A Semantic Approach to Secure Information Flow

K. Rustan M. Leino[1] and Rajeev Joshi[2]

[1] DEC SRC, Palo Alto, CA 94301, USA
rustan@pa.dec.com
[2] University of Texas, Austin, TX 78712, USA
joshi@cs.utexas.edu

Abstract. A classic problem in security is to determine whether 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 several 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 checking 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).

0 Introduction

A classic problem in security is that of determining whether a given program has *secure information flow* [BLP73,Den76]. 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 [Lam73]. For instance, it may be possible to deduce something about the initial values of the high-security variables by 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 applied to reasoning about a variety of covert flows, including termination behavior and timing dependent flows.

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 relate our definition to the notion used elsewhere in the literature. In sections 4 and 5, we show how to rewrite our definition in the weakest precondition calculus so that it is amenable for use with tools for mechanical verification. We discuss related work in section 6 and end with a short summary in section 7.

1 Informal description of the problem

Throughout the rest of the paper, we assume that in each program considered, the variables are partitioned into two disjoint tuples h (denoting "high-security" variables) and k (denoting "low-security" variables). Informally, we say that 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

$$k := h \;;\; k := 6 \qquad (0)$$
$$h := k \;;\; k := h \qquad (1)$$
$$k := h \;;\; k := k - h \qquad (2)$$
$$\mathbf{if}\ \mathit{false}\ \mathbf{then}\ k := h\ \mathbf{end} \qquad (3)$$

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* [Den76].

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,

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

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:

$$k := h - \mathit{1} \;\square\; k := h + \mathit{1}$$

(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

$$\mathit{skip} \;\square\; k := h$$

is also considered 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 [Dij76]. The operational interpretation of the program

$$\textbf{if } \mathit{B0} \longrightarrow \mathit{S0} \;\square\; \mathit{B1} \longrightarrow \mathit{S1} \textbf{ fi}$$

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

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

(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:

$$\textbf{if } h = \mathit{0} \longrightarrow \mathit{skip} \;\square\; \mathit{true} \longrightarrow \mathit{loop} \textbf{ fi} \tag{5}$$
$$\textbf{if } h = \mathit{0} \longrightarrow \mathit{loop} \;\square\; \mathit{true} \longrightarrow \mathit{skip} \textbf{ fi} \tag{6}$$

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 the 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 who can detect nontermination. As we will see later, these two approaches yield the same definition for deterministic programs; it is only in the presence of nondeterminism that differences arise. Even then, our definition is at worst a little conservative, in that it classifies a program such as (6) as insecure. *(End of Remark.)*

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.

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)

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

Using the two observations above, this characterization may be understood as follows. First, note that the final occurrence of HH on each side means that only the final value of k (but not of h) is retained. Next, observe that the prefix "HH ;" in the first program means that 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 3, 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 condition that is more amenable for use in mechanical 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, nondeterminism and exceptions pose no additional problems in this approach, nor do data structures such as arrays, records, or objects. (In contrast, definitions based on type systems often need to be extended with the introduction of new constructs.) Finally, note that our definition leaves open the decision of which variables are deemed to be low-security (*i.e.*, observable by an adversary). Different choices may be used to reason about different kinds of covert flows, by introducing appropriate special variables (such as those used by Hehner [Heh84]) and including them in the low-security variables k. For example, one can reason about covert flows involving timing considerations by including in k a program variable that records execution time.

3 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 a relational semantics for total correctness. Thus, for the purposes of this section, a program is a relation over the space formed by extending the state space defined by h and k with the special "looping state" ∞ [RMD92]. We use the following notational conventions: Identifiers w, x, y, z denote program states; we write $x.k$ and $x.h$ to denote the values of k and h in state x. 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 the initial state x to final state y. We assume that every program S satisfies $x\langle S\rangle\infty \equiv x = \infty$ for all x and that $\infty.k$ differs from $x.k$ for all x that differ from ∞. The identity relation is denoted by "Id"; it satisfies $x\langle Id\rangle y \equiv x = y$ for all x and y. The symbol $\subseteq$ denotes relational containment and the operator ; denotes relational composition. We will use the facts that Id is a left- and a right-identity of composition and that ; is monotonic with respect to $\subseteq$ in both arguments.

