Practical Instance Hiding

Rida A. Bazzi and K. Selçuk Candan
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406 U.S.A.
{bazzi,candan}@asu.edu

Abstract

We study the problem of computing with encrypted data and code. We give general definitions for code and data hiding and we identify a new type of code hiding we call structure hiding. We introduce a general class of programs that we call piece-wise linear programs and we develop data and code hiding schemes for them. We prove that our schemes leak no information about the input or output of the computation and that they leak minimal information about the code. The hiding schemes we present are the first practical schemes for a large class of programs.

1 Introduction

Computing with encrypted data or encrypted code is desirable for many applications. In this paper, we are interested in the application of computing with encrypted data and code to mobile computing systems. In a mobile computing system, the mobile computing hardware has limited computation power and it is desirable to do computations remotely because of resource limitations. Unfortunately, executing an application remotely exposes the application code and data to non-trusted, potentially malicious, distributed computing infrastructures. To get around this difficulty, we envision a model in which the owner of the code pre-registers multiple *hidden* applications with a non-trusted distributed computing infrastructure. The applications can be replicated to increase availability. When the mobile owner wants to execute an application, it contacts the distributed infrastructure and provides its *encrypted* input to the closest computing element in the distributed infrastructure that contains a copy of its application. The computing element returns the *encrypted* output to the owner who recovers the actual output. The owner's goal is to benefit from the computation power and resources of the host while revealing minimum information about her data and/or code.

Work on computing with encrypted data or code uses many models of interaction that can differ in subtle ways. For example, the owner can interact with one or many hosts and the interaction can consist of a constant number or a variable number of rounds. Also, the assumptions about the computation powers of the owner and hosts can vary. Some results assume that the host is an oracle with unlimited computation power and that the owner can recover the results of the computation in polynomial time [1].

Another aspect of computing with encrypted data and function is *robusteness*, the ability of the owner to tell whether the results returned by the host are correct. A detailed description of related work appears in Section 2.

The applications we are targeting impose natural restrictions on acceptable solutions to the problem of computing with encrypted code and data. These restrictions do not necessarily apply to other application considered by other researchers. The following are the restrictions we impose on our solutions:

- No interaction. We allow no interaction between the owner and host beyond the sending of the initial input and the receipt of the results.
- Single hosts. We do not allow the owner to *divide* its application and send it to different hosts that do not communicate with each other as a means of hiding. We assume that hosts cannot be prevented from communicating with each other and we do not tolerate the communication overhead involved in having multiple hosts for one application.

In this paper, we propose efficient mobile code hiding techniques that provide security and reliability guarantees for a large class of practical programs. By security we mean that host sites can not gain unwarranted information. By reliability, on the other hand, we mean that host sites can not provide incorrect results. Consequently, these techniques provide the support needed for mobile code to survive in an open hostile environment.

This paper is organized as follows. Section 2 describes related work. Section 3 formally defines piece-wise linear programs and discusses their limitations. Section 5 provides formal definitions of various types of hiding. Section 6 presents our code hiding techniques for piece-wise linear programs. Section 6.3 present a generalization of our techniques to larger class of programs that can use arrays. Section 7 concludes the paper.

2 Related Work

Code Hiding and Tamper Resistance Related works differs greatly in the assumption they make. Sander and Tschudin [10] use homomorphic encryption functions to hide polynomials. Their encryption schemes are efficient, but their scheme has weaknesses that our previously developed scheme to hide polynomial does not have. For instance, the host can tell which coefficients are equal to zero and can construct tables to use in decrypting the coefficients. Also, if two coefficients are equal, then the corresponding encrypted coefficients will also be equal. Furthermore, they do not consider the distribution of the coefficients in their solution, and their argument for the security of their solution is informal. Formal theoretical work on input hiding and code hiding appears in [1, 3, 7] among others. In [7], Feigenbaum considers the problem of encrypting problem instances. In her work, she attempts to hide the input to a function, but does not hide the function itself. Also, she does not give a formal definition of what hiding means. In [1], the results of [1] are refined and a formal definition of hiding and information leakage is presented. In [1], it is assumed that the owner of the code has polynomial-time computation power and that the host is an oracle. This leads to some hiding solutions for some problems in which the owner hide polynomial-time functions by calculating the answer locally. In [3], a general hiding scheme for boolean functions is presented. Unfortunately, the scheme uses a standard representation of boolean function that can introduce an exponential overhead, which makes it impractical. Also, the scheme only applies to boolean functions and not to general functions. A drawback of the models used in [1, 3] and models used by other researchers is that it allows a polynomial number of communication rounds between the host and the owner of the code. For the applications we are considering, it is not A different approach is taken in [11]. In [11], it is assumed that up to t machines in the system are faulty. Faulty platforms are detected by replicating the agent computation and executing them on different hosts. Results are obtained by collecting the results and the use of voting. The solution requires that the computation be deterministic so that comparisons between the results of different machines be meaningful. In addition the solution uses interesting encryption techniques to make sure that faulty machines cannot collude to spoof results. The work of [11] does not consider techniques to detect code and data tampering as we do.

Code Obfuscation Aucsmith suggested the use of code obfuscation which makes code unintelligible by reordering it, but does not study formally its security. Code obfuscation as used by Aucsmith cannot form a basis for provable agent security. The main idea is to develop scenarios in which an attacker is confused about the real execution of the code. This can be done by adding goto's to the code and even using multi-threads to make tracing harder. Work on code obfuscation tends to be informal and ad hoc and lacks the rigor that our approach presents. While there a lot of work on code obfuscation([4] and [5] are good examples), we do not discuss it because it is informal and ad hoc and is not relevant to *provable* code hiding and tamper resistance techniques.

Steganography Steganography deals with the hiding of messages so that potential eavesdroppers do not even know that a message is being sent. For example, in null ciphers, the first letter of each word spells out a message. Other example includes hiding information in images and the recent work on digital watermarking. Steganography work is usually concerned with hiding information and does not deal with hiding programs.

Host Integrity Host integrity is the problem of protecting the host from the mobile code. Necula [9] developed formal techniques that enable the host to prove that the mobile code is benign. Schneider [11] developed formal techniques to specify and enforce security restrictions on mobile code. Host integrity is complementary to the problem of code integrity and uses different techniques from the one needed for hiding and tamper resistance.

