Information Flow Control for Location-based Services

Nishkam Ravi, Marco Gruteser* and Liviu Iftode
Department of Computer Science, Rutgers University
*WINLAB, ECE Department, Rutgers University
{nravi@cs.rutgers.edu, gruteser@winlab.rutgers.edu, iftode@cs.rutgers.edu}
Key words: Static program analysis, information-flow control, location privacy

Abstract

This paper presents a framework for preserving location privacy without affecting the quality of service. In this framework, the services migrate a piece of code to a trusted server that is assumed to have location information of all the interesting subjects. The code executes on the trusted server, reads location information and sends back results. We present Non-inference, an information-flow control model that guarantees that the code does not leak exact location information. We discuss the design, implementation and evaluation of a static program analysis technique that enforces non-inference for location based services.

1 Introduction

Improvements in positioning technology and better coverage of wireless networks make universal location tracking of people and objects increasingly viable. This enables a broad range of new applications, for example monitoring of traffic jams and road conditions through floating car data, point of interest queries, and alerts that notify you of a friend in close proximity. While technically feasible, a major concern remains maintaining user's location privacy [27].

Many of these Location Based Services (LBS) work with aggregate information derived from the position measurement, rather than recording the position measurements itself. For example, the traffic monitoring application would compute mean vehicle velocity on any give road segment and feed this data into navigation systems or control traffic management system such as ramp meters. Theoretically, such applications can be structured to preserve privacy by only revealing the aggregate data instead of personal location records. However, not knowing the detailed implementation of a service and assuming that they cannot blindly trust a given service provider, users have no way to decide whether a service preserves privacy.

One solution is a general trusted anonymization service that reduces spatiotemporal resolution until the data meets a k-anonymity [26, 13] constraint before it passes data to applications.[1] However, this reduction in resolution can lead to inferior quality of service. Different services require different levels of accuracy, thus for some services a generalized cloaking service may unnecessarily reduce the accuracy of the service.

In this paper, we propose a *service-specific* location privacy approach, centered on an information flow control analysis that verifies the privacy properties of custom data aggregation functions. Consider a scenario where a trusted location server maintains mobile subjects' position information. Location information consists of identity of the subject, its location, and the timestamp when the subject was present at that location. When an LBS provider needs location information, it can request to install an aggregation module on the trusted server. The module can access location records on the server and communicate aggregate results (e.g.,

[1] A subject is considered k-anonymous if it cannot be distinguished from at least k - 1 other subjects. For example, if the location information sent by a mobile subject is perturbed to replace the exact coordinates by a spatial interval, such that the locations of at least k - 1 other subjects belong to that interval, then the adversary cannot match the location of the subject to its identity without a certain amount of uncertainty. The uncertainty would increase with k, providing better privacy.

distance between two cars, density of vehicles in region, or mean velocity of vehicles on road segment) to the LBS provider. This eliminates the need to reveal raw location information to the LBS as the desired result is computed on the trusted server itself using *exact* location information. In order to preserve privacy, the trusted server needs to ensure that the data sent to the LBS is indeed aggregated and *location-safe*. That is, the LBS provider must be unable to deduce the location of any mobile subject from the received information. Since an aggregation module, due to malice or incompetence, may covertly leak information, this requires a detailed analysis of the aggregation module. This analysis can be performed either by the operator of the trusted location server or another trusted third party (like VeriSign) that guarantees correct behavior of the module. To enable the analysis and certification of a large number of different aggregation functions there is a need for tools to assist in verifying code for location-safety. In this context, we describe a system based on an information flow analysis, tailored to aggregation modules for time-series information such as location traces.

1.1 Contributions

The problem of verifying location-safety of an aggregation module can be regarded as an *information-flow control* problem. Information-flow control policies tend to impose restrictions on the manner in which sensitive data flows through a program/system. The standard information-flow control model called *Non-interference* [22, 5] requires that any possible variation in private data must not cause a variation in public data. That is, if the value of a public variable q depends on that of a private variable p then non-interference is violated. In effect, non-interference isolates private data from public data. In doing so, it guarantees that the publicly observable behavior of a system/program that does not reveal *anything* about its private behavior. However, data isolation is an extreme measure, and in many real applications it is not possible to isolate private data from public data. For example, in our case the migratory code computes results based on private location information. These results are transmitted back to the LBS and are public. Since private variables affect the value of public variables, it is not possible to enforce non-interference in this case. The question then is: Is it possible to envisage an information-flow control model that does not require data isolation and yet preserves privacy? Under what assumptions can such a model be realized?

In this paper, we propose a weaker model of information-flow control called *non-inference*. *Non-inference* requires that the adversary should not be able to *infer* the value of a private variable based on the values of public variable. In the most general case, it can be shown that non-inference is undecidable (we provide a proof in [23]). However, if we can assume that multiple executions of the untrusted code are independent, then non-inference is decidable (see proof in [23]). To understand this better, consider the following code snippet:

```
int f(int loc1, int loc2){
    int avg_loc = (loc1+loc2)/2;
    output(avg_loc);
}
```

Function f computes the average of two integers *loc1* and *loc2* (which are private), and stores it in a public variable *avg_loc*. It is easy to see that function f would violate *non-interference*, as the public variable *avg_loc* depends on the values of private variables *loc1* and *loc2*. This function would, however, satisfy *non-inference*, as the adversary cannot infer the value of either *loc1* or *loc2* from *avg_loc*. The adversary does get some information about the private variables, which is their average. Now consider another function g that returns the product of two integers:

```
int g(int loc1, int loc2){
    int mult_loc = loc1*loc2;
    output(mult_loc);
}
```

Function g also satisfies *non-inference*, as the adversary cannot infer the value of *loc1* or *loc2* from *mult_loc*. However, if the adversary knows both *avg_loc* and *mult_loc* it is easy to infer the values of *loc1* and *loc2* which are private variables.

