ELSEVIER

Science of Computer Programming 37 (2000) 113–138

Science of Computer Programming

www.elsevier.nl/locate/scico

A semantic approach to secure information flow

Rajeev Joshi [a,*], K. Rustan M. Leino [b]

[a] *Department of Computer Sciences, The University of Texas, Austin, TX 78712, USA*
[b] *Compaq Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301, USA*

Abstract

A classic problem in security is that of checking that a program has *secure information flow*. Informally, this problem is described as follows: Given a program with variables partitioned into two disjoint sets of "high-security" and "low-security" variables, check whether observations of the low-security variables reveal any information about the initial values of the high-security variables. Although the problem has been studied for serveral decades, most previous approaches have been syntactic in nature, often using type systems and compiler data flow analysis techniques to analyze program texts. This paper presents a considerably different approach to check secure information flow, based on a semantic characterization. A semantic approach has several desirable features. Firstly, it gives a more precise characterization of security than that provided by most previous approaches. Secondly, it applies to any programming constructs whose semantics are definable; for instance, the introduction of nondeterminism and exceptions poses no additional problems. Thirdly, it can be used for reasoning about indirect leaking of information through variations in program behavior (e.g., whether or not the program terminates). Finally, it can be extended to the case where the high- and low-security variables are defined abstractly, as functions of actual program variables. The paper illustrates the use of the characterization with several examples and discusses how it can be applied in practice.

0. Introduction

A classic problem in security is that of determining whether a given program has *secure information flow* [3,4]. In its simplest form, this problem may be described informally as follows: Given a program whose variables are partitioned into two disjoint sets of "high-security" and "low-security" variables, check whether observations of the low-security variables reveal anything about the initial values of the high-security variables. A related problem is that of detecting *covert flows*, where information is leaked indirectly, through variations in program behavior [12]. For instance, it may be possible to deduce something about the initial values of the high-security variables by

* Corresponding author.
E-mail addresses: joshi@cs.utexas.edu (R. Joshi), rustan@pa.dec.com (K.R.M. Leino).

0167-6423/00/$ - see front matter
PII: S0167-6423(99)00024-6

examining the resource usage of the program (e.g., by counting the number of times it accesses the disk head).

Although this problem has been studied for several decades, most of the previous approaches have been syntactic in nature, often using type systems and compiler data flow analysis techniques to analyze program texts. In this paper, we present a considerably different approach to secure information flow, based on a semantic notion of program equality. A definition based on program semantics has several desirable features. Firstly, it provides a more precise characterization of secure information flow than that provided by most previous approaches. Secondly, it is applicable to any programming construct whose semantics are defined; for instance, nondeterminism and exceptions pose no additional problems. Thirdly, it can be used for reasoning about indirect leaking of information through variations in program behavior (e.g., whether or not the program terminates). Finally, it can be extended to the case where the high- and low-security variables are defined abstractly, as functions of actual program variables.

The outline of the rest of the paper is as follows. We start in Section 1 by informally describing the problem and discussing several small examples. We present our formal characterization of security in Section 2. In Section 3, we describe the notational conventions we use and also provide a short overview of relational and weakest precondition semantics of sequential programs. In Section 4, we formulate our security condition in relational terms and show that it corresponds to the notion used elsewhere in the literature. In Sections 5 and 6 we show how to express our condition in the weakest precondition calculus, which is somewhat more convenient to use in verification. In Section 7, we discuss the main obstacles to applying our approach in practice and propose a stronger condition that is easier to check. This is followed by the discussion in Section 8 of how our techniques may be used even when the high- and low-security variables are defined abstractly. We discuss related work in Section 9 and end with a short summary in Section 10.

1. Informal description of the problem

Consider a program whose variables are partitioned into two disjoint tuples h (denoting "high-security" variables) and k (denoting "low-security" variables). Informally speaking, we say that such a program is *secure* if:

> Observations of the initial and final values of k do not provide any information about the initial value of h.

(Notice that it is only the *initial* value of h that we care about.) We illustrate this informal description of the problem with a few examples. Throughout our discussion, we refer to an "adversary" who is trying to glean some information about the initial value of h. We assume that this adversary has knowledge of the program text and of the initial and final values of k.

The program

$$k := h$$

is not secure, since the initial value of h can be observed as the final value of k.

However, the program

$$h := k$$

is secure, since k, whose value is not changed, is independent of the initial value of h.

Similarly, the program

$$k := 6$$

is secure, because the final value of k is always 6, regardless of the initial value of h.

It is possible for an insecure program to occur as a subprogram of a secure program. For example, in each of the four programs

$$k := h \ ; \ k := 6 \tag{0}$$

$$h := k \ ; \ k := h \tag{1}$$

$$k := h \ ; \ k := k - h \tag{2}$$

$$\textbf{if } \mathit{false} \textbf{ then } k := h \textbf{ end} \tag{3}$$

the insecure program $k := h$ occurs as a subprogram; nevertheless, the four programs are all secure.

There are more subtle ways in which a program can be insecure. For example, with h, k of type boolean, the program

$$\textbf{if } h \textbf{ then } k := \mathit{true} \textbf{ else } k := \mathit{false} \textbf{ end}$$

is insecure, despite the fact that each branch of the conditional is secure. This program has the same effect as $k := h$, and the flow of information from h to k is called *implicit* [4].

In the insecure programs shown so far, the exact value of h is leaked into k. This need not always be the case: a program is considered insecure if it reveals *any* information about the initial value of h. For example, if h and k are of type integer, neither of the two programs shown below,

$$k := h * h$$
$$\textbf{if } 0 \leqslant h \textbf{ then } k := 1 \textbf{ else } k := 0 \textbf{ end},$$

transmits the entire value of h, but both programs are insecure because the final value of k does reveal something about the initial value of h.

A nondeterministic program can be insecure even if the adversary has no knowledge of how the nondeterminism is resolved. For example, the following program is insecure, because the final value of k is always very close to the initial value of h:

$$k := h - 1 \ \square \ k := h + 1$$

(The operator $\square$ denotes demonic choice: execution of $S \square T$ consists of choosing any one of S or T and executing it.) The program

$$skip \square k := h$$

is also insecure, because if the initial and final values of k are observed to be different, then the initial value of h is revealed.

Finally, we give some examples of programs that transmit information about h via their termination behavior. The nicest way to present these examples is by using Dijkstra's **if fi** construct [6]. The operational interpretation of the program

$$\textbf{if } B0 \rightarrow S0 \square B1 \rightarrow S1 \textbf{ fi}$$

is as follows. From states in which neither $B0$ nor $B1$ is true, the program loops forever; from all other states it executes either $S0$ (if $B0$ is true) or $S1$ (if $B1$ is true). If both $B0$ and $B1$ are true, the choice between $S0$ and $S1$ is made arbitrarily. Now, the deterministic program

$$\textbf{if } h = 0 \rightarrow loop \square h \neq 0 \rightarrow skip \textbf{ fi} \tag{4}$$

(where *loop* is the program that never terminates) is insecure, because whether or not the program terminates depends on the initial value of h. Next, consider the following two nondeterministic programs:

$$\textbf{if } h = 0 \rightarrow skip \square true \rightarrow loop \textbf{ fi} \tag{5}$$

$$\textbf{if } h = 0 \rightarrow loop \square true \rightarrow skip \textbf{ fi} \tag{6}$$

Note that program (5) terminates only if the initial value of h is 0. Although there is always a possibility that the program will loop forever, if the program is observed to terminate, the initial value of h is revealed; thus the program is considered insecure. Program (6) is more interesting. If we take the view that nontermination is indistinguishable from slow execution, then the program is secure. However, if we take the view that an adversary is able to detect infinite looping, then it can deduce that the initial value of h is 0, and the program should be considered insecure.