3 Piece-wise Linear Programs

In this section we define the classes of programs for which we provide code and instance hiding.

We consider programs in which all assignments and conditionals are linear in the variables. Such programs are common in scientific applications.

Informally, piece-wise linear programs consist of linear assignments and conditions, for loops with constant index bounds, arrays whose indices linearly depend only on the loop counters and constants. Consequently a large subset of practical programs can be expressed using piece-wise linear programs and be hidden by the techniques we propose.

In what follows, we formally define piece-wise linear programs.

Definition 3.1 (Constant) *A constant is an element of Z_m, the set of integers modulo m.*

Definition 3.2 (Variable) *A variable is a symbol from an infinite set of symbols, $\mathcal{V}$ (excluding the symbols used to denote the elements of Z_m and other meta symbols).*

Definition 3.3 (Statement) *A statement is an* **assignment** *statement, an* **if** *statement, a* **for** *statement, or a*

compound statement. Given a statement, s, the variables contained within s are denoted as $var(s)$.

Definition 3.4 (Compound Statement) *A compound statement is a finite sequence of statements.*

Definition 3.5 (Assignment Statement) *An assignment statement A has the form*

$$\mathbf{x} = M\mathbf{x};$$

where $\mathbf{x}$ is a vector consisting of l distinct variables, and M is an $l \times l$ matrix for some $l \in N$. We say that M is the assignment matrix of A.

Intuitively, a matrix describes a sequence of linear assignments. The following example shows the mapping from common code (in C syntax) to an assignment matrix:

Example 3.1 *The two statements*

```
y = x + 1;
z = 2 * w + y + 5;
```

can be represented by the following assignment statement:

$$\begin{bmatrix} w \\ x \\ y \\ z \\ u \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \\ u \end{bmatrix}$$

Note that the above assignment statement contains a new variable, u, that is not contained in the original code. This variable, u, is called the unit variable; it is assumed to be initialized to 1 at the beginning of each program and its value is never changed. Also, in the assignment statement notation, the new values of the variables are calculated from the old values in parallel, whereas in the two statements above, they are calculated serially (z depends on x in the matrix and not on y and the constant is 6 not 5). This latter point is discussed in more details in Section 6, where we introduce assignment blocks.

This matrix-based representation provides a convenient way to develop transformation-based code hiding and tamper resistance techniques. Consequently, we use matrices to describe our transformations as well as the source and transformed programs.

Definition 3.6 (Conjunctive Condition) A conjunctive condition $C^{\wedge}$ has the form

$$M\mathbf{x} \leq 0;$$

where $\mathbf{x}$ is a sequence of l variables, and M is a $c \times l$ matrix for some $c \in N$. We say that M is the condition matrix of $C^{\wedge}$.

The following example shows how a conjunction with many sub-conditions may be expressed using a single matrix:

Example 3.2 We can represent the conjunctive condition

```
(2 * x >= 5) and (z <= 3 * w + x)
```

in terms of a condition matrix and variable vector multiplication, as follows:

$$\begin{bmatrix} 0 & -2 & 0 & 0 & 5 \\ -3 & -1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \\ u \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that conjunctive conditions cannot express "or ($\vee$)". For conditions which contain "$\vee$", we use disjunctive normal form (DNF): $(C_1^\wedge \vee \ldots \vee C_k^\wedge)$.

Definition 3.7 (DNF Condition) A DNF condition, $C^\vee$, is of the form

$(C_1^\wedge \vee \ldots \vee C_k^\wedge)$

where each $C_i^\wedge$ is a conjunctive condition, and $k \geq 1$.

Definition 3.8 (If Statement) An if statement I has the form

`if` (C) `PS` else NS;

where C is a DNF condition, and PS and NS are compound statements.

The *if* statement uses assignment and condition matrices as their building blocks. In addition to assignment matrices, the for statement has a constant that detrmines the number of iterations.

Definition 3.9 (For Statement) A for statement F has the form

`for` (a) FS;

where $a \geq 1$ is an integer constant and FS is a compound statement.

The *for* statements we define are expressive enough to cover a large class of practical programs, but they are not equivalent to the more general *for* statements in common high-level programming languages.

Definition 3.10 (Piece-wise Linear Program) A piece-wise linear program, P, is a pair (S, V), where S is a finite sequence of statements and V is a set of variables such that $\bigcup_{s \in S} var(s) \subseteq V$. S typically consists of the concatenation of two sequences S_I and S_E. S_I is the *initialization* sequence and S_E is the *execution* sequence. Let $\mathcal{P}$ be the set of all piece-wise linear programs.

In our work, we identify various properties of a program that can be hidden. In different situation, we might hide one or more of these properties. We expect to have a tradeoff between the number and types of properties hidden and the cost and strength of the hiding. In what follows we define the structure or flow graph of a piece-wise linear program. Although a structure does not uniquely identify a program, it can still reveal crucial information about its content to a hostile host. The following is a formal definition of program structure.

Definition 3.11 (Structure of a Piece-wise Linear Program) The structure of a piece-wise linear program P is a graph $\sigma = G(P)$ satisying the following.

- The structure, σ, has one entry node, e_σ, and a set of exit nodes, X_σ.

- The structure, σ, of A, where A is an assignment statement is a graph consisting of a single node and no edges ($X_\sigma = \{e_\sigma\}$).
- The structure, σ, of "S_1 $S2$", where S_1 and S_2 are compound statements is the graph G obtained by joining every *exit* node of $G(S_1, V)$ to every *entry* node of $G(S_2, V)$. Furthermore, we have, $e_\sigma = e_{G(S_1,V)}$ and $X_\sigma = X_{G(S_2,V)}$.
- The structure, σ, of "**if** (C) PS *else* FS" is the graph G obtained by introducing a new node, e_σ (entry node), and linking it to the entry nodes of $G(PS, V)$ and $G(FS, V)$. The exit nodes, X_σ, of G are the union of the exit nodes of $G(PS, V)$ and $G(FS, V)$
- The structure, σ, of "**for** (a) FS" is the graph G' obtained by introducing one new entry node, e_σ and one exit node, x_σ ($X_\sigma = \{x_\sigma\}$) to $G = G(FS, V)$ and the following links: e_Σ to e_G, X_G to X_Σ, and X_G to e_G.

