible to do so (if not, we use an arbitrary but legal transformation). We represent a data transformation using a pair (S, \mathbf{s}) . In this pair, S is termed as the data transformation matrix and is $m \times m$ for an m -dimensional array. \mathbf{s} is called the shift vector and has m entries. Redirection transforms, the reference $L\mathbf{i} + \mathbf{o}$ to:

$$SL\mathbf{i} + S\mathbf{o} + \mathbf{s}.$$

We want SL to be the identity matrix (I_D) and $S\mathbf{o} + \mathbf{s}$ to be the zero vector. We solve this system of equations as follows. First, from $SL = I_D$ we solve for S . After that, we substitute this S in the second equation ($S\mathbf{o} + \mathbf{s} = \mathbf{0}$), and determine \mathbf{s} .

- **Inverse Semantic Transformation:** This is performed to obtain \mathbf{O}' from \mathbf{O}'' , and can also be represented using a matrix (M).

$$\mathbf{O}' = M\mathbf{O}''. \quad (7)$$

Note that, at this point, we apply the inverse of the redirection used earlier, and do not apply the inverse of the loop transformation used earlier. This is because, C and C'' are semantically equivalent and the outputs generated by them are equivalent (in the context of this paper).

- **Inverse Redirection:** The purpose of this transformation is to obtain the original memory locations of the output elements computed. Recall that the redirection transforms reference $L\mathbf{i} + \mathbf{o}$ to $SL\mathbf{i} + S\mathbf{o} + \mathbf{s}$. To obtain the original reference from this, we use the data transformation (Y, \mathbf{y}) . This gives us:

$$Y\{SL\mathbf{i} + S\mathbf{o} + \mathbf{s}\} + \mathbf{y},$$

which expands to

$$YSL\mathbf{i} + YS\mathbf{o} + Y\mathbf{s} + \mathbf{y}.$$

Since we want $YSL = L$ and $YS\mathbf{o} + Y\mathbf{s} + \mathbf{y} = \mathbf{o}$, we determine Y and \mathbf{y} as:

$$Y = S^{-1} \quad \text{and} \quad \mathbf{y} = -S^{-1}\mathbf{s}.$$

5 Determining Computation Matrix and Handling Affine Programs

An important problem is to determine matrix C , given a code fragment. Recall that this matrix captures the relationship between \mathbf{I} and \mathbf{O} . Let us assume for now that the program variables in \mathbf{I} and \mathbf{O} are disjoint; that is, the two vectors have no common variables. In this case, it is easy to convert a linear code fragment to C . As an example, consider the code fragment below:

```
a := d+e+f;
b := g-2e;
c := 3f+4d;
```

For this fragment, the input variables are e , d , f and g , and the output variables are a , b and c . Consequently, we can express \mathbf{I} , \mathbf{O} , and C as:

$$\mathbf{I} = \begin{pmatrix} d \\ e \\ f \\ g \end{pmatrix}; \quad \mathbf{O} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}; \quad \text{and} \quad C = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & -2 & 0 & 1 \\ 4 & 0 & 3 & 0 \end{pmatrix}.$$

However, the problem becomes more difficult if there are dependencies in the code. Our solution to this problem is to use multiple C "sub-fragments". As an example, let us consider the following code fragment:

```
a := d-5c+2g;
b := e+f;
c := g+4d;
h := 3e-4d;
```

Since variable c is used both on the right-hand-side of the first statement and on the left-hand-side of the third statement, we cannot directly apply the method used in the previous case. However, we can (logically) divide the statements into two groups. The first group contains the first two statements, whereas the second group contains the remaining two statements. Note that, the only data dependence in the code (an anti-dependence in this case) goes from the first group to the second group; that is, the original code fragment is divided over the data dependence. After this division, we can assign a separate C matrix to each sub-fragment. In this example, we have:

$$\mathbf{I}_1 = \begin{pmatrix} c \\ d \\ e \\ f \\ g \end{pmatrix}; \quad \mathbf{O}_1 = \begin{pmatrix} a \\ b \end{pmatrix}; \quad C_1 = \begin{pmatrix} -5 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{I}_2 = \begin{pmatrix} d \\ e \\ g \end{pmatrix}; \quad \mathbf{O}_2 = \begin{pmatrix} c \\ h \end{pmatrix}; \quad C_2 = \begin{pmatrix} 4 & 0 & 1 \\ -4 & 3 & 0 \end{pmatrix}.$$

The C_1 and C_2 matrices are then used to represent the computation performed by the two sub-fragments.