Remark. Readers may be wondering just how much time an adversary would have to spend in order to "detect infinite looping", so the second viewpoint above requires a little explanation. One way to address the issue of nontermination is to require that machine-specific timing information (which an adversary may exploit to detect nontermination) be made explicit in the programming model (e.g., by adding a low-security timer variable, which is updated by each instruction). Another way, which we adopt in this paper, is to strengthen the definition of security by considering powerful adversaries that can detect nontermination. As we will see later (Section 6.0) nontermination is an issue only in the presence of nondeterminism. Even then, our definition is at worst a little conservative, in that it classifies a program such as (6) as insecure.

We hope that these examples, based on our informal description of secure information flow, have helped give the reader an operational understanding of the problem. From now on, we will adopt a more rigorous approach. We start in the next section by formally defining security in terms of program semantics.

2. Formal characterization

Our formal characterization of secure information flow is expressed as an equality between two programs. We use the symbol $\doteq$ to denote program equality based on total correctness and write "S is secure" to mean that program S has secure information flow. (For now, we do not assign a concrete meaning to $\doteq$; that will be done in later sections.)

A key ingredient in our characterization is the program

"assign to h an arbitrary value"

which we denote by HH ("havoc on h"). Program HH may be used to express some useful properties. Firstly, observe that the difference between a program S and the program "HH ; S" is that the latter executes S after setting h to an arbitrary value. Secondly, observe that the program "S ; HH" 'discards' the final value of h resulting from the execution of S. We use these observations below in giving an informal understanding of the following definition of security.

Definition (*Secure information flow*)**.**

$$S \text{ is secure} \equiv (HH \ ; S \ ; HH \ \doteq \ S \ ; HH) \tag{7}$$

Using the two observations above, this characterization may be understood as follows: The occurrence of "$;HH$" on each side indicates that only the final values of k are of interest whereas the occurrence of "$HH;$" on the left side indicates that the program starts with an arbitrary assignment to h. Thus, the two programs are equal provided that the final value of k produced by S does not depend on the initial value of h. In Section 4, we provide a more rigorous justification for this definition, by relating it to a notion of secure information flow that has been used elsewhere in the literature, but for now, we hope that this informal argument gives the reader some operational understanding of our definition. In the rest of this section, we discuss some of the features of our approach.

Firstly, note that we have not stated the definition in terms of a particular style of program semantics (e.g., axiomatic, denotational, or operational). Sequential program equality can be expressed in any of these styles and different choices are suitable for different purposes. For instance, in this paper, we will use a relational semantics to justify our characterization, but we will use weakest precondition semantics to obtain a formulation that is more convenient to use in verification. Secondly, observe that our

definition is given purely in terms of program semantics; thus it can be used to reason about any programming construct whose semantics are defined. For instance, it can be used for programs with nondeterminism and exceptions, as well as complex data structures such as arrays, records, or objects. (In contrast, syntactic definitions such as those based on type systems, typically need to be extended with the introduction of new programming constructs.) Finally, note that our definition leaves open the decision of how h and k are defined. Like some other approaches, one can reason about different kinds of covert flows by introducing appropriate special variables and including them in the low-security variables k. (For instance, one can deal with covert flows involving timing considerations by including in k a program variable that records execution time, c.f. [10].) Unlike other approaches, however, our definition does not require that h and k be actual variables in the program. As we describe in Section 8, our approach may be used even when h, k are defined abstractly, as functions of actual program variables.

3. Notation and semantics primer

In later sections of this paper, we study our security condition using relational program semantics, weakest precondition semantics and Hoare logic. This section explains the notation we use, provides a short introduction to these semantics, and lists properties relating them.

Much of our notation follows Dijkstra and Scholten's book [8]. We use an infix, left-associative dot to denote function application. For $\mathcal{Q}$ denoting either $\forall$ or $\exists$, we write

$$(\mathcal{Q} j : r.j : t.j)$$

to denote the quantification over all j satisfying $r.j$. Identifier j is called the *dummy*, $r.j$ is called the *range*, and $t.j$ is called the *term* of the quantification. When the range is *true* or is understood from context, it is often omitted. We use similar notation to define sets, and write

$$\{ j : r.j : t.j \}$$

to mean the set of all elements of the form $t.j$ for j satisfying $r.j$.

3.0. Relational semantics

In relational semantics, a program is viewed as a relation over the extended state space obtained by adding the special "looping state" ∞ to the space formed by taking the cartesian product of the domains of the program variables. Program equality $\doteq$ in this semantics is relational equality $=$. We use the following notational conventions: Identifiers w, x, y, z denote program states (including ∞). For any program variable v, we write $v.x$ to denote the value of v in state x. (We use the convention that $v.\infty$ is a

special value $\perp$ that lies outside the domain of v.) For any relation S and states x and y, we write $x\langle S\rangle y$ to denote that S relates x to y; this means that there is an execution of program S from (initial) state x to (final) state y. We assume that every program S satisfies

$$(\forall x :: \infty\langle S\rangle x \equiv x = \infty) \tag{8}$$

which states that there is no exit from the looping state (c.f. [7]).

The identity relation is denoted by "*Id*" ; it satisfies $x\langle Id\rangle y \equiv x = y$ for all x, y. The universal relation is denoted by "**true**" ; it satisfies $x\langle \mathbf{true}\rangle y$ for all x, y. The symbols $\subset, ;, \neg, \cup$ denote relational containment, composition, complement, and union, respectively. We will use the facts that *Id* is a left- and a right-identity of composition and that ; distributes over $\cup$ and is therefore monotonic with respect to $\subseteq$.

A program S is called *miraculous* if for some initial state x there is no final state y such that $x\langle S\rangle y$. A *non-miraculous* program therefore corresponds to a relation that is *left-total*, that is, a relation that satisfies

true $\subseteq S$; **true**.

The relational semantics of the program *HH* are given as follows:

$$(\forall x, y :: x\langle HH\rangle y \equiv k.x = k.y)$$

Note that the relation *HH* is both reflexive and transitive:

$$Id \subseteq HH \tag{9}$$

$$HH ; HH \subseteq HH \tag{10}$$

3.1. Weakest precondition semantics and Hoare triples

In the *weakest precondition* semantics [6,8]. a program S is characterized by two predicate transformers $wlp.S$ ("weakest liberal precondition") and $wp.S$ ("weakest precondition") which are informally defined as follows: For any predicate p,

- $wlp.S.p$ holds in exactly those initial states from which every terminating computation of S ends in a state satisfying p, and
- $wp.S.p$ holds in exactly those initial states from which every computation of S terminates in a state satisfying p.

In relational terms, the predicate transformers *wlp.S* and *wp.S* satisfy the following conditions: For any program S and any predicate p

$$(\forall x :: (wlp.S.p).x \equiv (\forall y : x\langle S\rangle y : y = \infty \vee p.y)) \tag{11}$$

$$(\forall x :: (wp.S.p).x \equiv (\forall y : x\langle S\rangle y : y \neq \infty \wedge p.y)) \tag{12}$$

These two transformers are related by the following pairing property: for any S

$$(\forall p :: wp.S.p = wlp.S.p \wedge wp.S.true) \tag{13}$$

For any program with a variable v, we define the unary predicate transformer $[v : \_]$ (read "v-everywhere") as

$$[v : p] = (\forall M :: wlp.\text{“}v := M\text{”}.p)$$

where M ranges over the domain of v (not including $\perp$). This unary predicate transformer has all the properties of universal quantification; in particular, it is universally conjunctive. When v denotes the set of all variables of the program, we will abbreviate $[v : \_]$ by $[\_]$ (read "everywhere").

In weakest precondition semantics, program equality $\doteq$ is equality of wlp and wp :

$$S \doteq T \equiv (\forall p :: [wlp.S.p \equiv wlp.T.p] \wedge [wp.S.p \equiv wp.T.p])$$

which, on account of the pairing property (13), can be simplified to

$$S \doteq T \equiv (\forall p :: [wlp.S.p \equiv wlp.T.p]) \wedge [wp.S.true \equiv wp.T.true]$$

The wlp and wp semantics of the program HH are

$$(\forall p :: [wlp.HH.p \equiv [h : p]])$$
$$[wp.HH.true \equiv true]$$

Informally speaking, the *total-correctness Hoare triple* $\{p\}\ S\ \{q\}$ states that, when started in any state satisfying predicate p, program S is guaranteed to terminate in a state satisfying predicate q. Formally, this triple is defined in terms of wp as follows. For any program S and predicates p and q

$$\{p\}\ S\ \{q\} \equiv [p \Rightarrow wp.S.q]. \tag{14}$$

4. Security in the relational calculus

In this section, we formally justify definition (7) by showing that it is equivalent to the notion used elsewhere in the literature. Since that notion was given in operational terms, we find it convenient to use relational semantics.