Non-inference, therefore, allows information to flow from private variables to public variables, but in order to prohibit the adversary from learning the value of any private variable from public variables, it is necessary to assume that executions are independent. Execution independence requires that the results obtained from one execution should not aid the results obtained from another execution of the same or different application. This is usually the case for location-based applications. If *loc1* and *loc2*

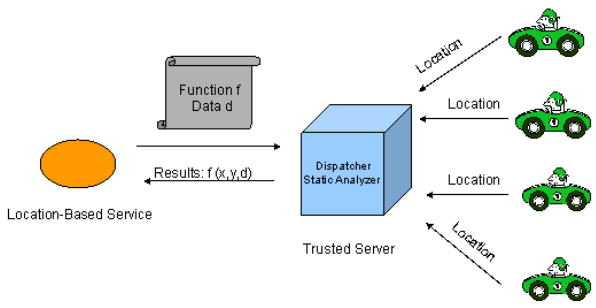

Figure 1. Framework

are x-coordinates of two moving vehicles, and g is executed after f, then the inputs *loc1* and *loc2* would be different for the two functions. The reason for this is that location changes with time. This makes the execution of g independent of the execution of f. We revisit these assumptions and describe the properties of *non-inference* in Section 4.

The key contributions of this paper are: (1) the concept of non-inference and an analysis of its theoretical properties, (2) a framework for the application of non-inference to service-specific location privacy, and (3) enforcement of non-inference for location based services using static program analysis. The remainder of the paper is organized as follows. In Section 2 we describe our model and give examples of a few applications that can benefit from our framework. In Section 3 we survey related work. Section 4 presents non-inference and its application to location-based services. Implementation details and evaluation results are presented in Section 5. We conclude in Section 6 with directions for future work.

2 Model and Example Applications

In our model, the location information of mobile nodes is maintained on a trusted server. An LBS that needs to compute a result based on location information sends a piece of code to the trusted server along with some data which is optional. The code executes on the trusted server, reads location information resident on the server, and the data sent by the LBS. It then computes some results and sends them back to the LBS. Without loss of generality, we assume that the mobile code is a single function encapsulated in a Java class. Note that a program with multiple functions can be reduced to a single function by inlining function bodies. We assume that all input and output variables are scalars and of the same type length. The type length is determined by the minimum number of bits required to store location information. This assumption is necessary in order to ensure that exact values of two variables cannot be stored in one variable, which is important for our solution. In our current implementation, we assume all variables to be of type *byte* (8-bit integers). The function is invoked from the *Dispatcher* class, which feeds the data sent by the LBS and the location information to the function. Before invoking the function, the *Dispatcher* class invokes the *StaticAnalyzer* that first checks to see if the function is pre-certified, if not it analyzes the function and determines if it satisfies non-inference. The *Dispatcher* also ensures that there is a minimal time gap between two code executions in order to guarantee execution independence as discussed before. Figure 1 shows the bird-eye view of our model.

Several traffic monitoring services can be enabled by aggregate functions that calculate density of vehicles, or average speed of vehicles in a region. Recently, Delcan technology has begun deployment of a system in Maryland that mines cell-phone data to determine traffic conditions such as jams and slowdowns [1]. By measuring the time of handoffs from cell to cell, the location and speed of a vehicle is calculated (assuming that the driver's phone is turned on). In our framework, Delcan would fit in as the untrusted LBS provider, and the cellular service provider would fit in as the trusted server which tracks the location of every cell-phone. Whenever Delcan needs location information, it would migrate a piece of code to the trusted server which would return aggregate information. Based on the aggregate information, Delcan would supply information about traffic conditions to the subscribers.

The framework proposed in this paper could be used to implement privacy-preserving geographic route discovery function for environments with long-lasting flows and relaxed scalability constraints. Let us assume that there is a *Routing Service* that provides next-hop information to nodes. Any node that wants to forward data to a destination location, invokes the Routing Service with the location of the destination, and requests for the address of the best next hop. The Routing Service migrates a piece of code that calculates distance between two nodes, to the trusted server which maintains loca-

tion information of all nodes. Through the distance function the Routing Service learns the distance of every node from the destination location. It can then supply next hop information to the requesting node, without learning the location information of any node. In addition to geographical routing, distance function can be used by applications such as media applications, streaming applications, or content sharing applications.

3 Related Work

3.1 Location Privacy

Early work on context privacy (which includes location) focused mostly on a policy-based approach [10, 15]. Policies could be of one of the following two kinds: those specified by the service providers and those specified by the users. The policies served as a mutual agreement on the manner in which data was to be collected, shared and used. The main problem with this method was that the enforcement of these policies was loosely defined. The users essentially had to trust the service providers for abiding by these policies.

This was followed by the application of k-anonymity [26] to location information using data perturbation techniques. Location information was depersonalized and perturbed before being forwarded to the LBS. This method is the state of the art in location privacy. It was first studied in [13]. The main drawback of this method is that it may lead to inferior quality of service where accurate location information is desired. Also, it uses a global value of k which is decided statically, where a smaller value of k could provide desired privacy levels without affecting quality of service significantly.

3.2 Information-flow control

Information-flow policies are end-to-end security policies that provide more precise control of information propagation than access control models. A large body of work exists in this field, most of which is purely theoretical in nature. Here we brief on the most noteworthy pieces of work in this area that have led to the evolution of this field.

The lattice model for multilevel security systems was first proposed by Denning et al [8]. In this model, objects are assigned different security levels, where objects can be files, segments or program variables depending on the level of the detail. The security levels are organized as a lattice. Information is allowed to flow only from low security levels to high security levels. This has been the universally accepted model for representing multilevel security systems ever since. A special case is when there are only two security levels: *public* and *private*.

State of the art in information-flow control is *Non-interference* [5, 22], which was introduced by Goguen and Meseguer in their seminar paper [22]. Intuitively, non-interference requires that high-security information does not affect the low-security observable behavior of the system. In other words, private data does not influence public data. Non-interference is fairly easy to enforce using language-based techniques [9, 17, 28, 18]. In particular, static program analysis has demonstrated advantages of little run-time overhead, the capabilities of managing implicit information flows, and provable guarantees. Jif [28, 18] is a popular static analyzer that enforces non-interference on annotated Java programs for multilevel security systems. Central to this implementation is the notion of a *principal*, which is an entity (e.g user, process, party) that can have confidentiality concern with respect to data. Principals can express ownership on data (i.e program variables) via *security labels* which are analogous to security levels in multilevel security systems. These security labels are used to track flow of information between program variables. Illegal information flows are prohibited using type analysis.