We use the following format for writing quantified expressions [DS90]: 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 sometimes omitted. We use a similar convention 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 $r.j$.

The relational semantics of program HH are given as follows.

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

Note that the relation HH is both reflexive and transitive:

$$Id \ \subseteq \ HH \tag{8}$$

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

Using these properties, condition (7) may be rewritten in relational terms as follows.

$$\begin{array}{ll}
 & S \text{ is secure} \\
= & \quad \{ \text{ Definition (7), program equality } \doteq \text{ is relational equality } = \ \} \\
 & HH \ ; S \ ; HH \quad = \quad S \ ; HH \\
\Rightarrow & \quad \{ \text{ (8) and } ; \text{ monotonic, hence } HH \ ; S \ ; Id \ \subseteq \ HH \ ; S \ ; HH \ \} \\
 & HH \ ; S \ \subseteq \ S \ ; HH \\
\Rightarrow & \quad \{ \text{ Applying ``} ; HH \text{'' to both sides, using } ; \text{ monotonic } \} \\
 & HH \ ; S \ ; HH \ \subseteq \ S \ ; HH \ ; HH \\
\Rightarrow & \quad \{ \text{ (9) and } ; \text{ monotonic, hence } S \ ; HH \ ; HH \ \subseteq \ S \ ; HH \ \} \\
 & HH \ ; S \ ; HH \ \subseteq \ S \ ; HH \\
\Rightarrow & \quad \{ \text{ (8) and } ; \text{ monotonic, hence } Id \ ; S \ ; HH \ \subseteq \ HH \ ; S \ ; HH \ \} \\
 & HH \ ; S \ ; HH \quad = \quad S \ ; HH
\end{array}$$

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

$$S \text{ is secure} \quad \equiv \quad (HH \ ; S \ \subseteq \ S \ ; HH) \tag{10}$$

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

$$\begin{array}{ll}
 & HH; S \ \subseteq \ S; HH \\
= & \quad \{ \text{ Definition of relational containment } \} \\
 & (\forall y, w \ :: \ y \langle HH; S \rangle w \quad \Rightarrow \quad y \langle S; HH \rangle w \ ) \\
= & \quad \{ \text{ Definition of relational composition, twice } \}
\end{array}$$

$$
\begin{array}{ll}
& (\forall\, y,w \,::\, (\exists\, x \,::\, y\langle HH\rangle x \;\wedge\; x\langle S\rangle w\,) \\
& \qquad \Rightarrow\; (\exists\, z \,::\, y\langle S\rangle z \;\wedge\; z\langle HH\rangle w\,)) \\
= & \qquad \{\ \text{Relational semantics of } HH \text{, shunting between range and term}\ \} \\
& (\forall\, y,w \,::\, (\exists\, x \,::\, y.k = x.k \;\wedge\; x\langle S\rangle w\,) \\
& \qquad \Rightarrow\; (\exists\, z \,:\, y\langle S\rangle z \,:\, z.k = w.k\,)) \\
= & \qquad \{\ \text{Predicate calculus}\ \} \\
& (\forall\, y,w \,::\, (\forall\, x \,::\, y.k = x.k \;\wedge\; x\langle S\rangle w \\
& \qquad\qquad \Rightarrow\; (\exists\, z \,:\, y\langle S\rangle z \,:\, z.k = w.k\,))) \\
= & \qquad \{\ \text{Unnesting of quantifiers}\ \} \\
& (\forall\, x,y,w \,::\, y.k = x.k \;\wedge\; x\langle S\rangle w \\
& \qquad \Rightarrow\; (\exists\, z \,:\, y\langle S\rangle z \,:\, z.k = w.k\,)) \\
= & \qquad \{\ \text{Nesting, shunting}\ \} \\
& (\forall\, x,y \,:\, y.k = x.k \,:\, (\forall\, w \,::\, x\langle S\rangle w \\
& \qquad\qquad\qquad \Rightarrow\; (\exists\, z \,:\, y\langle S\rangle z \,:\, z.k = w.k\,))) \\
= & \qquad \{\ \text{Set calculus}\ \} \\
& (\forall\, x,y \,:\, y.k = x.k \,:\, \{\, w \,:\, x\langle S\rangle w \,:\, w.k\,\} \subseteq \{\, z \,:\, y\langle S\rangle z \,:\, z.k\,\}) \\
= & \qquad \{\ \text{Expression is symmetric in } x \text{ and } y\ \} \\
& (\forall\, x,y \,:\, y.k = x.k \,:\, \{\, w \,:\, x\langle S\rangle w \,:\, w.k\,\} \;=\; \{\, z \,:\, y\langle S\rangle z \,:\, z.k\,\})
\end{array}
$$