In the relational semantics, condition (7) is expressed as follows:

S is secure
$= \quad \{$ Definition (7), program equality $\doteq$ is relational equality $=$ $\}$
$HH\ ;\ S\ ;\ HH = S\ ;\ HH$
$\Rightarrow \quad \{$ (9) and $;$ monotonic, hence $HH\ ;\ S\ ;\ Id \subseteq HH\ ;\ S\ ;\ HH$ $\}$
$HH\ ;\ S \subseteq S\ ;\ HH$
$\Rightarrow \quad \{$ Applying "$;\ HH$" to both sides, using $;$ monotonic $\}$
$HH\ ;\ S\ ;\ HH \subseteq S\ ;\ HH\ ;\ HH$
$\Rightarrow \quad \{$ (10) and $;$ monotonic, hence $S\ ;\ HH\ ;\ HH \subseteq S\ ;\ HH$ $\}$
$HH\ ;\ S\ ;\ HH \subseteq S\ ;\ HH$
$\Rightarrow \quad \{$ (9) and $;$ monotonic, hence $Id\ ;\ S\ ;\ HH \subseteq HH\ ;\ S\ ;\ HH$ $\}$
$HH\ ;\ S\ ;\ HH = S\ ;\ HH$

Since the second expression equals the final one, we have equivalence throughout, and we have

$$S \text{ is secure} \equiv (HH \ ; S \subseteq S \ ; HH) \tag{15}$$

This result is useful because it facilitates the following derivation, which expresses security in terms of the values of program variables.

$$\begin{array}{ll}
 & HH \ ; S \subseteq S \ ; HH \\
= & \quad \{ \text{ definition of relational containment } \} \\
 & (\forall x, y :: x\langle HH \ ; S\rangle y \Rightarrow x\langle S \ ; HH\rangle y \ ) \\
= & \quad \{ \text{ definition of relational composition, twice } \} \\
 & (\forall x, y :: (\exists w :: x\langle HH\rangle w \wedge w\langle S\rangle y \ ) \\
 & \qquad \Rightarrow (\exists z :: x\langle S\rangle z \wedge z\langle HH\rangle y \ )) \\
= & \quad \{ \text{ relational semantics of } HH, \text{ trading twice } \} \\
 & (\forall x, y :: (\exists w : k.x = k.w : w\langle S\rangle y \ ) \\
 & \qquad \Rightarrow (\exists z : x\langle S\rangle z : k.z = k.y \ )) \\
= & \quad \{ \text{ predicate calculus } \} \\
 & (\forall x, y :: (\forall w : k.x = k.w : \\
 & \qquad\qquad w\langle S\rangle y \Rightarrow (\exists z : x\langle S\rangle z : k.z = k.y \ ))) \\
= & \quad \{ \text{ unnesting quantifiers } \} \\
 & (\forall w, x, y : k.x = k.w : \\
 & \qquad w\langle S\rangle y \Rightarrow (\exists z : x\langle S\rangle z : k.z = k.y \ )) \qquad (*) \\
= & \quad \{ \text{ nesting } \} \\
 & (\forall w, x : k.x = k.w : (\forall y :: w\langle S\rangle y \\
 & \qquad\qquad \Rightarrow (\exists z : x\langle S\rangle z : k.z = k.y \ ))) \\
= & \quad \{ \text{ Set calculus } \} \\
 & (\forall w, x : k.x = k.w : \{ y : w\langle S\rangle y : k.y \} \subseteq \{ z : x\langle S\rangle z : k.z \}) \\
= & \quad \{ \text{ Expression is symmetric in } w \text{ and } x \ \} \\
 & (\forall w, x : k.x = k.w : \{ y : w\langle S\rangle y : k.y \} = \{ z : x\langle S\rangle z : k.z \})
\end{array}$$

Thus, we have established that, for any S,

$$\begin{array}{lll}
S \text{ is secure} \equiv (\forall w, x : k.w = k.x : & \{ y : w\langle S\rangle y : k.y \} \\
 & = & \{ z : x\langle S\rangle z : k.z \} \ )
\end{array} \tag{16}$$

This condition says that the set of possible final values of k is independent of the initial value of h. It has appeared in the literature [2] as the definition of secure information flow. (Similar definitions, restricted to the deterministic case, have appeared elsewhere [19,20].) Thus, one may view the derivation above as a proof of the equivalence of (7) with respect to the notion used by others.

Note that we have also shown (see formula $(*)$ in the calculation above)

$$\begin{array}{l}
S \text{ is secure} \equiv (\forall w, x, y : k.w = k.x \wedge w\langle S\rangle y : \\
\qquad\qquad (\exists z : x\langle S\rangle z : k.z = k.y \ ))
\end{array} \tag{17}$$

This formulation of security will be useful in Section 6.

5. Security in the weakest precondition calculus

In this section and the next, we show how our definition of secure information flow may be expressed in the weakest precondition calculus [6]. Our first formulation, presented in this section, involves a quantification over predicates; it is therefore somewhat inconvenient to use. In the next section, we show how this formulation can be written more simply as a condition involving a quantification over the domain of k.

In the weakest precondition semantics, the security condition (7) may be expressed as follows:

$$
\begin{array}{ll}
 & HH \,;\, S \,;\, HH \doteq S \,;\, HH \\
= & \quad \{ \text{ Program equality in terms of } wlp \text{ and } wp \ \} \\
 & \quad (\forall\, p :: [wlp.(HH \,;\, S \,;\, HH).p \equiv wlp.(S \,;\, HH).p] \,) \\
 & \quad \wedge\ [wp.(HH \,;\, S \,;\, HH).true \equiv wp.(S \,;\, HH).true] \\
= & \quad \{ \ wlp \text{ and } wp \text{ of } HH \text{ and } ; \ \} \\
 & \quad (\forall\, p :: [\,[h : wlp.S.[h : p]\,] \equiv wlp.S.[h : p]\,] \,) \qquad (18) \\
 & \quad \wedge\ [\,[h : wp.S.true] \equiv wp.S.true] \qquad (19)
\end{array}
$$

The last formula above contains expressions in which a predicate q satisfies

$$[\,[h : q] \equiv q].$$

Predicates with this property occur often in our calculations, so it is convenient to introduce a special notation for them and identify some of their properties. This is the topic of the following subsection.

5.0. Cylinders

Informally speaking, a predicate q that satisfies $[q \equiv [h : q]\,]$ has the property that its value is independent of the variable h. We refer to such predicates as "h-cylinders", or simply as "cylinders" as h is understood from context. For notational convenience, we define the set Cyl of all h-cylinders:

Definition (*Cylinders*)**.** For any predicate q,

$$q \in Cyl \equiv [q \equiv [h : q]\,] \qquad (20)$$

The following lemma provides several equivalent ways of expressing that a predicate is a cylinder.

Lemma 0. *For any predicate q, the following are all equivalent to $q \in Cyl$.*

(i) $[q \equiv [h : q]\,]$.
(ii) $(\exists\, p :: [q \equiv [h : p]\,])$.
(iii) $\neg q \in Cyl$.