Non-interference, however, is a very strict requirement for most systems where private data often interferes with public data. Its applicability is therefore extremely limited. This was explicitly stated in [24]. Since then there has been research on relaxing non-interference. Giacobazzi et al proposed *Abstract Noninterference* [12], where they used abstract interpretations to generalize the notion of non-interference by making it parametric to what the attacker can analyze about the information flow. Many downgrading [2] scenarios can be formally characterized in this framework. However, this framework is mainly theoretical. To practically apply this theory in building program analysis tools, we need to design ways to express security policies and mechanisms to enforce them. *Approximate*

[2] Downgrading/declassification is a term used for lowering the security class of an object, e.g from private to public. This allows sensitive information to be leaked when necessary.

Non-interference [21] is a model in which the notion of non-interference is approximated in the sense that it allows for some exactly quantified leakage of information. This is characterized via a notion of process similarity. In this model, programs leaking more information are considered less secure. However, comparing the quantity of information leakage does not have direct sensible meanings in most situations.

In a recent work [16] Li and Zdancewic proposed a generalized framework of *downgrading policies*. A downgrading policy defines how and when the security level of a particular sensitive data entity can be downgraded to allow leakage when necessary. Using this framework, the user can specify downgrading policies which are enforced using a type system. The main contribution of this work is the formalization of a framework that enables downgrading. The real challenge is to define downgrading policies that are applicable to real systems.

Unlike non-interference and its derivatives, non-inference allows information to flow from private variables to public variables, but requires that the adversary does not infer value of any private variable from public variables. This makes it much more generally applicable. Non-inference certifies a *program execution* as opposed to a program, and therefore requires that multiple executions be independent.

3.3 Mobile Code Analysis

There has been some research on certifying untrusted mobile code for safe execution using both static analysis [20, 18, 7] and execution monitoring [11]. SASI [11] uses a reference monitor to observe execution of the untrusted code and aborts execution whenever a security policy is violated. In the Proof-Carrying Code (PCC) [20] approach the code carries a *formal proof* of its safety, which is verified by the receiver using a theorem prover prior to execution. Most of this research has focused on certifying *safety* of the code in terms of memory access and resource access.

4 Non-Inference

In this section, we define the problem of certifying program execution, discuss the assumptions and present the solution. We denote a program as a function f which takes inputs $i_1, i_2, ..i_n$ and produces outputs $o_1, o_2....o_k$. Let P denote the set of private input variables and Q the set of public input variables. All output variables are assumed to be public. Function f satisfies non-inference if and only if $\neg(\exists i_k \in P|\{f, o_1, o_2....o_k\} \rightarrow i_k)$. Inference is denoted by the symbol $\rightarrow$. Informally speaking, a program satisfies non-inference if an adversary cannot infer the exact value of a private input variable from the output variables and the program code.

The first question is: Is it possible to decide if a program satisfies non-inference? It can be shown that in the most general case (when no assumptions are made about the program or the capabilities of the adversary), non-inference is undecidable. (We provide a proof in [23]). However, non-inference is decidable if we can assume that multiple executions of untrusted code are independent of each other. (We prove this in [23]). To understand this better, consider the following code snippet:

```
int h(int x1, int x2, int i){
    int out=0;
    if(i == 1){
       out = (x1+x2)/2;
       output(out);
    }else{
       out = x1*x2;
       output(out);
}
```

Here *x1* and *x2* are private variables and i is a public variable. Function h returns the average of *x1* and *x2* if $i = 1$, otherwise it returns the product of the two variables. Body of function h and the value of the input variable i are sent by the untrusted service. From one execution of h, the untrusted service learns either the product or the average. It cannot infer the value of *x1* or it x2 from *out*. Now consider two different executions of h with i=1 and i=2 respectively. The first execution returns average of *x1* and *x2*, while the second execution returns product of *x1* and *x2*. By knowing both average and product, the adversary can infer the values of both *x1* and *x2*.

The scope of non-inference is limited to one single execution of the program. It is not possible to enforce non-inference across different executions, unless they are independent of each other. This restricts the applicability of non-inference to certain kind of applications. Specifically, those that satisfy independent executions. In this example, if *x1* and *x2* are x-coordinates of two moving vehicles, and

there is a time gap between the two executions of h, then the values taken by *x1* and *x2* would be different for the two executions. This makes the two executions of h independent of each other. Non-inference is therefore applicable to location based applications.

4.1 Deciding Non-Inference for Location Based Applications

We use static program analysis to decide non-inference. The analysis consists of two phases. In the first phase, global data-flow analysis is used to construct information-flow relations between the variables(V) and expressions(E) of the code T under examination. From these information-flow relations, dependency information between private input variables ($P \in V$) and output variables ($O \in V$) is derived and stored in a matrix M. In the second phase, we decide if the information-flow relations stored in the matrix satisfy non-inference. For this, the information-flow relations are treated as linear equations and the theory of solvability of linear equations is applied. This is the main idea used in deciding non-inference.

4.1.1 Information-Flow Relations

We define three information-flow relations [6]:
R_1 from V to E
R_2 from E to V
R_3 from, V to V.

$R_1(v, e)$ can be interpreted as "the value of variable v on entry to T *may be used* in the evaluation of expression e in T". For example, the code S:
if $(a > 10)$ then $(m = a + b;\ n = m * a)$ else $k = b$
contains three expressions: $\{a > 10, a + b, m * a\}$. The value of variable a on entry to this code may be used in evaluating all the three expressions, while the entry value of b may be used in evaluating only $(a + b)$. The entry value of m will not be used for evaluating any of the three expressions. For this code, we have: $R_1(a, a > 10)$, $R_1(a, a + b)$, $R_1(a, m * a)$ and $R_1(b, a + b)$.

$R_2(e, v)$ can be interpreted as "the value of expression e in T *may be used* in obtaining the exit value of variable v". In the previous example, both expressions $(a > 10)$ and $(a + b)$ may be used in obtaining the exit value of m. Similarly, all the three expressions $\{a > 10, a + b, m * a\}$ may be used in obtaining the exit value of n. For S, we have $R_2(a > 10, m)$, $R_2(a + b, m)$, $R_2(a > 10, n)$, $R_2(a + b, n)$, $R_2(m * a, n)$.

$R_3(v_1, v_2)$ can be in interpreted as "the entry value of variable v_1 may be used in obtaining the exit value of variable v_2 in T". $R_3(v_1, v_2)$ implies that either: (i) the entry value of v_1 may be used in obtaining value of some expression e which in turn may be used in obtaining the exit value of v_2 in T, or (ii) there is an assignment statement $v_2 = v_1$ which is preserved in T. An assignment statement $x = y$ is said to be *preserved* in T, if there exists a path in T that does not reassign x.