Thus, we have established that, for any S,

$$
S \text{ is secure} \;\equiv\; (\forall\, x,y \,:\, x.k = y.k \,:\; \{\, w \,:\, x\langle S\rangle w \,:\, w.k\,\} = \{\, z \,:\, y\langle S\rangle z \,:\, z.k\,\}\,) \tag{11}
$$

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 [BBLM94] as the definition of secure information flow. (Similar definitions, restricted to the deterministic case, have appeared elsewhere [VSI96,VS97b].) Thus, one may view the derivation above as a proof of the equivalence of (7) with respect to the notion used by others.

4 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 [Dij76]. 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.

Recall that for any program S, the predicate transformers $wlp.S$ ("weakest liberal precondition") and $wp.S$ ("weakest precondition") 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.

The two predicate transformers are related by the following pairing property: For any program S ,

$$(\forall p :: wp.S.p \;=\; wlp.S.p \wedge wp.S.true) \tag{12}$$

We assume that for all statements S considered here, the predicate transformer $wlp.S$ is *universally conjunctive* (i.e., it distributes over arbitrary conjunctions), and (hence) monotonic [DS90].

We start by introducing some notation. For any program with a variable named v , we define the unary predicate transformer $[v : \_]$ (read " v -everywhere") as follows: For any predicate p ,

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

where M ranges over the domain of v . This unary predicate transformer has all the properties of universal quantification; in particular, it is universally conjunctive. Furthermore, for any variables v, w and any predicate p , we have

$$[v : [w : p]] \;=\; [w : [v : p]]$$

Recall that we are interested in programs whose variables are partitioned into k and h . For any such program, we write $[p]$ (read "everywhere p ") as shorthand for $[h : [k : p]]$. The wlp and wp semantics of the program HH are:

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

Program equality in the weakest precondition calculus is given by equality of wp and wlp :

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

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

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

Using this definition of program equality, we now rewrite security condition (7) in the weakest precondition calculus as follows.

$$\begin{array}{ll}
 & HH \,;\, S \,;\, HH \doteq S \,;\, HH \\
= & \qquad \{ \text{ Program equality in terms of } wlp \text{ and } wp \; \} \\
 & \quad (\forall p :: [wlp.(HH \,;\, S \,;\, HH).p \;\equiv\; wlp.(S \,;\, HH).p]) \\
 & \wedge\; [wp.(HH \,;\, S \,;\, HH).true \;\equiv\; wp.(S \,;\, HH).true] \\
= & \qquad \{ \; wlp \text{ and } wp \text{ of } HH \text{ and } ; \; \} \\
 & \quad (\forall p :: [[h : wlp.S.[h : p]] \;\equiv\; wlp.S.[h : p]]) \qquad (13) \\
 & \wedge\; [[h : wp.S.true] \;\equiv\; wp.S.true] \qquad (14)
\end{array}$$

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

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

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.

4.0 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 *,*

$$q \in Cyl \quad \equiv \quad [\, q \;\equiv\; [h \,:\, q]\,] \tag{15}$$

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. $[\, q \;\Rightarrow\; [h \,:\, q]\,]$
iii. $(\exists\, p :: [\, q \;\equiv\; [h \,:\, p]\,]\,)$
iv. $\neg q \in Cyl$

Proof. Follows from predicate calculus. *(End of Proof.)*

4.1 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 (13) as follows.