Related work on hiding does not consider hiding the structure of the program because it is concerned with hiding mathematical functions that have no structure. In contrast, we consider functions that are defined by programs and that might not be easily deduced from an inspection of the code.

The definitions presented so far describe the static properties of piece-wise linear programs. The code hiding techniques that we propose assume that the input program can be converted into this piece-wise linear form.

Hiding the static properties prevents hosts from getting unauthorized information by analyzing the code. However, we also need to be concerned about the dynamic properties of a program, or the information that can be revealed during the program execution. In the next section, we introduce these dynamic aspects.

4 Execution of Piece-wise Linear Programs

In this section, we introduce dynamic properties of a piece-wise linear program and, more specifically, define its execution and input-output relationships.

The main dynamic property of a program is the set of values that its variables take at different points of execution. This is defined folmally as follows.

Definition 4.1 (Value assignment) A value assignment of a set V of variables is a mapping $m : V \mapsto Z_m^{|V|}$. Let $\mathcal{A}(\mathcal{V})$ be the set of all value assignments for $\mathcal{V}$.

Definition 4.2 (Program state) A state of a program $P = (S, V)$ is a value assignment of V. A program state specifies a value for each of the variables of the program.

Sometimes, we talk about the values of a subset of V in a given state. These values are obtained by restricting the state to the variables of interest. One such restriction is the input of a program. This input is what the owner of the mobile code sends to the host along with the executable program.

Definition 4.3 (Input variables of a program) A set of input variables, IV_P of a program $P = (S, V)$ is a subset of V.

Definition 4.4 (Input of a program) An *input* of a program $P = (S, V)$ is a value assignment of IV_P.

Informally, the input of a program describes the values of the input variables at the initial state of the program execution:

Definition 4.5 (Initial state of a program) An *initial state*, i_0, of a program $P = (S, V)$ for an input, i, is a value assignment of V such that $\forall_{v \in IV_P}$, $i(v) = i_0(v)$.

A program execution, then, is a sequence of program states. The initial state described above is the first state of the program execution:

Definition 4.6 (Execution) The execution, $exec(P, i)$, of a program $P = (S, V)$ on input i is a sequence $M(i_0) = m_0, m_1, \ldots, m_n$ of program states such that $m_0 = i_0$ is an initial state corresponding to i, and the remaining states satisfy the following conditions.

- If $M_1(i_1) = m_{10}, \ldots, m_{1k}$ is the execution of $P_1 = (S_1, V)$ with initial state i_1 and $M_2(i_2) = m_{20}, \ldots, m_{2k'}$ is the execution of $P_2 = (S_2, V)$ with initial state i_2, then $M_3(i_1) = M_1(i_1), M_2(m_{1k})$ is the execution of $P = (S_1\ S_2, V)$.
- If $P = (S, V)$, where S is the assignment statement $\mathbf{x} = M\mathbf{x}$, then the execution of P with initial state i_1 is the sequence i_1, m_1, where $m_1(v) = i_1(v)$ if $v \notin \mathbf{x}$ and $m_1(v)$ is the jth entry of $M\mathbf{x}$ if v is the jth variable of $\mathbf{x}$.
- If $P = (S, V)$, where S is the *if* statement $if\ C\ PS\ else\ NS$, then the execution of P with initial state i_1 is
  - the sequence M_P, where M_P is the execution sequence of (PS, V) with initial state i_1, if C is true for the value assignment i_1, or
  - the sequence M_N, where M_N is the execution sequence of (NS, V) with initial state i_1, if C is false for the value assignment i_1.
- If $P = (S, V)$, where S is the *for* statement $for\ (a)\ SF$, then the execution sequence of P with initial state i_1 is
  - if $a \neq 0$, the sequence M_1, M_{a-1}, where $M_1 = m_0, m_1, \ldots, m_k$ is the execution sequence of (FS, V) with initial state i_1 and M_{a-1} is the execution sequence of $(for(a-1)SF, V)$ with initial state m_k, or
  - if $a = 0$, the singleton sequence, i_1.

The last state of the program execution is called the output state. The output of the program is the set of value taken by its output variable at the end of its execution. This output is what the host returns to the owner of the mobile code. Below we define it formally.

Definition 4.7 (Output variables of a program) A set of output variables, OV_P of a program $P = (S, V)$ is a subset of V.

Definition 4.8 (Output of a program) The output, $out(P, i)$, of a program $P = (S, V)$, given input i, is the restriction of the state of P at the end of its execution to its output variables, OV_P. Note that $out(P, i) \in A(OV_P$.

The input and output variables of a program are usually those that are significant for the owner of the code. The relationship between the input and output variables determine the function of the program and is an important property of the program. Although, a given input-output relationship does not uniquely identify a program, since it describes the program semantics, it usually is more valuable to a hostile host than the individual assignments, conditions, or the program structure (unless the hot's goal is to reverse engineer the code). Consequently, we need to hide the input-output relationship of the program from the hosts.

Definition 4.9 (Input-output relationship of a program) The input-output relationship of a program, $P = (S, V)$, is a mapping $IO\_rel(P) : A(IV_P) \mapsto A(IO_P)$.

Now that we defined piece-wise linear programs, their structures, executions, and input-output relationships, we are ready to define code hiding and discuss hiding approaches we apply to code compiled for different tasks.

5 Code Hiding for Piece-wise Linear Programs

Code hiding is the process of transforming a given program into an other such that the host can not recognize the individual statements, the structure, and the input-output relationships. In this section, we formally define code hiding. The definitions we propose are an important contribution of our work and is needed for any credible and provable techniques for providing mobile code integrity.

In previous sections, we have described many static and dynamic properties of programs. We say that a host recognizes a property of the input code if the host can compute that property with a non-negligible probability. Below, we define what does it mean for two programs to have the same properties. In all definitions, we assume that the programs have the same sets of input and output variables.

Definition 5.1 (Output equivalence) Two programs P_1 and P_2 are called output equivalent ($P_1 =_{o,i} P_2$), for an input i iff $out(P_1, i) = out(P_2, i)$.