R_3 can be expressed in terms of R_1 and R_2 as follows: $R_3 = R_1R_2 \cup A$, where $A = \{(v_1, v_2)$, s.t there is an assignment statement $v_2 = v_1$ that is preserved in $T\}$. In example S, R_1R_2=$\{(a, m), (a, n), (b, m), (b, n)\}$, $A = \{(b, k)\}$, and $R_3 = R_1R_2 \cup A = \{(a, m), (a, n), (b, m), (b, n), (b, k)\}$.

4.1.2 Construction of information-flow relations

The information-flow relations R_1 and R_2 are constructed using *use-def* [4] and *def-use* [4] analyses. *Use-def* analysis is used to create "use-definition chains" or "ud-chains". "Use-definition chains" are lists, for each use of a variable, of all the definitions that *reach* that use. We say a variable is *used* at statement s if its r-value may be required. For example, the statement: $(m = a+b)$, contains a use of a, a use of b and a definition of m. In the sequence of statements:
$a = 5; m = c + d; m = a + b; p = m + n; q = a + t;$
definition d_1: $(m = c + d)$ is overwritten by definition d_2:$(m = a + b)$. We say that definition d_2 *reaches* the use of m in statement $(p = m + n)$.

Def-use analysis is used to create "definition-use chains" or "du-chains". The du-chaining problem is to compute for a definition d of variable x, the set of uses s of x such that there is a path from d to s that does not redefine x. In the sequence of statements:
$a = 5; m = c + d; m = a + b; p = m + n$
definition d_2:$(m = a + b)$ reaches the use of m in $(p = m+n)$. Similarly, definition d_3:$(a = 5)$ reaches the use of a in statement $(m = a + b)$, and the use of a in statement $(q = a + t)$.

By taking transitive closures of du-chains and ud-chains, we construct relations R_1 and R_2. From R_1, R_2, and A we construct relation R_3. Matrix M is constructed from R_3 for input and output variables. For simplicity sake, readers can assume that

```
int f(byte x1, byte y1, byte x2, byte y2, int k){
       byte x, y, dist, avg_x, avg_y;
       x = (x2 - x1)^2;
       y = (y2 - y1)^2;
       dist = sqrt(x + y);
       output(dist);
       if(k > 100){
           avg_x = (x1 + x2)/2;
           avg_y = (y1 + y2)/2;
           output(avg_x);
           output(avg_y);
      }
}
```

Figure 2. Function that calculates distance between two cars and average values of coordinates

$M = R_3(P, O)$. While constructing M, we also take into account the *path* information (i.e which expression or assignment statement was responsible in establishing the dependency between a particular output and input variable) which is provided by relations R_1 and R_2. This is illustrated in Section 5, through a case study (*IfAttack*).

Figure 2 is the example of a function that calculates the exact distance between two cars. Variables $x1, y1, x2, y2$ are private input variables, k is a public input variable, $dist, avg\_x, avg\_y$ are output variables. (x_1, y_1) is the location of the first car and (x_2, y_2) is the location of the second car. The function also computes the average value of coordinates if the value of k is greater than 100. The information-flow relations for this function and the matrix M are given in Figure 3. Note that $R_3 = R_1 R_2 \cup A$ and $M = R_3(P, O)$.

4.1.3 Solving information-flow relations

Given information-flow relations, how do we decide if the program satisfies non-inference? Our analysis [23] shows that although non-inference is decidable in the case of independent program executions, it is not clear if a *generic* polynomial-time solution is possible. We solve this problem in context of location privacy, where it is reasonable to assume that all input and output variables are 8-bit integers. We treat information-flow relations as linear equations. It can be shown that a program satisfies non-inference, if the system of linear equations given by: $M^T P = O$, and all its subsystems are unsolvable. P is the set of private input variables, O is

$P = \{x1, y1, x2, y2\}, O = \{dist, avg\_x, avg\_y\}$
$V = \{x1, y1, x2, y2, k, x, y, dist, avg\_x, avg\_y\}$
$E = \{(x2 - x1)^2, (y2 - y1)^2, sqrt(x + y), (dist > k), (x1 + x2)/2, (y1 + y2)/2\}$
$A = \{I\}$, I represents Identity: $(v = v)$

$$R_1 = V \times E = \begin{bmatrix} 1&0&1&0&1&0 \\ 0&1&1&0&0&1 \\ 1&0&1&0&1&0 \\ 0&1&1&0&0&1 \\ 0&0&0&1&1&1 \\ 0&0&1&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{bmatrix}$$

$$R_2 = E \times V = \begin{bmatrix} 0&0&0&0&0&1&0&1&0&0 \\ 0&0&0&0&0&0&1&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&1 \\ 0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

$$R_3 = V \times V = \begin{bmatrix} 1&0&0&0&0&1&0&1&1&0 \\ 0&1&0&0&0&0&1&1&0&1 \\ 0&0&1&0&0&1&0&1&1&0 \\ 0&0&0&1&0&0&1&1&0&1 \\ 0&0&0&0&1&0&0&0&1&1 \\ 0&0&0&0&0&1&0&1&0&0 \\ 0&0&0&0&0&0&1&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

$$M = P \times O = \begin{bmatrix} 1&1&0 \\ 1&0&1 \\ 1&1&0 \\ 1&0&1 \end{bmatrix}$$

Figure 3. Information-flow relations for the distance example

the set of output variables and M^T is the transpose of matrix M.

We briefly recapitulate the theory of solvability of linear equations. A system of linear equations given by $AX = B$, has a solution only if : rank(A) = rank($[AB]$) = N, where A is M-by-N, X is N-by-1 and B is M-by-1. The notation $[AB]$ means that B is appended to A as an additional column. X is the matrix of unknowns, A is the coefficient matrix, and B is the right-hand-side matrix. The rank of a matrix denotes the number of independent rows in the matrix and hence the number of independent equations in the set. To get a solution, the rank of the coefficient matrix A should be equal to the number of unknowns. If all the equations in the set are assumed to be independent, it suffices to check if $N = M$.