$$\begin{array}{ll}
& (\forall\, p :: [\,[h \,:\, wlp.S.[h \,:\, p]\,] \;\equiv\; wlp.S.[h \,:\, p]\,]\,) \\
= & \qquad \{\ \text{Definition of } Cyl \text{ (15)}\ \} \\
& (\forall\, p :: wlp.S.[h \,:\, p] \in Cyl\,) \\
= & \qquad \{\ \text{One-point rule}\ \} \\
& (\forall\, p, q : [\, q \;\equiv\; [h \,:\, p]\,] : wlp.S.q \in Cyl\,) \\
= & \qquad \{\ \text{Nesting and trading}\ \} \\
& (\forall\, q :: (\forall\, p :: [\, q \;\equiv\; [h \,:\, p]\,] \;\Rightarrow\; wlp.S.q \in Cyl\,)) \\
= & \qquad \{\ \text{Predicate calculus}\ \} \\
& (\forall\, q :: (\exists\, p :: [\, q \;\equiv\; [h \,:\, p]\,]\,) \;\Rightarrow\; wlp.S.q \in Cyl\,) \\
= & \qquad \{\ \text{Lemma 0.iii, and trading}\ \} \\
& (\forall\, q : q \in Cyl : wlp.S.q \in Cyl\,)
\end{array}$$

Similarly, we rewrite the expression (14) as follows.

$$\begin{array}{ll}
& [\,[h \,:\, wp.S.true] \;\equiv\; wp.S.true] \\
= & \qquad \{\ \text{Definition of } Cyl \text{ (15)}\ \} \\
& wp.S.true \in Cyl
\end{array}$$

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

$$S \text{ is secure} \;\equiv\; (\forall\, p : p \in Cyl : wlp.S.p \in Cyl\,) \;\wedge\; wp.S.true \in Cyl \tag{16}$$

5 A simpler characterization

Using (16) to check whether a given program S is secure requires evaluation of the following term:

$$(\forall\, p \;:\; p \in \mathit{Cyl} \;:\; wlp.S.p \in \mathit{Cyl}\,) \tag{17}$$

Since this term involves a quantification over all cylinders, it is somewhat 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 .

To explain how this simplification is brought about, we introduce the notions of *conjunctive* and *disjunctive spans*. These notions are defined formally below, but, informally speaking, for any set X of predicates, the conjunctive span $\mathcal{A}.X$ is the set of predicates obtained by taking conjunctions over the subsets of X . Similarly, the disjunctive span $\mathcal{E}.X$ is the set of predicates obtained by taking disjunctions over the subsets of X . The main theorem of this section asserts that the range of p in (17) may be replaced by any set of predicates whose conjunctive span is the set Cyl . The usefulness of the theorem is demonstrated in subsection 5.1, where we show that there is a simple set of predicates whose conjunctive span is Cyl .

We use the following notational conventions in this section. For any set X of predicates, we write $\neg\, X$ to mean the set $\{\, q \;:\; q \in X \;:\; \neg\, q \,\}$. We also write $\forall.X$ to mean the conjunction of all the predicates in X , and $\exists.X$ to mean the disjunction of all the predicates in X . Note that as a result of these conventions, we have $[\neg(\forall.X) \;\equiv\; \exists.(\neg X)]$.

5.0 Spans

For any set X of predicates, define the sets $\mathcal{A}.X$ and $\mathcal{E}.X$ as follows.

Definition (Spans)

$$\begin{aligned} \mathcal{A}.X &= \{\, XS \;:\; XS \subseteq X \;:\; \forall.XS \,\} \\ \mathcal{E}.X &= \{\, XS \;:\; XS \subseteq X \;:\; \exists.XS \,\} \end{aligned}$$

The two notions are related by the following lemma.

Lemma 1 *For any set X of predicates,*

$$\neg\, \mathcal{E}.X \;=\; \mathcal{A}.(\neg\, X)$$

Proof. Follows from predicate calculus. *(End of Proof.)*

We are now ready to present the main theorem of this section.

Theorem 0 *Let f be a universally conjunctive predicate transformer and let X be any set of predicates. Then*

$$(\forall\, p \;:\; p \in X \;:\; f.p \in \mathit{Cyl}\,) \;\equiv\; (\forall\, q \;:\; q \in \mathcal{A}.X \;:\; f.q \in \mathit{Cyl}\,)$$

Proof. Note that the left-hand side follows from the right-hand side since $X \subseteq \mathcal{A}.X$; thus, it remains to prove the implication

$$(\forall p : p \in X : f.p \in Cyl) \;\Rightarrow\; (\forall q : q \in \mathcal{A}.X : f.q \in Cyl)$$