Definition 5.2 (Execution equivalence) Two programs P_1 and P_2 are called execution equivalent ($P_1 =_e P_2$) iff for all possible inputs, i, $out(P_1, i) = out(P_2, i)$. In other words, $IO\_rel(P_1) = IO\_rel(P_2)$.

Definition 5.3 (Structure equivalence) Two programs P_1 and P_2 are called structure equivalent ($P_1 =_s P_2$) iff $G(P_1) = G(P_2)$.

Code hiding is the process of transforming a given program and the associated data in such a way that the host can not find a program that has the same properties as the input program. First, we formally define code and value transformations.

Definition 5.4 (Code transformation) A code transformation is a mapping $CT : \mathcal{P} \mapsto \mathcal{P}$. We call $CT(P)$ the transformed program and P the source program.

Definition 5.5 (Value transformation) A mapping $VT : \mathcal{A} \mapsto \mathcal{A}$ is a value transformation if for every value assignment of a finite set of variables V, $VT(V)$ is a value assignment of V.

Definition 5.6 (Code-value transformation) A triplet of transformations, $\langle CT_H, VT_H, VT_R \rangle$, is called a *code-value transformation* if given any program P and input i, $CT_H(P)$ is an other program, such that

$$out(P, i) = VT_R(out(CT_H(P), VT_H(i))).$$

CT_H is called a code hiding transformation, VT_H is called an input hiding transformation, and VT_R is called an output recovering transformation.

In general the host may not be able to find a program with the exact same properties as the original, but it may be able to learn some information that can help it to guess these properties with non-negligible probability. We define hiding as transforming a given program and its input in such a way that, for the host, guessing the properties of the original program using the clues revealed by the transformed code and the input is as hard as guessing these properties without the help of the clues. Since, different transformation techniques may reveal different amount of clues, we use a function f which quantifies the amount of clues revealed:

Definition 5.7 (Hiding) Given $P_H = CT_H(P)$ and $i_H = VT_H(i)$, a code-value transformation, $\langle CT_H, VT_H, VT_R \rangle$, is called

- *f-input hiding* iff the probability of finding a value assignment $i_1 = i$ is $prob(i_1 = i \mid (P_H, i_H) = (CT_H(P), VT_H(i))) = p + O(f(|i|))$, where $p = prob(i_1 = i)$.
- *f-structure hiding* iff the probability of finding a program P_2 such that $G(P_2) =_s G(P)$ is $prob(G(P_2) =_s G(P) \mid (P_H, i_H) = (CT_H(P), VT_H(i)) = p + O(f(|P| + |i|)_)$, where $p = prob(G(P_2) =_s G(P))$.
- *f-weak code hiding* iff the probability of finding a program P_2 such that $P_2 =_o P$ is $prob(P_2 =_o P \mid (P_H, i_H) = (CT_H(P), VT_H(i))) = p + O(f(|P| + |i|))$, $p = prob(P_2 =_o P)$.
- *f-code hiding* iff the probability of finding a program P_2 such that $P_2 =_e P$ is $prob(P_2 =_e P \mid (P_H, V_H) = (CT_H(P), VT_H(i))) = p + O(f(|P| + |i|))$, where $p = prob(P_2 =_e P)$.

A code-value transformation is called input, structure, weak code, and code hiding respectively if and only if $1/f(x)$ is a sup-polynomial in x.

Note that hiding can also be defined over a period of time: the same program, with different inputs, executed many times at the host could provide more clues than the sum of the clues of the individual executions. In this paper, we do not address the needs of this model.

Definition 5.8 (Valid code hiding) A hiding transformation, $\langle CT_H, VT_H, VT_R \rangle$, is called valid iff the cost of calculating $VT_H(i)$, and $VT_R(out(CT_H(P), i))$ is less than the cost of calculating $out(P, i)$.

Note that in the definition of valid code hiding we do not include the cost to transform the program because that is done only once.

Definition 5.9 (Acceptable code hiding) A hiding transformation, $\langle CT_H, VT_H, VT_R \rangle$, is called *acceptable subject to c overhead* iff the cost of calculating $out(CT_H(P), i)$ is $O(c(X))$; where X is the cost of calculating $out(P, i)$ and c is an upper-bound function. Typically $f(X)$ is negligible compared to X

In addition to the definitions above, we can define *local* versions of the definitions that state that the hiding apply to any locality of the program. The hiding transformations we develop have this locality property.

6 Random Linear Transformations for Piece-wise Linear Code Hiding

In this section, we describe the code and transformation processes in greater detail. We first start with some introductory definitions.

Definition 6.1 (Permutation Sequence) A permutation of length l is a sequence of integers of length l that contains all the integers from 1 to l. A permutation of a sequence, s, prescribes the new location of each element of s in the permuted sequence, s'.

One of the first steps in code hiding is to *pad* the program variables such that the host can not identify the input variables.

Definition 6.2 (Variable Padding) Let $\mathbf{x}$ and $\mathbf{y}$ be two sequences of distinct variables. A *padding* π *of* $\mathbf{x}$ *by* $\mathbf{y}$ is a variable sequence $\mathbf{x}_p$ which is a random permutation of the sequence $\mathbf{x}, \mathbf{y}$. The permutation of the padding π is denoted Π_p.

Since variable padding involves permutation of the program variables, the corresponding assignment and condition matrices need to be similarly permuted.

Definition 6.3 (Matrix Permutation) The permutation, $\Pi_{rc}(M)$, of a matrix M of size $l \times l$ with permutations Π_c and Π_r of size l is obtained by permuting the rows of M with Π_r and then permuting the columns of the resulting matrix with Π_c. The permutation of the rows of M with p_r is denoted $\Pi_r(M)$. The permutation of the columns of M with p_c is denoted $\Pi_c(M)$.

Theorem 6.1 $\Pi_r(\Pi_c(M)) = \Pi_c(\Pi_r(M)) \qquad (= \Pi_{rc}(M)) = \Pi_{cr}(M)))$.

Proof 6.1 $\Pi_r(\Pi_c(M))[i,j] = M[\Pi_r(i), \Pi_c(j)] = \Pi_c(\Pi_r(M))[i,j]$, *where* $\Pi(i)$ *is the result of applying a permutation* Π *on element* i *in the range* $1, \ldots, l$.