So far, our formulation has focused on handling linear computations. We now discuss how our approach can be extended to affine computations. These computations are different from linear computations in that the relationship between \mathbf{I} and \mathbf{O} is expressed as:

$$\mathbf{O} = C\mathbf{I} + \mathbf{e}, \tag{8}$$

as opposed to $\mathbf{O} = C\mathbf{I}$, used in the linear case. Here, \mathbf{c} is a constant vector, i.e., it contains the constant terms used in the assignment statements that form the computation. Let us define our loop transformation in this case as follows:

$$C' = TC + \mathbf{t}, \quad (9)$$

where \mathbf{t} is a constant vector, whose entries are to be determined (along with those of T). In this case, the server computes $\mathbf{O}' = TCI + T\mathbf{c} + \mathbf{t}$. After receiving \mathbf{O}' from the remote (untrusted) server, the client calculates:

$$\mathbf{O} = M\mathbf{O}' + \mathbf{m}, \quad (10)$$

where \mathbf{m} is a constant vector. Hence, for correct execution, we need to have:

$$C\mathbf{I} + \mathbf{c} = MTCI + MT\mathbf{c} + M\mathbf{t} + \mathbf{m} \quad (11)$$

This means that the following two equalities have to be satisfied:

$$C = MTC \quad (12)$$

$$\mathbf{c} = MT\mathbf{c} + M\mathbf{t} + \mathbf{m} \quad (13)$$

The details of the solution are omitted due to space concerns.

6 Selection of T and M

In this section, we study the required relationship between T and M to ensure correctness. First, we focus on scalar codes, and then array based codes.

6.1 Scalar Codes

We start with Equation (3), and proceed as follows:

$$\begin{aligned} \mathbf{O} &= M\mathbf{O}' \\ \mathbf{O} &= MC'\mathbf{I} \\ \mathbf{O} &= MTCI \\ C\mathbf{I} &= MTCI \end{aligned}$$

Since $\mathbf{I} \neq \mathbf{0}$ (zero vector), from this last equality, we can obtain:

$$C = MTC. \quad (14)$$

In other words, M must be left inverse of T . Let us now discuss the dimensionalities of matrices T and M . Assuming that \mathbf{I} has n entries and \mathbf{O} has m entries, C is $m \times n$. Therefore, the only dimensionality requirement regarding T is that it needs to have m columns, and similarly M needs to have m rows. Thus, matrices T and M are $k \times m$ and $m \times k$, respectively. That is, we have a flexibility in selecting k . There is also an alternate way of generating C' from C . More specifically, we can have $C' = CT$. In this case, we can proceed as follows:

$$\begin{aligned}
\mathbf{O} &= M\mathbf{O}' \\
\mathbf{O} &= MC'\mathbf{I} \\
\mathbf{O} &= MCT\mathbf{I} \\
C\mathbf{I} &= MCT\mathbf{I}
\end{aligned}$$

Since $\mathbf{I} \neq \mathbf{0}$ (zero vector), from this last equality, we can obtain:

$$C = MCT. \quad (15)$$

However, it should be noticed that with this formulation, we do not have a flexibility in selecting the dimensions of transformation matrices T and M . Specifically, T should be an $n \times n$ matrix, and M should be an $m \times m$ matrix; i.e., we need to work with square matrices. In this case, given a T matrix, we can determine M by solving the resulting linear system of equations. Alternately, we can adopt the following strategy. Let us select an M matrix first, and define a new matrix Q as $Q = MC$. Using this, we can proceed as follows:

$$\begin{aligned}
C &= MCT \\
C &= QT \\
Q^T C &= Q^T QT \\
(Q^T Q)^{-1} Q^T C &= T
\end{aligned}$$

Notice that the last equality gives us the T matrix. It must be noted, however, that in order to use this strategy, we need to select a suitable M such that $Q^T Q$ is invertible, i.e., it is non-singular.

6.2 Array Based Codes

Selection of the matrices T and M for array based codes is similar to their selection for scalar codes. Starting with Equation (7), we proceed as follows:

$$\begin{aligned}
\mathbf{O}' &= M\mathbf{O}'' \\
\mathbf{O}' &= MC''\mathbf{I} \\
\mathbf{O}' &= MTC'\mathbf{I} \\
C'\mathbf{I} &= MTC'\mathbf{I}
\end{aligned}$$

Note that the last equality is obtained from the penultimate one because loop transformation changes only the order by which the elements are accessed during execution but not the result of the execution itself. So, it is correct to equate \mathbf{O}' and $C'\mathbf{I}$; and since $\mathbf{I} \neq \mathbf{0}$ (zero vector), from this last equality, one can obtain:

$$C' = MTC'. \quad (16)$$

In other words, M must be left inverse of T . Note that similar to the case with scalar codes, there is also an alternate way of generating C'' from C' . More specifically, we can set C'' to $C'T$. The details are omitted due to space concerns.