We prove this implication by showing that for any predicate q in $\mathcal{A}.X$, the antecedent implies that $f.q$ is a cylinder. By the definition of a conjunctive span, there is a subset XS of X such that $[q \equiv \forall.XS]$. From the definition of cylinders (15), $XS \subseteq X$, and the antecedent, we have:

$$(\forall p : p \in XS : [f.p \equiv [h : f.p]]) \tag{18}$$

Now, we observe:

$$
\begin{array}{ll}
 & f.q \in Cyl \\
= & \quad \{ \text{ Choice of } XS \text{ } \} \\
 & f.(\forall p : p \in XS : p) \in Cyl \\
= & \quad \{ \text{ } f \text{ is universally conjunctive } \} \\
 & (\forall p : p \in XS : f.p) \in Cyl \\
= & \quad \{ \text{ Definition of cylinders (15) } \} \\
 & [(\forall p : p \in XS : f.p) \equiv [h : (\forall p : p \in XS : f.p)]] \\
= & \quad \{ \text{ } [h : \_] \text{ is universally conjunctive } \} \\
 & [(\forall p : p \in XS : f.p) \equiv (\forall p : p \in XS : [h : f.p])] \\
= & \quad \{ \text{ (18) } \} \\
 & [(\forall p : p \in XS : f.p) \equiv (\forall p : p \in XS : f.p)] \\
= & \quad \{ \text{ Predicate Calculus } \} \\
 & true
\end{array}
$$

(End of Proof.)

From the standpoint of mechanical verification, the usefulness of the result above is due to the fact that there is a simple set of predicates whose conjunctive span is Cyl .

5.1 A simpler quantification

Consider the following two sets of predicates, where M ranges over the domain of k .

$$PP = \{ M :: \text{“}k = M\text{”} \} \tag{19}$$

$$NN = \{ M :: \text{“}k \neq M\text{”} \} \tag{20}$$

It follows directly from these definitions that

$$PP = \neg NN \tag{21}$$

The relationship of these sets to Cyl is given by the following lemma.

Lemma 2 *With PP and NN as defined above, we have*

i. $\mathcal{E}.PP = Cyl$

ii. $\mathcal{A}.NN \;=\; Cyl$

Proof. We give an informal sketch of the proof here; details are left to the reader. By definition, Cyl consists of exactly those predicates that are independent of h, that is, they depend on the variable k only. But every predicate on k may be written as a disjunction of predicates, one for each value in the domain of k for which the predicate holds; thus part (i) follows. Part (ii) follows directly from part (i), observation (21), Lemma 1, and Lemma 0.iv. *(End of Proof.)*

Using the fact that $wlp.S$ is universally conjunctive for any statement S, and Lemma 2.ii, we apply Theorem 0 with $f, X \;:=\; wlp.S, NN$ to obtain the following reformulation of (16):

$$S \text{ is secure} \;\equiv\; (\forall\, q \;:\; q \in NN \;:\; wlp.S.q \in Cyl\;) \;\wedge\; wp.S.true \in Cyl.$$

Applying the definition of NN (20), this yields the following condition:

$$S \text{ is secure} \;\equiv\; (\forall\, M \;::\; wlp.S.(k \neq M) \in Cyl\;) \;\wedge\; wp.S.true \in Cyl \qquad (22)$$

Note that this is simpler than (16) since the quantification ranges over the domain of k.

5.2 Deterministic programs

In the case that S is also known to be deterministic and non-miraculous, we can further simplify the security condition (22). Recall that a deterministic, non-miraculous program S satisfies the following properties [DS90]:

$$(\forall\, p \;::\; [wp.S.p \;\equiv\; \neg wlp.S.(\neg p)]\;) \qquad (23)$$

$wp.S$ is universally disjunctive (24)

Consequently, the term involving wp in (22) is subsumed by the term involving wlp:

$$\begin{array}{ll}
 & wp.S.true \in Cyl \\
= & \quad \{\;\; (23), \text{ with } p := true \;\;\} \\
 & \neg wlp.S.false \in Cyl \\
= & \quad \{\;\; \text{Lemma 0.iv} \;\;\} \\
 & wlp.S.false \in Cyl \\
\Leftarrow & \quad \{\;\; false \in Cyl \;\;\} \\
 & (\forall\, q \;:\; q \in Cyl \;:\; wlp.S.q \in Cyl\;)
\end{array}$$