Next, we describe how to hide different types of statements using the padding technique.

6.1 Assignment Transformation

In order to hide an assignment matrix, we introduce a set of random variables and random assignments; i.e., we *pad* the assignment.

Definition 6.4 (Assignment Padding) A *padding* of an assignment statement "$\mathbf{x} = M\mathbf{x}$;" with variable sequence $\mathbf{y}$ is an assignment statement

$$\mathbf{x}_p = M_p \mathbf{x}_p;$$

where $\mathbf{x}_p$ (with permutation sequence Π_p) is a padding π of $\mathbf{x}$ by $\mathbf{y}$ and $M_p = \Pi_{pp}(M')$, where

$$M' = \begin{bmatrix} M & M_{xy} \\ 0 & M_{yy} \end{bmatrix}$$

and M_{xy} and M_{yy} are random matrices whose elements are chosen from Z_m^+ randomly according to the uniform distribution. M_p is called an assignment padding matrix of A with $\mathbf{y}$.

It is clear from M' that $\mathbf{y}$ depends on both $\mathbf{x}$ and $\mathbf{y}$ whereas $\mathbf{x}$ depends only on $\mathbf{x}$. It follows that an adversary can easily analyze the padded matrix and itentify the parts that are $\mathbf{x}$-related and the parts that are $\mathbf{y}$-related.

To achieve hiding, we first pad the assignment, and then we apply two random transformations, one to the left and on to the right of the resulting padded assignment matrix. The resulting matrix is random and totally independent of the original assignment matrix (in a sense that we will formalize below). In what follows, when we talk about a random matrix, we mean a matrix whose elements are chosen from Z_m^+ randomly according to the unifrom distribution.

Definition 6.5 (Assignment Hiding) The hiding of an assignment statement A of the form "$\mathbf{x} = M\mathbf{x}$;" with variable sequence $\mathbf{y}$, corresponding assignment padding matrix M_p, and invertible matrices I and J is

$$\mathbf{x}_p = M_h \mathbf{x}_p;$$

where $M_h = IM_pJ$.

Theorem 6.2 *If I and J are random independent matrices and M be a matrix independent from I and J such that one of the rows of M consists of elements of Z_m^+. Then, the matrix IMJ is independent from M.*

Proof 6.2 *Uses standard probability arguments and the fact that the entries in the matrices I and J are random elements of Z_m^+ and the fact that every column of M contains a non-zero entry.*

Theorem 6.3 *Let M, I and J be matrices that satisfy the conditions of Theorem 6.2, then*

$$p(M = M_0 \mid IMJ = H_0) = p(M = M_0)$$

Proof 6.3 *Follows from the fact that IMJ is independent of M (proved in the previous theorem).*

Theorem 6.3 shows that an assignment statement can be hidden by multiplying it my two random matrices (in fact, muliplying with a single matrix would suffice, but we need two matrices per assignment statemment when we hide a sequence of assignment statements).

Note that given a sequence, $S = M_1, M_2, \ldots, M_n$ of assignment matrices, hidden with the matrices $(I_1, I_2), (I_2^{-1}, I_3), \ldots (I_{n-1}^{-1}, I_1^{-1})$, the combined hidden assignment matrix, $M_{S,h}$, has the following property:

$$M_{S,h} = I_1 M_1 M2, \ldots, M_n I_1^{-1}.$$

Consequently, given an input vector, V, the following equality follows:

$$I^{-1} M_{S,h} I V = M_1 M_2 \ldots. M_n V$$

Next, we show, how hiding an assignment statement can be used in hiding a for statement.

Theorem 6.4 (Hidden Assignment Execution) Let $P = (S, V)$ be the program where S is "$\mathbf{x} = M\mathbf{x}$;" and let $V = invar(P) = outvar(P) = \mathbf{x}$. Let also $\mathbf{x}_o$ be $out(P, \mathbf{x}_i)$, i.e., the output of P on input $\mathbf{x}_i$ and let

$$\mathbf{x}_p = M_h \mathbf{x}_p;$$

be a hiding of "$\mathbf{x} = M\mathbf{x}$;" with invertible matrices I, and J, variable sequence $\mathbf{y}$, and corresponding assignment padding matrix M_p. The output of "$\mathbf{x}_p = M_h \mathbf{x}_p$;" on input $J^-1(\mathbf{x}_i, \mathbf{y}_i)$ is equal to $I\Pi_{pp}(\mathbf{x}_o, \mathbf{y}_o)$.

```
Hide(a ∈ Z_m; input_x ∈ Z_m^l; M ∈ Z_m^l × Z_m^l; x ∈ V^l; k, l ∈ Z_m)
begin
    /* Hiding the Assignment Matrix */
    I = random invertible matrix in (Z_m^+)^(l + k) × (Z_m^+)^(l + k)
    y = sequence of k variables in V none of which is a variable of x.
    M_y = random matrix in (Z_m^+)^l × (Z_m^+)^l
    Π_r = Π_c = Π_p = random permutation of 1, ..., l + k
    M_xy = [ M_x   0  ]
           [ 0    M_y ]
    M_p = Π_pp(M_xy)
    M_h = I M_p I^{-1}

    /* Hiding the input */
    input_y = random value assignment of y
    input_xy = input_x input_y;
    input_p = Π_p(input_xy);
    input_h = I input_p
end

Recover(output_h ∈ Z_m^l; M_h ∈ Z_m^l × Z_m^l; k, l ∈ Z_m)
begin
    output_p = I^{-1} output_h
    output_xy = Π_p^{-1}(output_p)
    output_x = first l elements of output_xy
end
```

Figure 1: Hiding and recovering for piece-wise linear programs

Theorem 6.5 (Hidden For Statement Execution) Let $P = (S, V)$ be the program where S is "**for** (a) $\mathbf{x} = M\mathbf{x}$;" and let $V = invar(P) = outvar(P) = \mathbf{x}$. Let also $\mathbf{x}_o$ be $out(P, \mathbf{x}_i)$, and let

$$\mathbf{x}_p = M_h \mathbf{x}_p;$$

be a hiding of "$\mathbf{x} = M\mathbf{x}$;" with invertible matrix I, variable sequence $\mathbf{y}$, and corresponding assignment padding matrix M_p. The output of "**for** (a) $\mathbf{x}_p = M_h\mathbf{x}_p$;" on input $I(\mathbf{x}_i, \mathbf{y}_i)$ is equal to $I\Pi_{pp}(\mathbf{x}_o, \mathbf{y}_o)$.