7 Multiple Server Case

In this section, we discuss how the proposed approach can be extended to the case with multiple servers. Due to space concerns, we focus only on scalar computations. The execution scenario in this case, which is depicted in Figure 2, can be summarized as follows. The client divides the computation (C) to i sub-computations, where the i^{th} sub-computation is represented by matrix C_i , where $1 \leq i \leq p$. In mathematical terms, this can be expressed as follows:

$$\mathbf{O} = \mathbf{C}\mathbf{I} \quad (17)$$

$$\begin{pmatrix} \mathbf{O}_1 \\ \mathbf{O}_2 \\ \vdots \\ \mathbf{O}_p \end{pmatrix} = \begin{pmatrix} C_1 & 0 & 0 & 0 \\ 0 & C_2 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & C_p \end{pmatrix} \begin{pmatrix} \mathbf{I}_1 \\ \mathbf{I}_2 \\ \vdots \\ \mathbf{I}_p \end{pmatrix}. \quad (18)$$

Note that C_i operates on input I_i and generates output O_i . The client then determines a T_i matrix and computes $C'_i = T_i C_i$. Then, C'_i is sent to the server i , which in turn computes $O'_i = C'_i I_i$, and sends it back. After having received O'_i from server i , the client calculates $O_i = M_i O'_i$, where M_i is the data transformation matrix used in conjunction with T_i . Note that, for correctness, we should have $M_i = T_i^{-1}$. After collecting O'_1, O'_2, \dots, O'_p and obtaining O_1, O_2, \dots, O_p , the client merges these outputs into the desired out vector \mathbf{O} . A similar analysis could be conducted by using the alternate formulation as well (see Section 6).

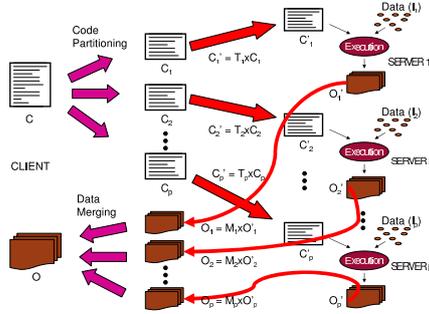


Fig. 2. High-level view of secure code execution in multiple untrusted servers. Note that the original computation, C , is divided into p sub-computations, and each sub-computation is set to get executed on a different server.

8 Example

In this section an example on our approach is presented. Due to space restrictions only an example based on scalar codes is presented.

Consider the following linear code fragment taken from [5]:

$$\begin{aligned} dx0 &= x0 - x1 - x12 \\ dy0 &= y0 - y1 - y12 \\ dx1 &= x12 - x2 + x3 \\ dy1 &= y12 - y2 + y3 \end{aligned}$$

The computation matrix for this computation is:

$$C = \begin{pmatrix} 1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \end{pmatrix}.$$

Given the input represented by

$$\mathbf{I} = (x0 \ x1 \ x2 \ x3 \ x12 \ y0 \ y1 \ y2 \ y3 \ y12)^T = (10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10)^T,$$

the original output can be computed as:

$$\mathbf{O} = (dx0 \ dy0 \ dx1 \ dy1)^T = C\mathbf{I} = \begin{pmatrix} 1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \end{pmatrix} (10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10)^T = \begin{pmatrix} -10 \\ -10 \\ 10 \\ 10 \end{pmatrix}.$$

We now discuss how the same computation is carried out in an untrusted remote server environment. Let us assume the following loop transformation matrix:

$$T = \begin{pmatrix} 1 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}.$$

In this case, the transformed computation matrix ($C' = TC$) is:

$$C' = \begin{pmatrix} 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 & -1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 \\ -1 & 1 & -1 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1 & 2 \end{pmatrix}.$$

Consequently, the computation performed by the remote server is:

$$\mathbf{O}' = C'\mathbf{I} = (-30 \ -10 \ 20 \ 20)^T.$$

After receiving the output generated by the server, the client needs to multiply it by $M = T^{-1}$. In this case, M can be found as:

$$M = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

Therefore, the resulting output is:

$$M\mathbf{O}' = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -30 \\ -10 \\ 20 \\ 20 \end{pmatrix} = \begin{pmatrix} -10 \\ -10 \\ 10 \\ 10 \end{pmatrix},$$

which is the same as the intended output that would be computed from $\mathbf{O} = C\mathbf{I}$.