We represent dependency between private input variables P and output variables O as a set of linear equations: $M^T P = O$ (as shown in Figure 4). The intuition behind this representation is as following:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \end{bmatrix} = \begin{bmatrix} dist \\ avg\_x \\ avg\_y \end{bmatrix}$$

$Rank(M^T) = 3 < (|P| = 4)$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \end{bmatrix} = \begin{bmatrix} dist \\ avg\_x \end{bmatrix}$$

$Rank(M_1^T) = 2 < (|P| = 4)$

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \end{bmatrix} = \begin{bmatrix} avg\_x \\ avg\_y \end{bmatrix}$$

$Rank(M_2^T) = 2 < (|P| = 4)$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \end{bmatrix} = \begin{bmatrix} dist \\ avg\_y \end{bmatrix}$$

$Rank(M_3^T) = 2 < (|P| = 4)$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \end{bmatrix} = \begin{bmatrix} dist \end{bmatrix}$$

$Rank(M_4^T) = 1 < (|P| = 4)$

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} avg\_x \end{bmatrix}$$

$Rank(M_5^T) = 1 < (|P_1| = 2)$

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} y1 \\ y2 \end{bmatrix} = \begin{bmatrix} avg\_y \end{bmatrix}$$

$Rank(M_6^T) = 1 < (|P_2| = 2)$

Figure 4. Linear Equations for the distance example

let $p_1, p_2, .., p_k \in P$ be the set of private input variables for which, $M(p_i, o) = 1$, then it can be said that $o = f(p_1, p_2, ..p_k)$. In other words, an output variable can be represented as a function of all the private input variables that may affect its value. Public input variables are treated as constants, as they are known to the adversary. The simplest representation of a function is a linear equation. In reality, the function may be a higher-order equation involving complex operations. By adopting linear equation as the representation of all the functions, we guarantee a conservative analysis. If the system of linear equations cannot be solved, then values of private input variables (which are the unknowns in these equations) cannot be inferred from the values of output variables. However, since our analysis is conservative, we may have false negatives. That is, if a system of equations can be solved then it does not necessarily mean that the program violates non-inference. But our analysis will reject such a program.

We assume that all the equations are independent. Let R be the number of rows of M^T and C be the number of columns. We check if $R < C$. We carry out this check for all the matrices that can be derived from M^T by deleting one or more rows of M^T. This is because we want to know if the value of *any* input variable can be inferred from output variables. In other words, we want to know if any subset of the linear equations can be solved. If the check is satisfied for M^T and all its sub-matrices, the program satisfies non-inference.

Figure 4 shows the system of equations $M^T P = O$ for the example program of Figure 2, and all the subsystems that can be derived from M^T. The check $R < C$ is satisfied by all the subsystems. The example program therefore satisfies non-inference.

5 Implementation and Evaluation

We have implemented a static analyzer that decides non-inference for Java programs. The implementation was done using Soot 2.2.1 [3] and Indus 0.7 [2]. Soot provides an API for analyzing and instrumenting Java bytecode. Indus provides an API for data-flow analysis. Our current implementation does not support inter-procedural analysis, and assumes that input and output variables are 8-bit integers.

In absence of real applications that make use of location, we evaluated our analyzer on a self-written benchmark of Java programs. Our benchmark consists of simple location based applications such as *Distance* (which returns distance between two cars), *Density* (which returns number of vehicles in a given region), *Speed* (which returns average speed of cars in a given region), *Average* (which returns average value x and y coordinates of the vehicles in the database). Our benchmark also includes some standard applications and attacks, such as *PasswordChecker* [25, 19], *Wallet* [25], *WalletAttack* [25], *AverageAttack* [25]. These are in addition to the several microscopic Java programs that were used in the testing of the implementation.

We try to answer the following questions while evaluating our analyzer:

- What kind of applications would benefit from non-inference?
- How bad are false-negatives?
- Can there be false-positives?

- What is the average running time of our analysis?

Case Study 1: *AverageAttack [25]*
Suppose x_1,..x_n stores the x-coordinates of n vehicles (which is private). The average x-coordinate computation is intended to release the average but no other information about x_1,..x_n:
Average $= (x_1 + .. + x_n)/n$; output($Average$)
It is possible to formulate a laundering attack on the *average* program that leaks the x-coordinate of vehicle i:

$x_1 = x_i$;..$x_n = x_i$;
Average $= (x_1 + .. + x_n)/n$; output($Average$)

The system of linear equations for *AverageAttack* given by $M^T P = O$ is:

$$\begin{bmatrix} 0 & .. & 0 & 1 & 0 & .. & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ .. \\ x_i \\ .. \\ x_n \end{bmatrix} = \begin{bmatrix} Average \end{bmatrix}$$

which is reduced to:

$$\begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} x_i \end{bmatrix} = \begin{bmatrix} Average \end{bmatrix}$$

$P_1 = \{x_i\}$, $Rank(M^T) = 1 = (|P_1| = 1)$
This system of equations is solvable. Therefore, *AverageAttack* does not satisfy non-inference and is rejected by our analyzer.

Case Study 2: *Wallet and WalletAttack [25]*
Consider an electronic shopping scenario. Suppose p stores the amount of money in a customer's electronic wallet (which is private), q stores the amount of money already spent (which is public), and c stores the cost of item to be purchased (which is public). The following code snippet(*Wallet*) checks if the amount of money in the wallet is sufficient and, if so, transfers the amount c from the wallet to the spent-so-far variable q, and outputs q:

if $(p > c)$ then $(p = p - c; q = q + c)$; output(q)

The system of linear equations for *Wallet* given by $M^T P = O$ is:

$$\begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} p \end{bmatrix} = \begin{bmatrix} q \end{bmatrix}$$

$$Rank(M^T) = 1 = (|P| = 1)$$

This system of linear equations is solvable. Therefore, *Wallet* is rejected by our analyzer. However, it is easy to see that *Wallet* satisfies non-inference as the adversary cannot learn the value of p from q. This is an example of an application where our analyzer reports a false-negative. The reason for this is that the expression $(p > c)$ *may* affect the value of q in statement $q = q + c$. Therefore, $R_2(p > c, q) = 1$. We have $R_1(p, p \geq c) = 1$. From here, $R_3(p, q) = 1$. Therefore our analysis assumes dependency between variables q and p. There is indeed an *implicit* flow of information from p to q. It is safer to assume that the adversary may be able to infer the value of q from p, although in this particular example it cannot. In general: *implicit information-flows* may result in false-negatives.