Proof. Follows from predicate calculus. □

We will also use the following result, whose proof follows from predicate calculus: For any predicate q

$$q \in Cyl \Rightarrow (\forall w,x : k.w = k.x : q.w \Rightarrow q.x) \tag{21}$$

5.1. Security in terms of cylinders

We now use the results in the preceding subsection to simplify the formulation of security in the weakest precondition calculus. We begin by rewriting (18) as follows.

$$\begin{array}{ll} & (\forall p :: [[h : wlp.S.[h : p]] \equiv wlp.S.[h : p]]) \\ = & \quad \{ \text{ Definition of } Cyl \ (20) \ \} \\ & (\forall p :: wlp.S.[h : p] \in Cyl) \\ = & \quad \{ \text{ One-point rule } \} \\ & (\forall p,q : [q \equiv [h : p]] : wlp.S.q \in Cyl) \\ = & \quad \{ \text{ Nesting and trading } \} \\ & (\forall q :: (\forall p :: [q \equiv [h : p]] \Rightarrow wlp.S.q \in Cyl)) \\ = & \quad \{ \text{ Predicate calculus } \} \\ & (\forall q :: (\exists p :: [q \equiv [h : p]]) \Rightarrow wlp.S.q \in Cyl) \\ = & \quad \{ \text{ Lemma 0(ii), and trading } \} \\ & (\forall q : q \in Cyl : wlp.S.q \in Cyl) \end{array}$$

Similarly, we rewrite the expression (19) as follows.

$$\begin{array}{ll} & [[h : wp.S.true] \equiv wp.S.true] \\ = & \quad \{ \text{ Definition of } Cyl \ (20) \ \} \\ & wp.S.true \in Cyl \end{array}$$

Putting it all together, we get the following condition for security: For any program S,

$$S \text{ is secure} \equiv (\forall p : p \in Cyl : wlp.S.p \in Cyl) \wedge wp.S.true \in Cyl \tag{22}$$

6. A simpler characterization

Using (22) to check whether a given program S is secure requires evaluation of the following term:

$$(\forall p : p \in Cyl : wlp.S.p \in Cyl) \tag{23}$$

Since this formula involves a quantification over all cylinders, it is inconvenient to use. In this section, we show how this quantification over predicates p can be reduced to a simpler quantification over the domain of k. In particular, we show that it suffices to consider only cylinders that are of the form "$k \neq M$", for M ranging over the domain of k.

First, we restrict the quantification in formula (23) as described above and rewrite the resulting expression in relational terms:

$$
\begin{array}{ll}
 & (\forall M :: wlp.S.(k \neq M) \in Cyl\ ) \\
\Rightarrow & \quad \{\ \text{relational property of cylinders (21)}\ \} \\
 & (\forall M :: (\forall w,x : k.w = k.x : (wlp.S.(k \neq M)).w \\
 & \qquad \Rightarrow (wlp.S.(k \neq M)).x\ )) \\
= & \quad \{\ \text{unnesting and relational definition of } wlp\ (11)\ \} \\
 & (\forall M,w,x : k.w = k.x : (\forall y : w\langle S\rangle y : y = \infty \vee k.y \neq M\ ) \\
 & \qquad \Rightarrow\ (\forall z : x\langle S\rangle z : z = \infty \vee k.z \neq M\ )\ ) \\
= & \quad \{\ \text{unnesting and contrapositive}\ \} \\
 & (\forall M,w,x,z : k.w = k.x \wedge x\langle S\rangle z : \\
 & \quad z \neq \infty \wedge k.z = M \Rightarrow (\exists y : w\langle S\rangle y : y \neq \infty \wedge k.y = M\ )\ ) \\
\Rightarrow & \quad \{\ \text{one-point rule and weakening (by dropping } y \neq \infty)\ \} \\
 & (\forall w,x,z : k.w = k.x \wedge x\langle S\rangle z : \\
 & \quad z \neq \infty \Rightarrow (\exists y : w\langle S\rangle y : k.y = k.z\ ))
\end{array}
$$

Next, we rewrite the condition $wp.S.true \in Cyl$:

$$
\begin{array}{ll}
 & wp.S.true \in Cyl \\
\Rightarrow & \quad \{\ \text{relational property of cylinders (21)}\ \} \\
 & (\forall w,x : k.w = k.x : (wp.S.true).w \Rightarrow (wp.S.true).x\ ) \\
= & \quad \{\ \text{relational property of } wp\ (12)\ \} \\
 & (\forall w,x : k.w = k.x : (\forall y : w\langle S\rangle y : y \neq \infty\ ) \\
 & \qquad \Rightarrow (\forall z : x\langle S\rangle z : z \neq \infty\ )) \\
= & \quad \{\ \text{unnesting and contrapositive}\ \} \\
 & (\forall w,x,z : k.w = k.x \wedge x\langle S\rangle z : \\
 & \qquad z = \infty \Rightarrow (\exists y : w\langle S\rangle y : y = \infty\ )) \\
\Rightarrow & \quad \{\ \text{using } z = \infty \wedge y = \infty \Rightarrow k.y = k.z\ \} \\
 & (\forall w,x,z : k.w = k.x \wedge x\langle S\rangle z : \\
 & \qquad z = \infty \Rightarrow (\exists y : w\langle S\rangle y : k.y = k.z\ ))
\end{array}
$$

Thus, we have

$$
\begin{array}{ll}
 & S \text{ is secure} \\
= & \quad \{\ \text{Condition (22)}\ \} \\
 & (\forall p : p \in Cyl : wlp.S.p \in Cyl\ ) \wedge wp.S.true \in Cyl \\
\Rightarrow & \quad \{\ \text{instantiate with cylinders of the form ``} k \neq M\text{''}\ \} \\
 & (\forall M :: wlp.S.(k \neq M) \in Cyl\ ) \wedge wp.S.true \in Cyl \\
\Rightarrow & \quad \{\ \text{the two calculations above}\ \} \\
 & (\forall w,x,z : k.w = k.x \wedge x\langle S\rangle z : \\
 & \qquad (z \neq \infty \Rightarrow (\exists y : w\langle S\rangle y : k.y = k.z\ )) \\
 & \quad \wedge\ (z = \infty \Rightarrow (\exists y : w\langle S\rangle y : k.y = k.z\ ))\ ) \\
= & \quad \{\ \text{pred calc}\ \} \\
 & (\forall w,x,z : k.w = k.x \wedge x\langle S\rangle z : (\exists y : w\langle S\rangle y : k.y = k.z\ )) \\
= & \quad \{\ \text{Condition (17)}\ \} \\
 & S \text{ is secure}
\end{array}
$$

Since the first and last lines are equivalent, we have equivalence throughout, and thus we have proved the following theorem.

Theorem 1. *For any program S*

$$S \text{ is secure} \equiv (\forall M :: wlp.S.(k \neq M) \in Cyl\ ) \wedge wp.S.true \in Cyl \tag{24}$$

Note that this is simpler than (22) since the quantification ranges over the domain of k.

6.0. Deterministic programs

In the case that S is also known to be deterministic and nonmiraculous, we can further simplify the security condition (24). A deterministic, nonmiraculous program S satisfies the following property [8]:

$$(\forall p :: [wp.S.p \equiv \neg wlp.S.(\neg p)]\ ) \tag{25}$$

Consequently, the term involving wp in (24) is subsumed by the term involving wlp:

$$\begin{array}{ll} & wp.S.true \in Cyl \\ = & \quad \{\ \ (25), \text{ with } p := true \quad \} \\ & \neg wlp.S.false \in Cyl \\ = & \quad \{\ \ \text{Lemma 0(iii)} \quad \} \\ & wlp.S.false \in Cyl \end{array}$$

Since it can also be shown that $wlp.S.false \in Cyl$ follows from $(\forall M :: wlp.S.(k \neq M) \in Cyl\ )$ (cf. [16]), the security condition for deterministic S is given by

$$S \text{ is secure} \equiv (\forall M :: wlp.S.(k \neq M) \in Cyl\ ) \tag{26}$$

Next, we show that condition (26) may also be expressed in terms of wp alone:

$$\begin{array}{ll} & (\forall M :: wlp.S.(k \neq M) \in Cyl\ ) \\ = & \quad \{\ \ \text{Lemma 0(iii)}\ \ p \in Cyl \equiv \neg\, p \in Cyl \quad \} \\ & (\forall M :: \neg\, wlp.S.(k \neq M) \in Cyl\ ) \\ = & \quad \{\ \ \text{Condition (25) above} \quad \} \\ & (\forall M :: wp.S.(k = M) \in Cyl\ ) \end{array}$$

Thus we have another way of expressing the condition for security of deterministic programs, namely,

$$S \text{ is secure} \equiv (\forall M :: wp.S.(k = M) \in Cyl\ ) \tag{27}$$

6.1. Examples

We now illustrate our formulae with some examples.