Thus, the security condition for deterministic S is given by

$$S \text{ is secure} \;\equiv\; (\forall\, M \;::\; wlp.S.(k \neq M) \in Cyl\;) \qquad (25)$$

Next, we show that condition (25) may also be expressed in terms of wp and PP instead of wlp and NN.

$$
\begin{array}{ll}
 & (\forall q \ : \ q \in \mathit{Cyl} \ : \ \mathit{wlp}.S.q \in \mathit{Cyl}\ ) \\
= & \qquad \{ \ \text{Theorem 0} \ \} \\
 & (\forall q \ : \ q \in \mathit{NN} \ : \ \mathit{wlp}.S.q \in \mathit{Cyl}\ ) \\
= & \qquad \{ \ \text{Negation is its own inverse, so rename dummy } q \text{ to } \neg p \ \} \\
 & (\forall p \ : \ \neg p \in \mathit{NN} \ : \ \mathit{wlp}.S.(\neg p) \in \mathit{Cyl}\ ) \\
= & \qquad \{ \ \text{Observation (21), and (23)} \ \} \\
 & (\forall p \ : \ p \in \mathit{PP} \ : \ \neg \mathit{wp}.S.p \in \mathit{Cyl}\ ) \\
= & \qquad \{ \ \text{Lemma 0.iv} \ \} \\
 & (\forall p \ : \ p \in \mathit{PP} \ : \ \mathit{wp}.S.p \in \mathit{Cyl}\ )
\end{array}
$$

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

$$S \text{ is secure} \ \equiv \ (\forall M \ :: \ \mathit{wp}.S.(k = M) \in \mathit{Cyl}\ ) \qquad (26)$$

5.3 Examples

We give some examples to show our formulae at work.

Firstly, consider the secure program (2). We calculate,

$$
\begin{array}{ll}
 & (k := h \,;\, k := k - h) \ \text{ is secure} \\
= & \qquad \{ \ \text{Security condition (22)} \ \} \\
 & \quad (\forall M \ :: \ \mathit{wlp}.(k := h \,;\, k := k - h).(k \neq M) \in \mathit{Cyl}\ ) \\
 & \wedge \ \mathit{wp}.(k := h \,;\, k := k - h).\mathit{true} \in \mathit{Cyl} \\
= & \qquad \{ \ \mathit{wlp} \text{ and } \mathit{wp} \text{ of } := \text{ and } ; \ \} \\
 & (\forall M \ :: \ (h - h \neq M) \in \mathit{Cyl}\ ) \ \wedge \ \mathit{true} \in \mathit{Cyl} \\
= & \qquad \{ \ \text{Lemma 0.i, and } \mathit{true} \in \mathit{Cyl} \ \} \\
 & (\forall M \ :: \ [h - h \neq M \ \equiv \ [h \ : \ h - h \neq M]\,]\ ) \\
= & \qquad \{ \ \text{Arithmetic} \ \} \\
 & (\forall M \ :: \ [0 \neq M \ \equiv \ [h \ : \ 0 \neq M]\,]\ ) \\
= & \qquad \{ \ \text{Definition of } [h \ : \ \_] \text{, predicate calculus} \ \} \\
 & \mathit{true}
\end{array}
$$

This shows that our method does indeed establish that program (2) is secure.

