SRC Technical Note
1997 - 032

17 December 1997

A Semantic Approach to Secure Information Flow

K. Rustan M. Leino and Rajeev Joshi

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Abstract

A classic problem in security is the problem of determining whether a given program has *secure information flow*. Informally, this problem may be described as follows: Given a program operating on public and private variables, check whether observations of the public variables before and after execution reveal any information about the initial values of the private variables. Although the 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. This paper presents a considerably different approach to checking secure information flow, based on a semantic characterization of security. A semantic approach has several desirable features. Firstly, it gives a more precise characterization of security than that possible by conservative methods based on type systems. Secondly, it applies to any programming constructs whose semantics are definable; for instance, nondeterminism and exceptions pose no additional problems. Thirdly, it can be applied to reasoning about indirect leaking of information through variations in program behavior (*e.g.*, whether or not the program terminates). The method is also useful in the context of automated verification, since it can be used to develop a mechanically-assisted technique for checking whether the flow of a given program is secure.

A Semantic Approach to Secure Information Flow

K. Rustan M. Leino
DEC SRC, Palo Alto, CA
rustan@pa.dec.com

Rajeev Joshi
University of Texas, Austin, TX
joshi@cs.utexas.edu

17 December 1997

0 Introduction

A classic problem in security is the problem of determining whether a given computer program has *secure information flow* [BLP73, Den76]. In its simplest form, the problem may be described informally as follows: Given a program operating on "high-security" and "low-security" variables, check whether observations of the initial and final values 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 may be leaked by variations in program behavior [Lam73]. For instance, it may be possible to conclude 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).

The problem of secure information flow is more important today than ever before, because more and more computers are becoming connected. The risk of information being leaked inappropriately is a concern in protecting high-security military and financial data as well as in protecting data stored on personal computers. For example, with the advent of the Web, there is a danger that a downloaded applet will leak private data (such as the contents of a local file) by communicating it to other sites.

Most previous approaches to checking secure information flow 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 possible by conservative methods based on types. Secondly, it is applicable to any programming construct whose semantics are defined; for instance, nondeterminism and exception handling constructs pose no additional problems. Thirdly, it can be applied to reasoning about a variety of covert channels, including termination behavior and timing channels. Finally, the method is also useful in the context of automated verification, since it can be used to develop a mechanically-assisted technique for certifying program flow.

The outline of the rest of the paper is as follows. We start in section 1 by informally defining the problem and discussing several small examples. We present the formal characterization of security in section 2. In section 3, we relate our definition to a 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. Finally, we discuss related work in section 6 and give a short summary in section 7.

1 Informal description of the problem

Throughout this paper, we assume that for each program that we consider, the program 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 give any information about the initial value of h.

We illustrate this notion with a number of program examples.

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,

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

is insecure, despite the fact that each branch of the conditional is secure. This program has the same effect as $k := h$, but 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 leaks *any* information about h. For example, neither of the two programs

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

transmits the entire value of h, but each of the programs is insecure because the final value of k does reveal some information about the initial value of h.

A nondeterministic program can be insecure even if an adversary has no control over 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 $S \ \square\ T$ proceeds by choosing any one of S or T and executing it.) The program

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

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

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

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

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

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

(where loop is the program that never terminates) is insecure, because whether or not the program terminates depends on the initial value of h. In addition, each of the nondeterministic programs

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

is also insecure. Program (5) terminates only if the initial value of h is 0; even though there is always a possibility that the program will loop forever, if the program is observed to terminate, the initial value of h is known. Similarly, if an adversary is able to detect infinite looping, then the nontermination of program (6) indicates that the initial value of h is 0.

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

In this section, we give a formal characterization of secure information flow. As stated in the introduction, our characterization is expressed as an equality between two program terms. We use the symbol $\doteq$ to denote program equality and we 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. Thus, the programs S and $HH ; S$ are equal provided that S does not depend on the initial value of h. Secondly, observe that the program $S ; HH$ "discards" the final value of h resulting from the execution of S. These observations help in giving an informal justification to the following definition of security.

Definition 0 (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, not of h, is of interest. 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 an 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 [Hoa96] and different choices are suitable for different purposes. For instance, in this paper, we will use a relational semantics to establish the correctness of our definition, but we will use weakest precondition semantics to obtain a definition 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 (*cf.* [Cri84, LvdS94]) pose no additional problems in this approach, nor do data structures such as arrays, records, or objects (*cf.* [Hoa72, Lei97]). 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 channels, 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 timing channels by introducing in k a program variable that models execution time.

3 Security in the relational calculus

In this section, we justify our characterization of security by showing that it is equivalent to a notion used elsewhere in the literature. Since that notion was given in operational terms, we find it convenient to use a relational semantics. Thus, for the purposes of this section, a program is a relation over the state space formed by variables h and k.

We use the following notational conventions. The identifiers w, x, y, z denote elements of a state space. We write $x.k$ and $x.h$ to denote the values of k and h in a 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 pre-state x to the post-state y. 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 right) identity of composition and that $;$ is monotonic with respect to $\subseteq$ in each of its 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 $t.j$ for all j satisfying $r.j$. We call j the *dummy*, $r.j$ the *range*, and $t.j$ the *term* of the quantification. When the range is *true* or is understood from context, we omit it. We use similar notation and conventions to define sets, by writing

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

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

The relational semantics of program HH are given by

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

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.