First, we apply the security condition to program (5), which is insecure because of its termination behavior. Letting N range over the domain of h, we have

$$
\begin{array}{ll}
 & (\textbf{if } h = 0 \longrightarrow \mathit{skip} \;[]\; \mathit{true} \longrightarrow \mathit{loop} \textbf{ fi}) \text{ is secure} \\
= & \quad \{ \text{ Security condition (24) } \} \\
 & (\forall M :: \mathit{wlp}.(\textbf{if } h = 0 \longrightarrow \mathit{skip} \;[]\; \mathit{true} \longrightarrow \mathit{loop} \textbf{ fi}).(k \neq M) \in \mathit{Cyl}\ ) \\
 & \wedge\ \mathit{wp}.(\textbf{if } h = 0 \longrightarrow \mathit{skip} \;[]\; \mathit{true} \longrightarrow \mathit{loop} \textbf{ fi}).\mathit{true} \in \mathit{Cyl} \\
= & \quad \{ \text{ } \mathit{wlp} \text{ and } \mathit{wp}, \text{ using } (\forall p :: [\mathit{wlp}.\mathit{loop}.p \equiv \mathit{true}]) \\
 & \qquad \text{and } [\mathit{wp}.\mathit{loop}.\mathit{true} \equiv \mathit{false}] \ \} \\
 & (\forall M :: ((h = 0 \Rightarrow k \neq M) \wedge (\mathit{true} \Rightarrow \mathit{true})) \in \mathit{Cyl}\ ) \\
 & \wedge\ ((h = 0 \vee \mathit{true}) \wedge (h = 0 \Rightarrow \mathit{true}) \wedge (\mathit{true} \Rightarrow \mathit{false})) \in \mathit{Cyl} \\
= & \quad \{ \text{ pred calc } \} \\
 & (\forall M :: (h = 0 \Rightarrow k \neq M) \in \mathit{Cyl}\ ) \wedge \mathit{false} \in \mathit{Cyl} \\
= & \quad \{ \text{ Lemma 0(i), and } \mathit{false} \in \mathit{Cyl} \ \} \\
 & (\forall M :: [h = 0 \Rightarrow k \neq M \equiv [h : h = 0 \Rightarrow k \neq M]\,]\ ) \\
\Rightarrow & \quad \{ \text{ instantiate with } M := 2 \ \} \\
 & [h = 0 \Rightarrow k \neq 2 \equiv [h : h = 0 \Rightarrow k \neq 2]\,] \\
= & \quad \{ \text{ } [p] \text{ is shorthand for } [h, k : p] \ \} \\
 & [h, k : h = 0 \Rightarrow k \neq 2 \equiv [h : h = 0 \Rightarrow k \neq 2]\,] \\
= & \quad \{ \text{ definition of } [v : \_] \text{ , twice; } \mathit{wlp} \text{ of } := \ \} \\
 & (\forall M, N :: \mathit{wlp}.(k, h := M, N).(h = 0 \Rightarrow k \neq 2 \\
 & \qquad \equiv [h : h = 0 \Rightarrow k \neq 2]\,)\ ) \\
\Rightarrow & \quad \{ \text{ instantiate with } M, N := 2, 2 \ \} \\
 & 2 = 0 \Rightarrow 2 \neq 2 \equiv [h : h = 0 \Rightarrow 2 \neq 2] \\
= & \quad \{ \text{ pred calc, identity of } \equiv \ \} \\
 & [h : h \neq 0] \\
= & \quad \{ \text{ definition of } [h : \_] \ \} \\
 & (\forall N :: \mathit{wlp}.(h := N).(h \neq 0)\ ) \\
\Rightarrow & \quad \{ \text{ instantiate with } N := 0 \ \} \\
 & 0 \neq 0 \\
= & \quad \{ \text{ pred calc } \} \\
 & \mathit{false}
\end{array}
$$

Next, using program (4), we illustrate how one can reason about secure termination behavior of deterministic programs using *wlp*.

$$
\begin{array}{ll}
 & (\textbf{if } h = 0 \longrightarrow \mathit{loop} \;[]\; h \neq 0 \longrightarrow \mathit{skip} \textbf{ fi}) \text{ is secure} \\
= & \quad \{ \text{ Security condition for deterministic programs (26) } \} \\
 & (\forall M :: \mathit{wlp}.(\textbf{if } h = 0 \longrightarrow \mathit{loop} \;[]\; h \neq 0 \longrightarrow \mathit{skip} \textbf{ fi}).(k \neq M) \in \mathit{Cyl}\ ) \\
= & \quad \{ \text{ } \mathit{wlp} \text{ of } \textbf{if} \ \} \\
 & (\forall M :: ((h = 0 \Rightarrow \mathit{true}) \wedge (h \neq 0 \Rightarrow k \neq M)) \in \mathit{Cyl}\ ) \\
= & \quad \{ \text{ Definition of } \mathit{Cyl} : \text{ note } h \neq 0 \Rightarrow k \neq M \text{ depends on } h \ \} \\
 & \mathit{false}
\end{array}
$$

7. Practical considerations

Despite the simple mathematical nature of our characterization of secure information flow, there is one serious obstacle to using it in practice: the property $p \in Cyl$ is neither monotonic nor antimonotonic in p with respect to the ordering $\Rightarrow$. Consequently, security is not preserved by refinement. (As an illustration of this, note that the secure program "assign to k an arbitrary value" is refined by the insecure program "$k := h$".)

This leads to two difficulties. The first is that, since sequential programs are typically implemented by refining their nondeterminism, there is the possibility that a (nondeterministic) secure program received from an adversary is rendered insecure when it is implemented deterministically. There are at least two ways to address this first difficulty. The first is to recognize that refinements are a concern only if the adversary is aware of how they are made. If we take the position that the adversary has absolutely no knowledge of how a program is refined in the process of its implementation (or how nondeterministic choices are resolved during execution), we can assert that its observations reveal no information about the initial value of h and so the implementation is secure. The second way to address this first difficulty is by noting that the problem does not arise for deterministic programs, since the latter are maximal in the refinement ordering. Thus the difficulty is avoided by requiring that programs received from an adversary be deterministic. This latter approach is similar to the one advocated by Roscoe [15], who gives several characterizations (corresponding to different observational models) for the secure information flow property for CSP processes. He makes a persuasive argument for requiring determinism by showing that these characterizations are all equivalent for deterministic processes.

The second difficulty is that in order to check whether $wlp.S.p$ is a cylinder, one has to compute the *exact* expression for $wlp.S$ for a given program S. For straight-line programs, this is straightforward, but for programs with iteration or recursion, determining the exact expression for wlp requires complex fixpoint computations, which makes this checking unattractive in practice.

One way to address this second difficulty would be to require that each program somehow be annotated with a proof of its security. Verifying security would then reduce to the task of checking whether the annotations were correct, a task that can be substantially simpler than proof construction. (This idea has recently been popularized by Necula and Lee, who call it *proof-carrying code* [14].)

In the rest of this section, we describe a restriction under which security can be checked without exact wlp computation. As we will see, our restriction allows us to check security by checking Hoare triples; thus it can be used to verify programs annotated with Hoare-style assertions. (Note that, since Hoare triples are closed under refinement, it follows that not only is any program satisfying our restriction secure, but so are all its refinements.)

We start by introducing two definitions. A program S is said to be *functional* with respect to variable k if the initial value of k determines its final value:

$$(\forall w,x,y,z : k.w = k.x \wedge w\langle S\rangle y \wedge x\langle S\rangle z : k.y = k.z)$$

A program S is a *function* with respect to variable k if it is also total in its domain:

$$S \text{ is a function w.r.t. } k \equiv S \text{ is functional w.r.t. } k \wedge S \text{ is left-total}$$

The relevance of these notions to secure information flow is given by the following theorem.