Now, we give an example of why it is important to have an analysis that is *conservative* and may occasionally report false-negatives. Consider the following code snippet (*WalletAttack*):

```
n = length(p)
while(n >= 0){
  c = 2^{n - 1}
  if(p > c){
    p = p - c;
    q = q + c;
    n = n - 1;
 }
}
output(q)
```

This code snippet leaks the value of p bit-by-bit to q. Our analyzer is able to detect it. The system of linear equations for *WalletAttack* given by $M^T P = O$ is:

$$\begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} p \end{bmatrix} = \begin{bmatrix} q \end{bmatrix}$$

$$Rank(M^T) = 1 = (|P| = 1)$$

This system of linear equations is solvable. *WalletAttack* is, therefore, rejected by our analyzer.

Case Study 3: *PasswordChecker [19, 25]*
Consider UNIX-style password checking where the system database stores hashes of password-salt pairs. *Salt* is a publicly readable string stored in the database for each user id, as a protection against dictionary attacks. For a successful login, a user is required to provide a password such that the hash of the password and salt matches the hash from the database. The following code snippet captures this functionality:

```
byte check(byte username, byte password){
  byte match =0;
  for(i = 0; i < database.length; i++){
    if(hash(username, password)
       == hash(salts[i], passwords[i])){
       match = 1;
       break;
    }
  }
  output(match);
}
```

This commonly used program is rejected by non-interference, as the value of public boolean variable *match* depends on private variables. It is easy to see that this program satisfies non-inference(as the username/password cannot be inferred from the binary output). Our analyzer accepts this program.

Case Study 4: *IfAttack*
Consider the following code snippet:

```
void fun(byte a, byte b, int i){
    byte out = 0;
    if(i >= 1){
      out = a;
    }else{
      out = a + b;
    }
    output(out)
}
```

Variables a and b are private input variables, variable i is a public input variable. If $i < 1$ then the output variable *out* leaks the value of input variable a, otherwise not. This program does not satisfy non-inference. Our analyzer is able to detect this. This is because while constructing matrix M from R_3, we use path information. For this code snippet,

$$R_3 = P \times O = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

The system of linear equations $M^T P = O$, is:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} out \\ out \end{bmatrix}$$

Note that we have two equations corresponding to the two paths. Path information is available from relations R_1, R_2 and A. A subsystem of this system is given by:

$$\begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} a \end{bmatrix} = \begin{bmatrix} out \end{bmatrix}$$

which is solvable, thereby violating non-inference.

5.1 Qualitative Analysis

What kind of applications would benefit from non-inference? *PasswordChecker* is a simple example of a commonly-used application that does not satisfy non-interference but satisfies non-inference. *Distance* (which calculates distances between two cars) is a function that satisfies non-inference, and whose quality would suffer if spatial/temporal cloaking is used. As described before, Geographical Routing Service can make use of *Distance*. Similarly, functions such as *Density* (which calculates density of vehicles in a region) or *Speed* (which calculates average speed of vehicles in a region) satisfy non-inference and can benefit from our framework. Traffic survey applications can make use of these functions. We have tested all these functions with our analyzer. Note that these are all examples of code-snippets that compute a result based on location information. These will be part of bigger applications that would run on the LBS side (such as Geographical Routing Service, or Traffic Information Service). In general, non-inference can be applied to services that need to use aggregate results based on the the locations of multiple subjects. Many of these applications can work with temporal/spatial cloaking as well, with compromised quality of service. While Spatial/temporal cloaking is more generally applicable, our framework is more suitable for applications that need fine grained location information. Non-inference is applicable to sensor data privacy in general. Location is an example of sensor data.

How bad are false-negatives? Through the *Wallet* example we showed that false-negatives are possible, as our analysis is conservative. In general, *implicit* information-flows can lead to false-negatives being reported. At the same time, through the *WalletAttack* example we showed why it is better to have a conservative analysis that occasionally reports false-negatives, as opposed to one that does not and can be attacked. Most of the applications which satisfy non-inference but are rejected by our analyzer, can be rewritten with minor syntactic modification to satisfy our analyzer.

Can there be false positives? We gave examples of a few well-known attacks (e.g *AverageAttack*, *WalletAttack*) that can break non-interference with declassification. These attacks are detected by our analyzer and rejected. Theoretically, we can show that it is not possible to launch attacks against

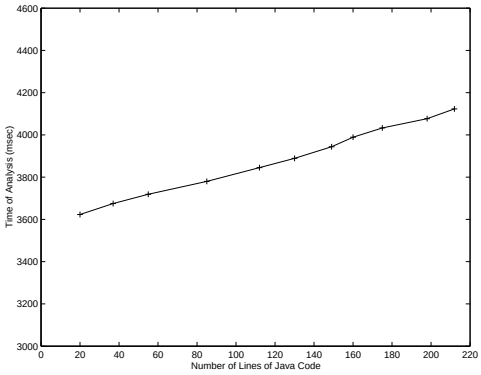

Figure 5. Running time of the static analyzer

our analyzer. Here we sketch the proof idea: let $p_1, p_2, .., p_k \in P$ be the set of private input variables for which, $M(p_i, o) = 1$, then it can be said that $o = f(p_1, p_2, ..p_k)$. In other words, an output variable can be represented as a function of all the private input variables that may affect its value. Public input variables are treated as constants, as they are known to the adversary. The simplest representation of a function is a linear equation. In reality, the function may be a higher-order equation involving complex operations. By adopting linear equation as the representation of all the functions, we guarantee a conservative analysis. It can therefore be said that if a system of linear equations cannot be solved, then values of private input variables (which are the unknowns in these equations) cannot be inferred from the values of output variables. However, the converse does not hold.

What is the average running time of our analysis? The experiments were carried out on an IBM ThinkPad R51 with 1.5GHz Intel processor and 256MB RAM, running the Linux operating system. For the benchmark consisting of 11 Java programs, the running time of our analyzer ranges between 3.6 and 4.2 seconds (as shown in Figure 5). Typically, the untrusted code would be some kind of an aggregation function which would be at most a few hundred lines of code. Our method of constructing information-flow relations has a worst case time complexity of $O(|E|^2 \times |V|^3)$ (where E is the set of expressions and V is the set of variables). The worst case time complexity of deciding solvability of information-flow relations is $O(|O| \times |P|^3)$ (where $O \in V$ is the set of output variables and $P \in V$ is the set of private input variables). The total worst case time complexity of our analysis is $O((|E|^2+|O|)\times|V|^3)$. $|V|$ and $|E|$ would be directly proportional to the code size. Figure 5 shows that the time of analysis increases with the size of code. There is a constant load time for the analyzer which is around 3500msec.