Theorem 6.4 can be used to devise a hiding scheme for programs that consist of a loop with an assignment statement as shown in Figure 1.

6.2 Condition Transformation

Unlike the assignment and **for** statements, an **if** statement contains a condition which also has to be hidden. These conditions are generally in DNF form. Since a DNF condition can be converted into a sequence of non-DNF conditions ("**if**$(a \vee b)$ S;" can be rewritten as "**if**(a) S *else* **if**(b) S;" or "**if**(a) S; **if**$(\neg a \wedge b)$ S;"), we only need to hide conjunctive conditions. In this section we only consider hiding the individual conditions and do not consider hiding the structure of the program.

Definition 6.6 (Conjunctive Condition Hiding) The hiding of a conjunctive condition, $C^\wedge$, of the form "$C^\wedge = (M\mathbf{x} \leq 0)$", where M is a matrix of size $c \times l$, is an other conjunctive condition, $C^{\wedge\prime}$, of the form

$$C^{\wedge\prime} = (M'\mathbf{x}_p \leq 0), \qquad \text{where } M' = \begin{bmatrix} C & 0_{c\times k} \end{bmatrix} \Pi_p^{-1} I^{-1}, \text{and}$$

l is the size of $\mathbf{x}$ and $(l+k)$ is the size of $\mathbf{x}_p$.

Theorem 6.6 (Hidden If Statement Execution) Let $P = (S, V)$ be the program where S is "**if** $(C^{\wedge})$ $\mathbf{x} = M_1\mathbf{x}$ **else** $\mathbf{x} = M_2\mathbf{x}$;" and let $V = invar(P) = outvar(P) = \mathbf{x}$. Let also $\mathbf{x}_o$ be $out(P, \mathbf{x}_i)$, let

$$\mathbf{x}_p = M_{1,h}\mathbf{x}_p;$$

be a hiding of "$\mathbf{x} = M_1\mathbf{x}$;", and let

$$\mathbf{x}_p = M_{2,h}\mathbf{x}_p;$$

be a hiding of "$\mathbf{x} = M_2\mathbf{x}$;" with invertible matrices I and J, variable sequence $\mathbf{y}$, and corresponding assignment padding matrix M_p. The output of

$$\textbf{if } (C^{\wedge\prime})\ \mathbf{x}_p = M_{1,h}\mathbf{x}_p \textbf{ else } \mathbf{x}_p = M_{2,h}\mathbf{x}_p\ ;$$

on input $J^-1(\mathbf{x}_i, \mathbf{y}_i)$ is equal to $I\Pi_{pp}(\mathbf{x}_o, \mathbf{y}_o)$.

Note that it is important that both branches of the **if** statement be hidden with teh same pair of matrices I and J.

6.2.1 Piece-wise Linear Program Transformation

So far, we described random linear code hiding transformations that apply to individual statements. Using these building blocks, now, we describe the transformation of a whole piece-wise linear program. First, we introduce the concept of assignment blocks that we use to separate the structure of the program from the assignments contained within. This allows us to hide the assignments independent of the structure, using linear transformations and padding as described earlier. The structure of the program is hidden using separate techniques, which we do not address here.

Partitioning a program into Assignment Blocks To hide a piece-wise linear program, the program is divided into assignment blocks separated by conditions and loops. To each assignment block corresponds one assignment matrix. To make that feasible some preprocessing is needed. The assignment matrix is obtained by expanding each of the assignment in terms of the initial values of the variables. Below, we give an example which depicts the expansion of an assignment block and the corresponding assignment matrix:

$$\begin{array}{l} x = y + z; \\ z = x + y; \\ y = y + z; \end{array} \qquad \longrightarrow \qquad \begin{array}{l} x = y_0 + z_0; \\ z = 2 \times y_0 + z_0; \\ y = 3 \times y_0 + z_0; \end{array} \qquad A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 3 & 1 \\ 0 & 2 & 1 \end{bmatrix}.$$

Hiding a Piece-wise Linear Program To achieve hiding of a program which consists of multiple assignment blocks, it is necessary to transform the program into a canonical form. The canonical form looks as follows.

for (a, i);
  if $(C_{11})A_{11}$;
  elseif $(C_{12})A_{12}$;
  $\ldots$
  else A_{1k_1};

  if $(C_{21})A_{21}$;
  elseif $(C_{22})A_{22}$;
  $\ldots$
  else A_{2k_2};
    .
    .
    .
  if $(C_{l1})A_{l1}$;
  elseif $(C_{l2})A_{l2}$;
  $\ldots$
  else A_{lk_l};

Different hiding matrices are used for each if-else block on the condition that the last hiding matrix used for the last block is equal to the inverse of the first hiding matrix used for the first block. Also, we require that the same permutation is used for all blocks, but we allow different padding matrices to be used even for assignment matrices within a block. Then, we can prove by induction that with these transformations, using the "recover" function will yield the correct output of a program.

One important property of our approach is that, the transformations guarantee not only the hiding of the entire program, but also the hiding of individual assignment blocks, or in fact, any subsequence of the program.

In fact, we can formally prove that all the individual assignments of the program are completely hidden and that the best the host can do is to guess what the individual assignements are. Nevertheless, the host can still calculate the eigenvalues of the execution along any given path and these are equal to those of the original program. While the eigenvalues of a path are not expected to be relevant in a program that has many branching point, their leakage can be significant for some programs.

6.3 Piece-wise Linear Programs with Arrays

In many programs, there is a very large number of indexical variables. In high level programming languages such variables are represented with arrays. In the previous section we introduced piece-wise linear programs, in which all statements and conditions have constant matrices. We did not discuss how the proposed framework handle arrays. In this section we introduce a new class of programs, *locally linear programs*, that use arrays. This class includes many programs that use arrays, including mathematical operations such as convolution.