Theorem 2. *For any program S,*

$$S \textit{ is function w.r.t. } k \Rightarrow S \textit{ is secure}$$

Proof. We begin by rewriting the definition of "functional w.r.t. k" in the relational calculus.

$$\begin{array}{ll}
 & S \text{ is functional w.r.t. } k \\
= & \quad \{ \text{ definition } \} \\
 & (\forall w,x,y,z : k.w = k.x \wedge w\langle S\rangle y \wedge x\langle S\rangle z : k.y = k.z) \\
= & \quad \{ \text{ trading, thrice } \} \\
 & (\forall w,x,y,z : w\langle S\rangle y \wedge k.y \neq k.z : k.w \neq k.x \vee \neg\, x\langle S\rangle z) \\
= & \quad \{ \text{ trading, de Morgan } \} \\
 & (\forall w,z :: (\forall x,y :: w\langle S\rangle y \wedge k.y \neq k.z \\
 & \qquad \Rightarrow \neg(k.w = k.x \wedge x\langle S\rangle z))) \\
= & \quad \{ \text{ pred calc } \} \\
 & (\forall w,z :: (\exists y :: w\langle S\rangle y \wedge k.y \neq k.z) \\
 & \quad \Rightarrow (\forall x :: \neg(k.w = k.x \wedge x\langle S\rangle z))) \\
= & \quad \{ \text{ de Morgan } \} \\
 & (\forall w,z :: (\exists y :: w\langle S\rangle y \wedge k.y \neq k.z) \\
 & \quad \Rightarrow \neg(\exists x :: k.w = k.x \wedge x\langle S\rangle z)) \\
= & \quad \{ \text{ relational composition, definition of } HH \} \\
 & (\forall w,z :: w\langle S ; \neg HH\rangle z \Rightarrow \neg\, w\langle HH ; S\rangle z) \\
= & \quad \{ \text{ relational calculus } \} \\
 & S ; \neg HH \subseteq \neg(HH ; S)
\end{array}$$

Remark. In the relational calculus (see, e.g., [7]), the fact that a relation f is "(left-) functional" is expressed as

$$f ; \neg Id \subseteq \neg f.$$

Note that this may be written as

$$f ; \neg Id \subseteq \neg(Id ; f).$$

One can view this as a special case of the result above, obtained by taking for k the entire set of underlying variables and letting h be a constant (i.e., a single variable whose domain has exactly one element), in which case HH is just Id.

Thus, for any S that is a function with respect to k, we have

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \quad \{ \ S \text{ is left-total} \ \} \\
 & \mathbf{true} \ \subseteq \ S \ ; \ \mathbf{true} \\
= & \quad \{ \ \text{relational calculus: distributing } ; \text{ over } \cup \ \} \\
 & \mathbf{true} \ \subseteq \ S \ ; \ \neg HH \ \cup \ S \ ; \ HH \\
\Rightarrow & \quad \{ \ S \text{ is functional w.r.t } k, \text{ calculation above} \ \} \\
 & \mathbf{true} \ \subseteq \ \neg(HH \ ; \ S) \ \cup \ S \ ; \ HH \\
= & \quad \{ \ \text{relational calculus: shunting} \ \} \\
 & HH \ ; \ S \ \subseteq \ S \ ; \ HH \\
= & \quad \{ \ (15) \ \} \\
 & S \text{ is secure } \ \square
\end{array}
$$

Remark. It should be noted that "S is a function w.r.t. k" does not mean that S is also deterministic, nor vice-versa. The programs HH and $k := h$ serve respectively as counterexamples.

The following corollary uses the theorem above to derive a formulation for security in terms of Hoare triples for total correctness.

Corollary 3. *For any non-miraculous program S and any function f,*

$$(\forall M :: \{ \ k = M \ \} \quad S \quad \{ \ k = f.M \ \}) \Rightarrow \ S \text{ is secure.}$$

Proof. In order to avoid case analyses, we extend f to $\perp$ by using the convention that $f.\perp = \perp$. (Recall that $\perp$ is the special value of k in the looping outcome ∞.) We start by rewriting the hypothesis in relational terms:

$$
\begin{array}{ll}
 & (\forall M :: \{ \ k = M \ \} \quad S \quad \{ \ k = f.M \ \}) \\
= & \quad \{ \ \text{Connection between total correctness Hoare triple and } wp \ \} \\
 & (\forall M :: [k = M \ \Rightarrow \ wp.S.(k = f.M)]) \\
= & \quad \{ \ \text{“[\_]” quantifies over all nonlooping states} \ \} \\
 & (\forall M :: (\forall w : w \neq \infty : (k = M).w \ \Rightarrow \ wp.S.(k = f.M).w)) \\
= & \quad \{ \ \text{Relational property of } wp \ (12) \text{ and pred calc} \ \} \\
 & (\forall M :: (\forall w : w \neq \infty : k.w = M \\
 & \qquad \Rightarrow (\forall y : w\langle S\rangle y : y \neq \infty \wedge k.y = f.M))) \\
= & \quad \{ \ \text{unnesting, trading} \ \} \\
 & (\forall M, w, y : w \neq \infty \wedge k.w = M \wedge w\langle S\rangle y : \\
 & \qquad\qquad y \neq \infty \wedge k.y = f.M) \\
= & \quad \{ \ \text{one-point rule and Leibniz} \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall w, y : w \neq \infty \wedge w\langle S\rangle y : y \neq \infty \wedge k.y = f.(k.w)) \\
\Rightarrow & \quad \{ \text{ weakening (by dropping } y \neq \infty) \} \\
 & (\forall w, y : w \neq \infty \wedge w\langle S\rangle y : k.y = f.(k.w)) \\
= & \quad \{ \text{ using } w = \infty \wedge w\langle S\rangle y \Rightarrow y = \infty, \text{ convention } f.\bot = \bot \} \\
 & (\forall w, y : w\langle S\rangle y : k.y = f.(k.w))
\end{array}
$$

Now, assuming the hypothesis, we show that S is a function w.r.t. k, and hence – by the theorem above – that S is secure. Since S is non-miraculous, the relation S is left-total, thus it only remains to show that S is functional w.r.t. k. We observe

$$
\begin{array}{ll}
 & S \text{ is functional w.r.t. } k \\
= & \quad \{ \text{ Definition } \} \\
 & (\forall w, x, y, z :: k.w = k.x \wedge w\langle S\rangle y \wedge x\langle S\rangle z \Rightarrow k.y = k.z) \\
\Leftarrow & \quad \{ \text{ weakening the antecedent by using the calculation above } \} \\
 & (\forall w, x, y, z :: k.w = k.x \wedge k.y = f.(k.w) \wedge k.z = f.(k.x) \\
 & \qquad \Rightarrow k.y = k.z) \\
\Leftarrow & \quad \{ \text{ Rule of Leibniz, since } f \text{ is a function } \} \\
 & \mathit{true} \quad \square
\end{array}
$$

We end this section by applying this corollary to a couple of examples.

7.0. Examples

As a first example, let S be the following program, where k, h are nonnegative integers.

$$h := |h| \ ; \ \textbf{while } 0 < h \ \textbf{do } h := h - 1 \ ; \ k := k + 1 \ \textbf{end} \ ; \ k := h$$

We sketch the proof that S is secure by applying the Corollary above with the constant function 0. The annotated program showing $\{ k = M \}\ S\ \{ k = 0 \}$ is shown below:

$$
\begin{array}{l}
\{ k = M \} \\
h := |h| \\
\{ 0 \leqslant h \} \\
; \ \textbf{while } \ 0 < h \ \textbf{do} \\
\qquad \{ \mathit{Invariant}: \ 0 \leqslant h \} \\
\qquad \{ \mathit{Variant\ function}: \quad h \} \\
\quad ; \ h := h - 1 \\
\quad ; \ k := k + 1 \\
\quad \textbf{end} \\
\{ h = 0 \} \\
k := h \\
\{ k = 0 \}
\end{array}
$$

As a second example, program (2) can be shown to be secure by proving the Hoare triple

$$\{\ k = M\ \} \quad k := h\ ;\ k := k - h \quad \{\ k = f.M\ \}$$

where f is the constant function 0. Note that Corollary 3 is not helpful for the two examples in Section 6.1, since it can only be used to prove programs secure, but not to declare them insecure. Finally, note that the Corollary cannot be applied to check a program such as

"assign to k an arbitrary value"

since such a program is not functional with respect to k.