9 Experiments

In this section, we explain how the overheads of the proposed mechanisms are calculated. To test the proposed approach, we implemented it within an optimizing compiler (built upon SUIF [2]) and performed experiments with three applications that model execution in a sensor-based image processing environment. The first of these applications, TRACK_SEL 2.0, implements a vehicle tracking algorithm which is used to support missile systems by maintaining surveillance against incoming targets and providing the data required for targeting, launch, and midcourse guidance. The second application, SMART_PLANNER, is an emergency exit planner. The application determines the best exit route in case of an emergency which is detected, in the current implementation, using heat sensors. Our third application is named CLUSTER and implements a dynamic cluster forming algorithm. It's main application area is energy-efficient data collection in a wireless sensor environment. All these three applications are written in C++, and their sizes range from 1,072 to 3,582 C++ lines (excluding comment lines). For each application in our experimental suite, we compared two different execution schemes. In the first scheme, which is not security oriented, the application is shifted from one workstation to another and executed there using local data. The second execution implements the proposed approach. Specifically, the application is first transformed and then sent to the remote machine and, when the results are received, they are transformed as explained in the paper. We measured the additional performance overhead incurred by our approach over the first execution scheme. More specifically, we computed the ratio

$$\frac{(\text{loop restructuring time} + \text{data transformation time})}{(\text{total execution time})},$$

where “total execution time” includes the time spent in computation in the remote machine and the time spent during communication. We found that the value of this ratio was 0.0421, 0.0388, and 0.0393 for the benchmarks TRACK_SEL 2.0, SMART_PLANNER, and CLUSTER, respectively. That is, the extra code/data transformations required by our approach do not bring significant performance overheads, which means that we pay a small price to hide the semantics of the application from the remote machine.

10 Concluding Remarks

This paper presents a novel, automated solution to the problem of protecting mobile applications from untrusted remote hosts. These applications based on scalar and array based codes, are automatically transformed with the help of an optimizing compiler to prevent reverse engineering.

Future work involves extending the proposed approach to cater to general purpose programs that cannot be readily expressed as a linear function of the inputs. In order to do so, a method to represent the non-linear code in an array

format is required. One possible way is to simply treat a variable, such as a , and a non-linear sub-expression that it appears in, such as a^2 , to be different variables. That is, we can assume $a^2 = x$ and use x in our formulation. This technique however does not solve the problem of an expression like a^b where b is itself a variable. Further, the problem of recognizing dependencies between an assignment to a and the use of x is complicated. Once the representation mechanism exists, it can be used with the transformation techniques to hide the semantics of the original code as shown in this paper.

References

1. M. Abadi and J. Feigenbaum. Secure circuit evaluation. *Journal of Cryptology*, 2(1):112, 1990.
2. S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. R. Murphy, R. S. French, M. S. Lam, and M. W. Hall. Multiprocessors from a Software Perspective *IEEE Micro*, June 1996, pages 52-61.
3. A. Bremner-Barr, H. Levy. Spoofing prevention method. In *Proc. of INFOCOM 2005, Volume 1*, pages 536 - 547, 2005.
4. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing*, pages 218–229, May 1987.
5. C. Li, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In *Proc. of the International Symposium on Microarchitecture*, 1997.
6. S. Loureiro, L. Bussard, and Y. Roudier. Extending tamper-proof hardware security to untrusted execution environments. In *Proc. of CARDIS*, 2002.
7. S. Loureiro and R. Molva. Function hiding based on error correcting codes. In *Proc. of the International Workshop on Cryptographic Techniques and Electronic Commerce*, pages 92–98, 1999.
8. T. Sander and C. F. Tschudin. Towards mobile cryptography. In *Proc. of the 1998 IEEE Symposium on Security and Privacy*, pp. 215–224, 1998.
9. A. Tripathi, N. Karnik. A Security Architecture for Mobile Agents in Ajanta. In *Proc. of the International Conference on Distributed Computing Systems*, 2000.
10. T. Sander and C. Tschudin. On software protection via function hiding. In *Proc. of the Second Workshop on Information Hiding*, 1998.
11. M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.
12. A. C. Yao. Protocols for secure computations. In *Proc. of the IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.
13. A. C. Yao. How to generate and exchange secrets. In *Proc. of the IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.
14. B. Yee. A sanctuary for mobile agents. *Technical Report CS97-537*, Department of Computer Science and Engineering, April 1997.