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

$$
\begin{array}{ll}
 & (\textbf{if } h = 0 \longrightarrow \mathit{skip} \ [\!] \ \mathit{true} \longrightarrow \mathit{loop}\ \textbf{fi}) \ \text{ is secure} \\
= & \qquad \{ \ \text{Security condition (22)} \ \} \\
 & \quad (\forall M \ :: \ \mathit{wlp}.(\textbf{if } h = 0 \longrightarrow \mathit{skip} \ [\!] \ \mathit{true} \longrightarrow \mathit{loop}\ \textbf{fi}).(k \neq M) \in \mathit{Cyl}\ ) \\
 & \wedge \ \mathit{wp}.(\textbf{if } h = 0 \longrightarrow \mathit{skip} \ [\!] \ \mathit{true} \longrightarrow \mathit{loop}\ \textbf{fi}).\mathit{true} \in \mathit{Cyl} \\
= & \qquad \{ \ \mathit{wlp} \text{ and } \mathit{wp}\text{, using } (\forall p \ :: \ [\mathit{wlp}.\mathit{loop}.p \ \equiv \ \mathit{true}]\ ) \\
 & \qquad\qquad\qquad\qquad \text{and } [\mathit{wp}.\mathit{loop}.\mathit{true} \ \equiv \ \mathit{false}] \ \} \\
 & \quad (\forall M \ :: \ ((h = 0 \ \Rightarrow \ k \neq M) \ \wedge \ (\mathit{true} \ \Rightarrow \ \mathit{true})) \in \mathit{Cyl}\ ) \\
 & \wedge \ ((h = 0 \ \vee \ \mathit{true}) \ \wedge \ (h = 0 \ \Rightarrow \ \mathit{true}) \ \wedge \ (\mathit{true} \ \Rightarrow \ \mathit{false})) \in \mathit{Cyl} \\
= & \qquad \{ \ \text{Predicate Calculus} \ \} \\
 & (\forall M \ :: \ (h = 0 \ \Rightarrow \ k \neq M) \in \mathit{Cyl}\ ) \ \wedge \ \mathit{false} \in \mathit{Cyl} \\
= & \qquad \{ \ \text{Lemma 0.i, and } \mathit{false} \in \mathit{Cyl} \ \}
\end{array}
$$

$$
\begin{array}{ll}
& (\forall\, M :: [h = 0 \;\Rightarrow\; k \neq M \;\;\equiv\;\; [h : h = 0 \;\Rightarrow\; k \neq M]\,]\,) \\
\Rightarrow & \quad \{ \text{ Instantiate with } M := 2 \; \} \\
& [h = 0 \;\Rightarrow\; k \neq 2 \;\;\equiv\;\; [h : h = 0 \;\Rightarrow\; k \neq 2]\,] \\
= & \quad \{ \; [p] \text{ is shorthand for } [h : [k : p]\,] \; \} \\
& [h : [k : h = 0 \;\Rightarrow\; k \neq 2 \;\;\equiv\;\; [h : h = 0 \;\Rightarrow\; k \neq 2]\,]\,] \\
= & \quad \{ \text{ Definition of } [v : \_] \text{, twice; } wlp \text{ of } := \; \} \\
& (\forall\, M, N :: wlp.(k, h := M, N).(\quad h = 0 \;\Rightarrow\; k \neq 2 \\
& \qquad\qquad\qquad\qquad\qquad\qquad \equiv \;\; [h : h = 0 \;\Rightarrow\; k \neq 2]\,)\,) \\
\Rightarrow & \quad \{ \text{ Instantiate with } M, N := 2, 2 \; \} \\
& 2 = 0 \;\Rightarrow\; 2 \neq 2 \;\;\equiv\;\; [h : h = 0 \;\Rightarrow\; 2 \neq 2] \\
= & \quad \{ \text{ Predicate Calculus, identity of } \equiv \; \} \\
& [h : h \neq 0] \\
= & \quad \{ \text{ Definition of } [h : \_] \; \} \\
& (\forall\, N :: wlp.(h := N).(h \neq 0)\,) \\
\Rightarrow & \quad \{ \text{ Instantiate } N := 0 \; \} \\
& 0 \neq 0 \\
= & \quad \{ \text{ Predicate Calculus } \} \\
& \mathit{false}
\end{array}
$$

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

$$
\begin{array}{ll}
& (\textbf{if } h = 0 \longrightarrow loop \;\square\; h \neq 0 \longrightarrow skip \textbf{ fi}) \text{ is secure} \\
= & \quad \{ \text{ Security condition for deterministic programs (25) } \} \\
& (\forall\, M :: wlp.(\textbf{if } h = 0 \longrightarrow loop \;\square\; h \neq 0 \longrightarrow skip \textbf{ fi}).(k \neq M) \in Cyl\,) \\
= & \quad \{ \; wlp \; \} \\
& (\forall\, M :: ((h = 0 \;\Rightarrow\; true) \;\wedge\; (h \neq 0 \;\Rightarrow\; k \neq M)) \in Cyl\,) \\
= & \quad \{ \text{ Definition of } Cyl \text{ : note } h \neq 0 \;\Rightarrow\; k \neq M \text{ depends on } h \; \} \\
& \mathit{false}
\end{array}
$$

6 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 [Den76], which is based on the Bell and La Padula security model [BLP73]. Most approaches to static certification of secure information flow (an area pioneered by Denning and Denning [Den76,DD77]) seem to fall into one of two general categories: type systems and data flow analysis techniques. In this section, we discuss these approaches and compare them to our work. A historical perspective of secure information flow appears in a book by Gasser [Gas88].