8. Abstract variables

As mentioned in Section 2, one of the features of our semantic approach is that h and k need not be actual variables of a program. Indeed, as we describe in this section, our definition of security may be used even when h, k are defined abstractly, as functions of the actual program variables.

To illustrate the idea, consider a program S operating on a nonempty list ℓ of votes recorded in an election. Suppose S is supposed to read list ℓ and report the name of the winner, but not reveal any information about the winning margin of the victor. (For the sake of simplicity, we assume that ties are resolved in a fixed manner, so that there is always a unique winner, and that the winning margin is always a positive integer.) To apply our condition to check security of S, we let k denote the function on ℓ that returns the name of the winner and let h denote the function on ℓ that returns the winning margin. (Note that both h and k are functions of ℓ, but not actual variables of the program.) We can then use these definitions of k and h to check whether the program leaks any information about the winning margin.

However, there is one precaution that needs to be observed when using our condition for security with h and k defined as functions of the underlying program variables. In justifying that our definition (7) using HH captures the intent that "observations of k do not leak any information about the initial value of h", we have (implicitly) assumed that h may be assigned a value independently of k. This allowed us to treat HH as an arbitrary assignment to h that leaves k unchanged. When h and k are disjoint program variables, this condition is met trivially and need not be stated explicitly. When they are functions over the same program variables, however, we need an additional condition to ensure that h may still be assigned values independently of k. With V ranging over the domain of the program variables and M, N ranging over the ranges of the functions k, h respectively, this condition is as follows:

$$(\forall M,N :: (\exists V :: k.V = M \wedge h.V = N)) \tag{28}$$

For instance, in the example above, M ranges over the set of candidate names and N ranges over the positive integers. It is easy to show that the independence condition (28) follows from the fact that given any name M and a winning margin N, there is at least one choice V for ℓ for which $k.V = M$ and $h.V = N$ (e.g., let V be the list consisting of the name M repeated N times).

In the following subsection, we illustrate the use of abstract variables with an example.

8.0. Example

We are given an array a which stores information about the employees of a company. Each element of a is a record with the following fields:

$$name, salary, rank,$$

where *rank* is of type $\{Manager, Subordinate\}$. Let S be a program which is supposed to compute the total salary *tot* of all subordinates. We show how we can check whether S leaks any information about the managers.

Clearly type-based schemes are difficult, if not impossible, to use, since records are distinguished by the value of the rank field, which cannot be checked syntactically. However, we can apply our semantic condition by defining h and k as functions of a and *tot* in the following way.

For any record r, introduce the shorthands *Man* and *Sub* by

$$Man.r \equiv r.rank = Manager$$
$$Sub.r \equiv r.rank = Subordinate$$

Next, define two functions *pm* and *ps* from arrays to arrays as follows. For any array A of records,

$$pm.A = \text{subarray of } A \text{ consisting of records } r \text{ satisfying } Man.r$$
$$ps.A = \text{subarray of } A \text{ consisting of records } r \text{ satisfying } Sub.r$$

Now h and k are defined as functions of the concrete variables a and *tot* as follows:

$$h = pm.a \quad \text{and} \quad k = \langle ps.a, tot \rangle.$$

Note that these definitions satisfy the independence condition (28). (For any given values N, M for h, k, choose the concrete variable a to be any interleaving of N and the first component of M and take *tot* to be the second component of M.)

To illustrate how these definitions may be used, we consider the following program S:

```
  tot := 0
; var j in
    j := 0
  ; while j ≠ size.a do
      if Sub.(a[j]) then tot := tot + a[j].salary end
    ; j := j + 1
    end
  end
```

We show that S is secure. Define auxiliary functions *ssum* and *psum* as follows: for any array A and any integer j

$$ssum.A = (\Sigma i : 0 \leqslant i < size.A : A[i].salary)$$

$$psum.j.A = (\Sigma i : 0 \leqslant i < j \wedge Sub.(A[i]) : A[i].salary)$$

We note the following property about *ssum*, *ps*, and *psum*: for any A

$$ssum.(ps.A) = psum.(size.A).A. \tag{29}$$

Finally, we define function f as follows: for any array A and integer T,

$$f.\langle A, T\rangle = \langle A, ssum.A\rangle.$$

To show that S is secure, it is enough – on account of Theorem 2 – to show that it satisfies the following Hoare triple, for any array A and any integer T :

$$\{\ k = \langle A, T\rangle\ \} \quad S \quad \{\ k = f.\langle A, T\rangle\ \}$$

We sketch the outline of this proof below by annotating S with assertions as follows:

```
    { ps.a = A ∧ tot = T }
  tot := 0
    { ps.a = A ∧ tot = 0 }
;   var j in
  j := 0
      { ps.a = A ∧ tot = j = 0 }
    ; while j ≠ size.a do
        { Invariant : ps.a = A ∧ tot = psum.j.a ∧ 0 ⩽ j ⩽ size.a }
        { Variant function : size.a − j }
      if Sub.(a[j]) then tot := tot + a[j].salary end
    ; j := j + 1
    end
  end
  { ps.a = A ∧ tot = psum.(size.a).a }
```

9. Related work

The problem of secure information flow has been studied for several decades. A commonly used mathematical model for secure information flow is Denning's lattice model [4], which is based on the Bell and La Padula security model [3]. Most approaches to static certification of secure information flow (an area pioneered by Denning and Denning [4,5]) seem to fall into one of two general categories: type systems and data flow analysis techniques. In this section, we discuss these general approaches and compare them to our work. A historical perspective of secure information flow appears in a book by Gasser [9]. Some newer work in this area includes the SLam calculus [11], Myers and Liskov's decentralized model whose certification process leaves some checks until run time [13], and the semantic approach by Sabelfeld and Sands [17].

9.0. Approaches based on type systems

The static certification mechanism proposed by Denning and Denning [5] is essentially a type checker for secure information flow. Each variable x occurring in a program is declared with a particular *security class*, denoted by $class.x$. These security classes are assumed to form a lattice, ordered by $\leqslant$, with meet (greatest lower bound) denoted by $\downarrow$ and join (least upper bound) denoted by $\uparrow$. The type checker computes the class of an expression as the join of the classes of its subexpressions. For example, for an expression involving addition, we have

$$class.(E + F) \;=\; class.E \uparrow class.F$$

A security class is also assigned to each statement, and is computed as the meet of the security classes of the variables assigned to by that statement. For instance,

$$class.(x := E) \;=\; class.x$$
$$class.(\textbf{if } E \textbf{ then } S \textbf{ else } T \textbf{ end}) \;=\; class.S \downarrow class.T$$

The type checker certifies a program S as being secure provided the following two conditions hold:

0. For every assignment statement $x := E$ in S, $class.E \leqslant class.x$
1. For every conditional statement **if** E **then** T **else** U **end** in S, $class.E \leqslant class.T$ and $class.E \leqslant class.U$.

Other programming constructs, such as loops, give rise to similar requirements.

Denning and Denning gave an informal argument for the soundness of their certification mechanism (i.e., a proof that the mechanism certifies only secure programs). Recently, Volpano et al. have given a more rigorous proof [19,20].

The advantage of using a type system as the basis of a certification mechanism is that it is simple to implement. However, most certification mechanisms based on types reject any program that contains an insecure subprogram. As we saw in examples

(0)–(3) of Section 1, a secure program may contain an insecure subprogram. In contrast, with a semantic approach like ours, it is possible to identify such programs as being secure. Another problem with such approaches is that they are difficult to use for reasoning about programs that leak information via termination behaviour. (Volpano and Smith [18] have attempted to extend their type-based approach to handle termination behaviour. However, their type system rejects any program that mentions h in a loop guard. Such an approach seems terribly restrictive.)