5.2 Discussion

Our analysis guarantees that the adversary does not learn exact location information. Currently, we do not provide guarantees on how many bits of a private input variable can be inferred from output variables. By crafting sophisticated attacks, the adversary can get partial information about the location. Partial information leakage, however, is a less serious concern for location privacy. It can be argued that hiding even one bit of location information can guarantee a high degree of uncertainty, as the adversary would not be able to distinguish a vehicle from other vehicles around it (similar to spatial cloaking in k-anonymity). Through our information flow analysis, we are able to successfully defend against the common class of attacks which involve complete information leakage, thereby raising the bar for the adversary.

The verification tool developed by us can be combined with manual analysis to prevent even partial information leakage. The tool would then assist the manual analyzer in identifying information leakage attacks, in the same way as a debugger assists the application developer in writing robust code. In order to develop an automatic tool for detecting partial information leakage, we need to combine the theory of abstract interpretation [12] with non-inference. Once non-inference is guaranteed, abstract interpretation can be used to quantify the amount of information leakage. We leave this for future work.

6 Conclusions and Future Work

We presented a new model for information-flow control called non-inference. Non-inference allows public data to be derived from private data but not vice versa. We discussed the theoretical implications of non-inference. We showed that non-inference is undecidable in general, but decidable for applications where multiple executions are inde-

pendent. We showed how it can be enforced *conservatively* using static analysis. The main idea is to apply the theory of solvability of linear equations to information-flow relations derived by data-flow analysis. We implemented a static analyzer that decides non-inference for Java programs and tested it on a benchmark. We discussed a framework in which non-inference can be applied to preserve location privacy without affecting quality of service.

If a program sent by an untrusted LBS violates non-inference, it could be because of one of the following two reasons: (i) the LBS is malicious and intends to learn private location information, or (ii) the program violates non-inference *unintentionally*. In this paper we have only looked at case (i). An interesting extension of this work would be to implement a static analyzer that is *constructive*. That is, if a program violates non-inference, it points out opportunities for modification in the program so that non-inference can be satisfied without affecting semantics. This can be extremely helpful for LBS providers in developing applications that satisfy non-inference.

References

[1] Cell phones to monitor traffic flow. http://yro.slashdot.org/article.pl.

[2] Indus. http://indus.projects.cis.ksu.edu/.

[3] Soot: A java optimization framework. http://www.sable.mcgill.ca/soot/.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[5] D. Bell and L. LaPadula. Secure computer system: Unified exposition and multics interpretation. *Technical Report ESD-TR-75-306, Electronics Systems Division, Mitre Corp.*, 1975.

[6] J.-F. Bergeretti and B. A. Carre. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 1985.

[7] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[8] D. E. Denning. A lattice model of secure information flow. *Commun. of the ACM*, 19(5), 1976.

[9] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. of the ACM*, 20(7), 1977.

[10] S. Duri, M. Gruteser, X. Liu, P. Moskowitz, R. Perez, M. Singh, and J.-M. Tang. Framework for security and privacy in automotive telematics. In *WMC '02: Proceedings of the 2nd international workshop on Mobile commerce*, pages 25–32. ACM Press, 2002.

[11] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, 2000.

[12] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004.

[13] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *MobiSys*, 2003.

[14] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8), 1976.

[15] X. Jiang and J. A. Landay. Modeling privacy control in context-aware systems. *IEEE Pervasive Computing*, 1(3), 2002.

[16] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. *SIGPLAN Not.*, 40(1), 2005.

[17] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, 1999.

[18] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.

[19] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW '04: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 2004.

[20] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, 1997.

[21] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, 2002.

[22] S. policies and security models. J.a. goguen and j. meseguer. In *In Proc. IEEE Symposium on Security and Privacy*, 1982.

[23] N. Ravi, M. Gruteser, and L. Iftode. Non-inference: A service-specific solution for location privacy. Technical Report DCS-TR-595, Rutgers University, 2005.

[24] P. Ryan, J. McLean, J. Millen, and V. Gligor. Noninterference, who needs it? In *In Proceedings of the 14th IEEE Computer Security Foundations Workshop*, 2001.

[25] A. Sabelfeld and A. Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security (ISSS'03)*, 2003.

[26] L. Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5), 2002.

[27] J. Warrior, E. McHenry, and K. McGee. They know where you are. *IEEE Spectrum*, July 2003.

[28] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Symposium on Operating Systems Principles*, 2001.

Appendix A

Model of Protection Systems: For sake of completeness, we first introduce the protection system as described in [14], which consists of the following parts:

- a finite set of generic rights R
- a finite set C of commands of the form:

```
command c(X1, X2, X3,...Xk){
   if r1 in (Xs1, Xo1) and
      r2 in (Xs2, Xo2) and
      ...
      rm in (Xsm, Xom)
   then
      op1
      op2
      ..
      opn
end
```

Where op_i is one of the primitive operations
enter r into (X_s, X_o)
delete r from (X_s, X_o)
create subject X_s
create object X_o
destroy subject X_s
destroy object X_o

Also, $r_1, r_2, ...r_m$ are generic rights $s_1, s_2, ...s_m$ and $o_1, o_2, ...o_m$ are integers.

A configuration of a protection system is a triple (S, O, P), where S is the set of subjects, O is the set of objects, $S \in O$, and P is an access matrix, with a row for every subject in S and a column for every object in O. $P[s, o]$ is the set of rights that subject s has over object o. An operation such as *enter r into* X_s, X_o) will enter right r in the access matrix at position (x_s, x_o), where x_s and x_o are actual parameters corresponding to formals X_s and X_o. An example of subjects and objects would be processes and files respectively.

Transition from a configuration $Q = (S, O, P)$ to another configuration $Q' = (S', O', P')$ under the execution of an operation *op*, is represented as $Q \vdash_{op} Q'$. Reachability of Q' from Q is represented as $Q \vdash_* Q'$.

Safety: Given a protection system, a command $c(X_1, X_2, ...X_k)$ is said to *leak a generic right r from configuration* $Q = (S, O, P)$ if c, when run on Q can enter right r into a cell of the access matrix which previously did not contain r.