6.3.1 Locally Linear Programs

We are using matrices to represent assignment statements in programs that do not use arrays. However, using the matrix representation for assignments on arrays is not appropriate because in each iteration, a different subset of array locations is accessed; consequently, assignment and condition matrices are not constant from one iteration to an other. Figure 2 shows the change in the assignment matrices

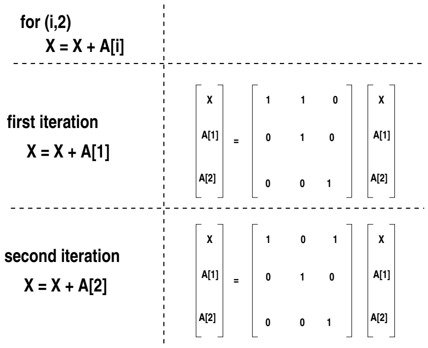

Figure 2: Change in the assignment matrices in different iterations of a for loop.

during the execution of a for loop (note that the for loop contais a loop index in addition to the number of iterations).

Note that the difference between the two matrices is only on the right hand corner of the first row. Consequently, both matrices can be expressed as a base matrix plus an other matrix which shifts at each iteration. Informally, in a locally linear program, a small locality in the non-base matrices are shifted in one direction at each iteration. In what follows, we formally define locality.

Definition 6.7 (Locally shifting Matrix Sequence) A sequence M of matrices $M(1), \ldots, M(d)$ of size $n \times n$ is locally shifting with locality l and period p if there exists a sequence of matrices $M_{l1}, \ldots M_{lp}$ such that for any $d \leq n$

- If $d+l \leq n$, then $M(d)[i][j] = 0$ if $j \notin [d, d+l]$ or $i \notin [d, d+l]$, and $M(d)[i][j] = M_{l(1+d \bmod p)}[i-d][j-d]$.
- If $d+l > n$, then $M(d)[i][j] = 0$ if $j \notin [1, d+l \; mod \; n] \cup [d, n]$ or $i \notin [1, d+l \; mod \; n] \cup [d, n]$, and $M(d)[i][j] = M_{l(1+d \bmod p)}[i-d][j-d]$ otherwise.

Definition 6.8 (Locality of a local matrix) Let $M(d)$ be an element in a locally shifting matrix sequence as defined above. The locality, $L(M(d))$ is defined as follows:

- If $d+l \leq n$, then $L(M(d)) = \{(i,j) \; : \; i \in [d, d+l] \; \wedge \; j \in [d, d+l]\}$.
- If $d+l > n$, then $L(M(d)) = \{(i,j) \; : \; i \in [1, d+l \; mod \; n] \cup [d, n] \; \wedge \; j \in [1, d+l \; mod \; n] \cup [d, n]\}$.

The base of a locality is equal to d.

A locally shifting matrix sequence must be given to the client in advance. For this purpose, we can use the distributivity property of linear transformations:

$$A(M(d) + N)A^{-1} = AM(d)A^{-1} + ANA^{-1},$$

where $M(d)$ is the base matrix, N is the shift matrix, and A is the random matrix used for hiding. The hiding matrix A of size $n \times n$ is such that $A[i,j] = A_l[1 + i \bmod l][1 + j \bmod l]$, where l is the size of the locality of the sequence M. Note that this approach is feasible only when the number of for loop iterations is very small. Also, this approach may be costly since the matrix size increases quadratically with the size of the arrays.

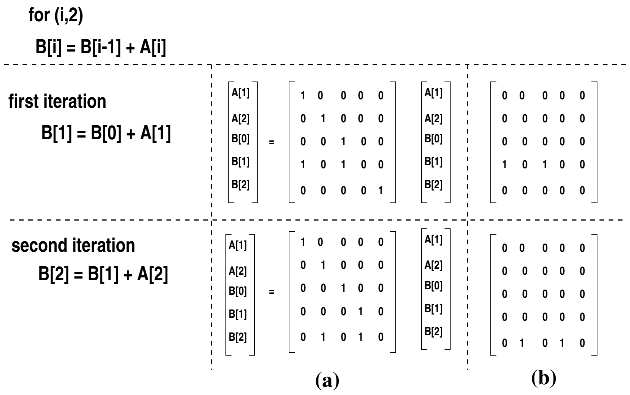

Figure 3: Variable sliding: (a) change in the assignment matrices in different iterations of the transformed for loop, (b) the shifting locality

Therefore, unless array sizes and for loop iterations are small, this approach is not efficient, and we need to use optimization techniques. For instance, if we use a special hiding matrix such that if we multiply the hiding matrix with a sequence of locally shifting matrices, then we obtain another sequence of hidden locally shifting matrices. Consequently, instead of providing the host with a base matrix and d very large shift matrices, we can provide it with $p \leq d$ (potentially smaller) hidden matrices. One such technique that we developed is called *variable sliding*, which we introduce next.

6.3.2 Variable Sliding for Locally Linear Programs

The reason why the hiding of the program given in Figure 2, i.e., `for (i,2) X = X + A[i];` is expensive is that the localities that correspond to the variable X and $A[i]$ are moving independently from each other: the locality of X stays the same, while the locality of $A[i]$ is shifted as i changes. Consequently, the whole matrix requires to be hidden in its entirety.

Note that, if we could make sure that all variables in a given statement are moving together, we could avoid this costly operation. Below, we explain this with an example:

Let us assume that we are given the for loop given in Figure 2. We can apply a transformation on this for loop that *slides variables* along with the shifting array elements:

```
B[0] = X;
for (i,2)
   B[i] = B[i-1] + A[i];
X = B[2];
```

The corresponding change in the assignment matrices is shown in Figure 3(a). Figure 3(b), on the other hand, shows only the shifting localities; all matrix entries that are not involved in the assignment is set to 0. We call this the locality shift matrix. Informally, given a shift matrix, S, of size $n \times n$, the assignment matrix, C, is found as follows:

$$S = \begin{bmatrix} 0 & 0 & 0 \\ 0 & L & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \longrightarrow \qquad \begin{bmatrix} I & 0 & 0 \\ 0 & L & 0 \\ 0 & 0 & I \end{bmatrix},$$

where L is an $l \times l$ locality and Is are diagonal matrices (that denote the implicit assignment statements for unaffected variables).