9.1. Approaches based on data flow analyses

The key idea behind approaches based on data flow analyses is to transform a given program S into a program S' that provides a simpler representation of the possible data flows in program S. This is done as follows. (We assume, as in the previous section, that we are given a lattice of security classes.) For every variable x in program S, program S' contains a variable x', representing the highest security class of the values used in computing the current value for x. To deal with implicit flows, S' also contains a special variable $local'$, representing the lowest security class of the values used to compute the guards that led to execution of the current instruction. For example, for every assignment statement in S of the form $x := y + z$, S' contains a corresponding statement

$$x' := y' \uparrow z' \uparrow local'$$

For a conditional statement in S such as

$$\textbf{if } x = y \longrightarrow S0 \;[]\; z < 0 \longrightarrow S1 \textbf{ fi}$$

S' contains a corresponding statement

$$\begin{array}{l}\textbf{var } old := local' \textbf{ in}\\ \quad local' := local' \uparrow x' \uparrow y' \uparrow z'\\ ; \textbf{ if } true \longrightarrow S0' \;[]\; true \longrightarrow S1' \textbf{ fi}\\ ; \; local' := old\\ \textbf{end}\end{array}$$

where $S0'$ and $S1'$ are the statements in S' that correspond to $S0$ and $S1$. If a program S has the variables k and h belonging to the security classes low and high (denoted $\bot$ and $\top$, respectively, where $\bot \leqslant \top$), then "S is secure" can be expressed as the following Hoare triple on S':

$$\{\; k' \leqslant \bot \wedge h' \leqslant \top \wedge local' \leqslant \bot \;\} \quad S' \quad \{\; k' \leqslant \bot \;\} \tag{30}$$

The first data flow analysis approach of this kind was given by Andrews and Reitman [1], whose treatment also dealt with communicating sequential processes. Banâtre et al. [2] used a variation of the method described above that attempts to keep track of the set of initial variables used to produce a value rather than only the security class of the

value. They also developed an efficient algorithm for their approach, similar to data flow analysis algorithms used in compilers, and attempted a proof of soundness. (Unlike our description above, Andrews and Reitman used the deterministic **if then else** construct rather than Dijkstra's **if fi** construct. Banâtre et al. used the **if fi** construct, but, as Volpano et al. point out, their soundness theorem is actually false for nondeterministic programs [20].)

The data flow analysis approach can provide more precision than the simple type system approach mentioned in the previous section. For example, the data flow analysis approach would certify programs (0) and (1). However, the approach still rejects some secure programs that our approach will certify. This comes about because of two reasons. The first reason is that the semantics of operators like $+$ and $-$ are lost in the rewriting of S into S'. Thus a program like (2), which is secure on account of that $h - h = 0$, is rejected by the data flow analysis approach. The second reason is that guards are replaced by *true* in the rewriting of S into S'. Thus, a program like (3), whose security depends on when control can reach a certain statement, is rejected.

One way to improve on this approach is to augment it with a logic, as suggested by Andrews and Reitman [1]. Instead of rewriting program S into S', one superimposes new variables (k', h', $local'$) and their updates onto program S, and then reasons about S using the Hoare triple (30) but with S instead of S'. A consequence of this approach is that one can rule out some impossible control paths, such as the one in program (3).

10. Summary

We have presented a simple and new mathematical characterization of what it means for a program to have secure information flow. The characterization is general enough to accommodate reasoning about a variety of covert flows, including nontermination. Unlike previous methods, which were based on type systems and compiler data flow analysis techniques, our characterization is in terms of program semantics, thus it is more precise than these syntactic approaches.

The precision of a semantic approach comes at a price, which includes finding various fix-points that characterize the semantics of iterative or recursive program constructs. We have shown that this cost can be reduced for a restricted set of programs, for which the security condition follows from a Hoare triple.

We have also shown how our definition may be applied when high- and low-security variables are defined abstractly.

Acknowledgements

We are grateful to Ernie Cohen and Martín Abadi for sharing their insights and comments on our work. Ernie suggested the use of *HH* in our characterization of

security. Martín provided valuable feedback on earlier drafts of this paper. Both helped in formulating Corollary 3.

Mark Lillibridge and Raymie Stata helped prove the formulation of security for deterministic programs (condition (27)). Greg Nelson suggested the use of $k \neq M$ in extending this formulation to nondeterministic programs. Rutger M. Dijkstra suggested the introduction of the "v-everywhere" notation as a means of avoiding the confusion between program variables and dummy variables.

We also thank Jayadev Misra, the members of the Austin Tuesday Afternoon Club, the participants at the September 1997 session of the IFIP WG 2.3 meeting in Alsace, France, and the anonymous referees for MPC 98 and the special issue of Science of Computer Programming, for providing many comments and suggestions which have improved the exposition.

References

[1] G.R. Andrews, R.P. Reitman, An axiomatic approach to information flow in programs, ACM Trans. Programm. Languages Systems 2 (1) (1980) 56–76.

[2] J.-P. Banâtre, C. Bryce, D. Le Métayer, Compile-time detection of information flow in sequential programs, in: Proc. European Symp. on Research in Computer Security, Lecture Notes in Computer Science, vol. 875, Springer, Berlin, 1994, pp. 55–73.

[3] D.E. Bell, L.J. La Padula, Secure computer systems: mathematical foundations and model, Tech. Rep. M74-244, MITRE Corporation, Bedford, Massachusetts, 1973.

[4] D.E. Denning, A lattice model of secure information flow, Commun. ACM 19 (5) (1976) 236–243.

[5] D.E. Denning, P.J. Denning, Certification of programs for secure information flow, Commun. ACM 20 (7) (1977) 504–513.

[6] E.W. Dijkstra. A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[7] R.M. Dijkstra, Relational calculus and relational program semantics, Tech. Rep. CS-R 9408, University of Groningen, Netherlands, 1994.

[8] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Texts and Monographs in Computer Science, Springer, Berlin, 1990.

[9] M. Gasser, Building a Secure Computer System, Van Nostrand Reinhold Company, New York, 1988.

[10] E.C.R. Hehner, Predicative programming Part I, Commun. ACM 27 (2) (1984) 134–143.

[11] N. Heintze, J.G. Riecke, The SLam calculus: programming with secrecy and integrity, in: Proc. 25th ACM Conf. Principles of Programming Languages, ACM Press, New York, 1998, pp. 1–12.

[12] B.W. Lampson, A note on the confinement problem, Commun. ACM 16 (10) (1973) 613–615.

[13] A.C. Myers, B. Liskov, A decentralized model for information flow control, in Proc 16th ACM Symp. on Operating System Principles, Oper. System Rev. 31 (5) (1997) 27–37.

[14] G.C. Necula, P. Lee, Safe Kernel Extensions Without Run-Time Checking, in: Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation (OSDI), October 1996, pp. 229–243.

[15] A.W. Roscoe, CSP and determinism in security modelling, in: Security and Privacy, IEEE, New York, 1995.

[16] K. Rustan, M. Leino, R. Joshi, A semantic approach to secure information flow, in: Proc. 4th Int. Conf. on Mathematics of Program Construction (MPC98), June 1998.

[17] A. Sabelfeld, D. Sands, A per model of secure information flow in sequential programs, in: Proc. European Symp. on Programming (ESOP'99), 1999, to appear.

[18] D. Volpano, G. Smith, Eliminating covert flows with minimum typings, in: Proc. 10th IEEE Computer Security Foundations Workshop, June 1997, pp. 156–168.

[19] D. Volpano, G. Smith, A type-based approach to program security, in: Theory and Practice of Software Development: Proc./TAPSOFT'97, 7th Int. Joint Conf. CAAP/FASE, Lecture Notes in Computer Science, vol. 1214, Springer, Berlin, April 1977, pp. 607–621.
[20] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, J. Comp. Security 4 (3) (1996) 1–21.