An initial configuration Q_0 is said to be *unsafe* for r (or *leaks* r) if there is a configuration Q and a command c such that
(1) $Q_0 \vdash_* Q$ and
(2) c leaks r from Q

Q_0 is safe for r if Q_0 is not unsafe for r.

In [14] Harrison et al prove that *it is undecidable whether a given configuration of a given protection system is safe for a generic right r.* In other words *safety problem is undecidable.* In addition, they prove that *the safety problem without the create primitive is decidable and complete in polynomial space.* It is, however, not clear if a polynomial time solution is possible.

Complexity of Non-inference: We will show that the *safety problem*(as described in the previous section) can be reduced to the problem of deciding non-inference. Through this, we prove that non-inference is undecidable. In addition, we reduce the problem of deciding non-inference for single execution of a program to the *safety problem without create primitive.* Through this, we prove that non-inference is decidable for independent executions.

Theorem: *Non-inference for an arbitrary program is undecidable.*

Proof: We reduce the safety problem to the problem of deciding non-inference. Given an initial configuration $Q = (S, O, P)$, we create a program M with a set of variables V such that $V = S \cup O - \{S_1, S_2\}$. Input variables $V_{input} = \{v|(S_1, v) = r\}$, where (x, y) represents the cell in matrix P corresponding to row of x and the column of y. Output variables $V_{output} = \{v|(S_2, v) = r\}$. Intuitively, S_1 can be looked upon as a subject that has rights over all input variables and S_2 as a subject that has rights over all output variables. All input variables in set S are labeled private, while those in O are labeled public. The statements of M are created by converting each row of the access matrix into an assignment statement. The assignment statement corresponding to the row for subject S_i is $S_i = \{\oplus\{S_j|P_{ij} = r\}\} \oplus \{\oplus\{O_k|P_{ik} = r\}\}$, where $\oplus$ corresponds to the concatenation operator.

M when executed, would create a dependency matrix for its variables which will correspond to the access matrix P. To understand this, note than an assignment statement in M of the form $v_1 = v_2$ essentially corresponds to a right r in the cell (v_1, v_2) of the matrix P. Thus rights in P represent data dependencies in M. If M violates non-inference, its execution will lead to a configuration such that by executing some command c, the adversary can enter a right r into a cell (S_2, x) where x is a private input variable of M. If M satisfies non-inference, then no such command can be executed. The problem of deciding safety for right r in configuration Q can be thus solved by creating a program M from Q and deciding non-inference for M.

Theorem: *Non-inference for independent executions is decidable.*

Proof: We reduce the problem of deciding non-inference for single execution of a program to the safety problem without the create primitive. Safety problem without the create primitive has been shown to be decidable [14].

Given a program M, let V be the set of variables of M and E the set of expressions in M. We create a configuration $Q = (S, O, P)$, such that $S = O = V \cup E \cup In \cup Out$ where In is the placeholder for a subject that has rights over all input variables of M and Out is the placeholder for a subject that has rights over all output variables of M.

M (in the intermediate representation) can have two types of statements: assignment statements or if-then-else statements. Matrix P is created from the statements of M. Assignment statements in M can be of three types: (1) $v_1 = v_2$ (2) $v = e$ (3) $e = f(v_1, v_2, ..v_n)$. where v_i represents a variable, e represents an expression and f represents a function that computes the value of an expression e from the variables occurring in it. P is created from the assignment statements in the following way: for every statement of the form $v_1 = v_2$, right r is entered in the cell corresponding to the row of v_1 and the column of v_2. Similarly, for every statement of the form $v = e$, right r is entered in the cell corresponding to

the row of v and the column of e. For statement of the form $e = f(v_1, v_2, ..v_n)$ (e.g $e = x + y$), right r is entered in the cells corresponding to the row of e and the columns of $v_1, v_2, ..v_n$. All diagonal cells of the matrix have the right r in them. The row corresponding to In has right r in all the columns corresponding to the input variables of M. Similarly, the row corresponding to Out has right r in all the columns corresponding to the output variables of M.

Assignment statements are responsible for explicit information-flow in a program. Conditionals lead to implicit information-flow. For example: if($x = 0$) then $y = 1$ else $y = 2$, creates an information-flow from x to y, since the value of y depends on the value of x. To accommodate for implicit information-flow due to conditionals in M, P is extended in the following manner: for every control variable (e.g x in the above example), right r is entered in the cells corresponding to the column of the control variable and the rows of all variables/expressions occurring on the left-hand-side of the assignment statements that are enclosed within the conditional corresponding to the control-variable. To understand this better, consider the following code snippet:

```
if(x = 0) then
   v1 = v2
else
   v3 = v4
end
```

Here x is the control variable. $v1$ and $v3$ occur on the left-hand-side of the statements enclosed inside the conditional, and hence their value is affected by the value of x. For this example, right r would be entered in the cells $(v1, x)$ and $(v3, x)$ of P.

Configuration $Q = (S, O, P)$ thus created, captures the possible information-flow dependencies in program M. If some command c can be executed on the configuration Q such that right r gets entered in a cell (Out, v), where v is a private input variable, then this translates to an adversary learning the value of v. In that case, M would violate non-inference. If no such command can be executed, then M would satisfy non-inference. Non-inference for M can therefore be decided by deciding safety for the configuration Q.

The *create* primitive is used for creating new subjects or objects. What does the absence of *create* primitive correspond to in the non-inference domain? We informally argue that it corresponds to single execution of a program. For understanding this, it is necessary to understand what primitive operations and commands in the protection system domain, correspond to in the non-inference domain. The command *enter r into (x, y)* in the protection system domain essentially corresponds to $x = f(y)$ in the non-inference domain, and represents information flow (or data dependency) from y to x. Similarly *delete r from (x, y)* corresponds to $x = v$ where v is not a function of y. This results in destroying the data dependency between x and y. The primitive *create subject s* or *create object o* essentially corresponds to creating temporary buffer variables in the non-inference domain, which an adversary can use to store values of output variables corresponding to an execution of a program. When the *create* primitive is taken out of the system, it implies that the adversary cannot create any temporary variables to hold values of variables resulting from one execution. This ensures that information that flows out of a program cannot be fed back into it. In other words, the program cannot be executed again with modified inputs that depend on the outputs of the previous execution. Multiple executions are possible, but they would be independent. This is what we mean by single execution or uni-directional information flow. Note that corresponding to commands that execute on configurations and result in violation of safety, are programs that an adversary would run on the output variables to deduce the value of a private input variable.