The advantage of a locality shift matrix of the above form, S, is that it can be hidden using a padding matrix, P, of size $n \times n$ of the form:

$$P = \begin{bmatrix} A_1 & 0 & 0 & \dots & 0 \\ 0 & A_2 & 0 & \dots & 0 \\ 0 & 0 & A_3 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & A_{\frac{n}{l}} \end{bmatrix},$$

where each A_i is a random matrix: whatever the location of L is in the original locality shift matrix, when S is multiplied with P and P^{-1}, it will lead into a hidden matrix, S_H, of the form:

$$S_H = \begin{bmatrix} 0 & 0 & 0 \\ 0 & L_H & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

where L_H is a matrix of size at most, $2l \times 2l$; the increase in the size is due to the different possible alignments of L and A_is in P. Since the matrix, S_H, consists mostly of 0s, the cost of the host using S_H during computations is very low, a function of l, which is independent of the number and the sizes of the arrays. Furthermore, since this locality can be described very cheaply, the size of the compiled code given to the host is also reduced significantly.

6.4 Tamper Resistance

So far, we have been focusing on the hiding transformations, In this section, we explain how we achieve tamper resistance; i,e., the owner's ability to assess the authenticity and correctness of the results of a computation. We require that detecting tampering with the results be done efficiently and reliably. There are two kinds of tampering that we are interested in protecting against:

- *Directed tampering* occurs when the host changes the returned values in such a way that the output results recovered by the owner are forced to take specific values.

- *Random tampering* occurs when the host randomly changes the returned results without knowing the effects of the tampering on the results recovered by the owner and in the hope that the owner will recover incorrect results.

In earlier work, we showed how tamper resistance can be achieved for the class of programs that calculate polynomials. The solution we devised protects against both forms of tampering.

The basic idea of tamper resistance is to enrich the program with *integrity code* that produces outputs related to the outputs of the program. The integrity code checks that the relationship between its outputs and the outputs of the program are satisfied. That is why we believe that some level of hiding is a prerequisite for tamper resistance. Nevertheless, hiding is not sufficient for tamper resistance because it does not protect against random tampering. To make these ideas clear, we give an example. In Figure 4, the added code calculates the square of the program output and the tamper test checks that the value calculated by the added code is indeed the square of the value calculated by the program. Tampering can occur if the host returns two values such that one is the square of the other. Hiding will make this task harder for the host because it does not know what variables to modify, but hiding *is not enough* to provide tamper resistance.

(a)	(b)	(c)
$y = x + 3x^2 + 5$	$y = x + 3x^2 + 5$ $y' = (x + 3x^2 + 5)^2$	**if** $(y^2 = y')$ success **else** failure

Figure 4: Tamper resistance: (a) Program; (b) program with added code; (c) tamper test

To provide tamper resistance for polynomials, we used a *dispersion* technique that guarantees that a change in one returned value, will affect *many* returned values and showed that the owner has a very small probability (impossible for all practical purposes) of tampering with the returned results.

7 Conclusions

In this paper, we described the efficient, provable code hiding and tamper resistance techniques we developed. These techniques provide the support needed for a mobile code to survive and strive in an open hostile environment. The work of Sander and Tschudin is the only work that we are aware of that tries to solve the same problem we are considering and uses a similar model of interaction. Unlike their work, we give a formal definition of code hiding and we develop techniques and prove them correct according to the definitions. An important contribution of our effort is the introduction and use of the formalism of standard cryptography in the context of code hiding and tamper resistance. We formally defined *code* and *structure* hiding and *tamper resistance* and we developed techniques to strongly hide programs that calculate polynomials.

We developed a tamper-resistant solution that hides a program that calculates a polynomial. We also developed techniques to hide various forms of piece-wise linear programs as described in the technical rational section. The techniques to hide piece-wise linear programs are the *only* techniques that we know of that can *provably* hide programs with loops and conditional statements.

We believe that a complete specification of the problem of hiding requires a consideration of the amount of knowledge the host has about the hidden program. While the definitions we propose are similar to traditional definitions of secure encryption techniques, we are unaware of any other code hiding work that evaluates the security of code hiding algorithms according to these measures.

It is clear from the description above that providing security can induce a certain amount of time and space overhead. In future work, we plan on studyng fundamental limitations on this overhead.

References

[1] Martín Abadi, Joan Feigenbaum, and Joe Kilian. On Hiding Information from an Oracle. *Journal of Computer and System Science*, 39:1, pages 21–50, August, 1989.

[2] David Aucsmith. Tamper resistant software: An implementation. In Ross Anderson, editor, *Information Hiding - Proceedings of the First International Workshop*, LNCS no. 1174, pages 317–333, 1996.

[3] D. Beaver, J. Feigenbaum, and V. Shoup. Hiding Instances in Zero-Knowledge Proof Systems. *Advances in Cryptology – CRYPTO '90*, pages 326–338, Springer-Verlag, 1990.

[4] Christian Collberg, Clark Thomborson, Douglas Low, Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, San Diego, CA, January, 1998.

[5] Christian Collberg, Clark Thomborson, Douglas Low, Breaking Abstractions and Unstructuring Data Structures. In *Proceeding of IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, 1998.

[6] Christian Collberg, Clark Thomborson. On the Limits of Software Watermarking. In *Proceedings of ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, 1999.

[7] Joan Feigenbaum. Encrypting Problem Instances, or, ..., Can You Take Advantage of Someone Without Having to Trust Him. *Advances in Cryptology – CRYPTO '85*, Springer-Verlag, 1985.

[8] Y. Minsky, R. van Renesse, F. B. Schneider, and S. D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the 1996 SIGOPS European Workshop*, July 1996.

[9] George C. Necula. Proof-carrying code. In *Proceedings of 24th ACM Symposium on Principles of Programming Languages*, pp. 106–119, Paris France, January, 1997.

[10] Tomas Sander and Christian F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. To appear in *Mobile Agent Security*, Springer Lecture Notes on Computer Science.

[11] F. B. Schneider. Towards Fault-Tolerant and Secure Agentry. Invited paper, in *Proceedings of 11th International Workshop on Distributed Algorithms*, Saarbrücken, Germany, Septmember 1997.

[12] Jan Vitek and Christian Tschudin (editors). *Mobile Object Systems: Towards the Programmable Internet.* LNCS np. 1222, Springer, 1997.