6.0 Approaches based on type systems

The static certification mechanism proposed by Denning and Denning [DD77] 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 $\leq$, 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,

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

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 , $class.E \leq class.x$
1. For every conditional statement $\mathbf{if}\ E\ \mathbf{then}\ T\ \mathbf{else}\ U\ \mathbf{end}$ in S , $class.E \leq class.T$ and $class.E \leq class.U$.

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 [VSI96,VS97b].

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 [VS97a] 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.)

6.1 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

$$x' := y' \uparrow z' \uparrow \mathit{local}'$$

For a conditional statement in S such as

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

S' contains a corresponding statement

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

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

$$\{\, k' \leq \bot \;\; \wedge \;\; h' \leq \top \;\; \wedge \;\; \mathit{local}' \leq \bot \,\} \quad S' \quad \{\, k' \leq \bot \,\} \tag{27}$$

The first data flow analysis approach of this kind was given by Andrews and Reitman [AR80], whose treatment also dealt with communicating sequential processes. Banâtre *et al.* [BBLM94] 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 [VSI96].)

The data flow analysis approach can provide more precision than the type system approach. For example, the approach certifies 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 $h - h = \mathit{0}$, 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 [AR80]. 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 (27) 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).

6.2 The use of determinism

It has been noted elsewhere that "semantic models always make implicit assumptions about what sort of things are interesting about a process' behaviour" [Ros95]. In the context of security, these assumptions specify what we consider observable by the adversary. We argued in section 2 that our definition can be used to model adversaries that exploit covert flows (*e.g.*, adversaries monitoring resource usage) by appropriately choosing the low-security variables. There is, however, one subtle issue that arises in the context of nondeterminism: security is not preserved by refinement. For example, the secure program "assign to k an arbitrary value" is refined by the insecure program "$k := h$". Since sequential programs are often implemented by refining their nondeterminism, this leads to the undesirable situation in which a secure program is rendered insecure by its implementation.

There are two ways of addressing this issue. The first is by recognizing 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 during implementation (or how nondeterministic choices are resolved during execution), we can assert that its observations reveal no information about the initial value of h . The second way of addressing the issue is by noting that the problem does not arise for deterministic programs, since the latter are maximal in the refinement ordering. Thus we can avoid the difficulty by requiring that secure programs be deterministic. [1] This latter approach is similar to the one advocated by Roscoe, 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.

7 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. We are currently investigating ways of using our characterization as a basis for developing a mechanically-assisted technique for verifying secure flow.

Acknowledgments. We are grateful to the following colleagues for sharing their insights and comments on our work: Martín Abadi, Ernie Cohen, Rutger M. Dijkstra, Mark Lillibridge, Jayadev Misra, Greg Nelson, Raymie Stata, 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 four anonymous referees.

[1] Actually, it suffices to place the weaker requirement that programs be deterministic with respect to the low-security variables in the following sense: the initial state determines the final value of k .

References

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, January 1980.

[BBLM94] Jean-Pierre Banâtre, Ciarán Bryce, and Daniel Le Métayer. Compile-time detection of information flow in sequential programs. In *Proceedings of the European Symposium on Research in Computer Security*, pages 55–73. Lecture Notes in Computer Science 875, Sprinter Verlag, 1994.

[BLP73] D. E. Bell and L. J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corporation, Bedford, Massachusetts, 1973.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Gas88] Morrie Gasser. *Building a secure computer system*. Van Nostrand Reinhold Company, New York, 1988.

[Heh84] Eric C. R. Hehner. Predicative programming Part I. *Communications of the ACM*, 27(2):134–143, February 1984.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[RMD92] R.M. Dijkstra. Relational calculus and relational program semantics. Eindhoven Institute of Technology, 1992.

[Ros95] A. W. Roscoe. CSP and determinism in security modelling. In *Security and Privacy*. IEEE, 1995.

[VS97a] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[VS97b] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer, April 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.