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

Since the second and last lines are equal, we have equivalence throughout; thus, we have shown

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

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

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

Thus, we have established that, for any S,

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

This condition says that the set of possible final values of k is independent of the initial value of h. The condition has appeared in the literature [BBLM94] as the definition of secure information flow, and similar definitions for deterministic programs have been used elsewhere [VSI96, VS97b]. Thus, one may view the derivation above as a proof of the soundness and completeness of our characterization (7) with respect to the notion used by others.

4 Security in the weakest precondition calculus

In this section, we show how our definition of secure information flow may be written succinctly in the weakest precondition calculus [Dij76]. In section 5, we go one step further by showing how to write this weakest precondition formulation of security as a first-order predicate.

We begin by introducing some useful notation. Recall that we are considering programs whose variables are partitioned by k and h. Thus, every predicate on the state space of such a program depends only on the variables k and h. We will use identifiers p and q to denote such predicates; for any such p we will write $[p]$ (read "everywhere p") as an abbreviation for $(\forall k, h :: p)$. For any program S, the predicate transformers $wlp.S$ ("weakest liberal precondition") and $wp.S$ ("weakest precondition") are informally defined as follows [Dij76]: 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 [Dij76]: For any programs S and T,

$$(\forall p :: [wp.S.p \quad \equiv \quad wlp.S.p \wedge wp.S.true]) \tag{12}$$

The predicate transformer $wlp.S$ is *universally conjunctive*, that is, it distributes over arbitrary conjunctions. Universal conjunctivity implies monotonicity [DS90]. The *wlp* and *wp* semantics of the program HH are:

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

Program equality in the weakest precondition calculus is defined as the equality of *wp* and *wlp* on each predicate; thus

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

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

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

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

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

The last formula above contains expressions having the form of a predicate q satisfying

$$[(\forall h :: q) \equiv q]$$

Predicates with this property occur often in our calculations, so it will be 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

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

Definition 1 (Cylinders) *For any predicate q,*

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

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.*

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

Proof. Follows from simple predicate calculus. □

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.

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

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

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

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 first-order characterization

Checking whether a given program S is secure using condition (16) given in the previous section involves checking that certain predicates are cylinders; this can be done with a theorem prover. (Checking whether a given predicate is a cylinder can also be done by the considerably cheaper and more conservative method of syntactically scanning the predicate for free occurrences of h.) Thus, checking the second conjunct on the right-hand side of (16) is straightforward. However, checking the first conjunct, namely

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

involves a quantification over predicates; thus, it is not well-suited for automated checking (nor does the syntactic scan for free occurrences of h work). In this section, we show how this higher-order quantification over p can be reduced to a first-order quantification over the domain of k.

To explain how the reduction is brought about, we need two new notions, namely that of *conjunctive* and *disjunctive spans*. Informally speaking, for any set X of predicates, the conjunctive (disjunctive) span of X, denoted $\mathcal{A}.X$ ($\mathcal{E}.X$), is the set of predicates obtained by taking conjunctions (disjunctions) over subsets of X. The main theorem of this section asserts that the range of p in the quantification (17) above 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 exist small sets of predicates whose conjunctive span is Cyl.

We will 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, we define sets $\mathcal{A}.X$ and $\mathcal{E}.X$ as follows.

Definition 2 (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, whose proof follows from simple predicate calculus.

Lemma 1 *For any set X of predicates,*

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

We are now ready to present the main theorem of this section, which states that one can establish the quantified expression

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

by establishing a quantified expression over a smaller range.

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

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

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. So, let q be any predicate in $\mathcal{A}.X$. By the definition of a conjunctive span, there is a subset XS of X such that $[q \;\equiv\; \forall.XS]$. Observe:

$$
\begin{array}{ll}
 & (\forall p \;:\; p \in X \;:\; f.p \in Cyl\,) \\
\Rightarrow & \quad \{ \;\; XS \subseteq X \text{, antimonotonicity of } \forall \text{ in its range } \; \} \\
 & (\forall p \;:\; p \in XS \;:\; f.p \in Cyl\,) \\
= & \quad \{ \;\; \text{Lemma 0.iii} \;\; \} \\
 & (\forall p \;:\; p \in XS \;:\; [f.p \;\;\Rightarrow\;\; (\forall h \;::\; f.p\,)]\,) \\
\Rightarrow & \quad \{ \;\; f \text{ is universally conjunctive, hence monotonic, also } [\forall.XS \Rightarrow p] \;\; \} \\
 & (\forall p \;:\; p \in XS \;:\; [f.(\forall.XS) \;\;\Rightarrow\;\; (\forall h \;::\; f.p\,)]\,) \\
= & \quad \{ \;\; \text{Range } p \in XS \text{ is independent of } k \text{ and } h \;\; \} \\
 & [(\forall p \;:\; p \in XS \;:\; f.(\forall.XS) \;\;\Rightarrow\;\; (\forall h \;::\; f.p\,))] \\
= & \quad \{ \;\; \text{Antecedent is independent of } p \;\; \} \\
 & [f.(\forall.XS) \;\;\Rightarrow\;\; (\forall p \;:\; p \in XS \;:\; (\forall h \;::\; f.p\,))] \\
= & \quad \{ \;\; \text{Interchange quantifications, range } p \in XS \text{ is independent of } h \;\; \} \\
 & [f.(\forall.XS) \;\;\Rightarrow\;\; (\forall h \;::\; (\forall p \;:\; p \in XS \;:\; f.p\,))] \\
= & \quad \{ \;\; f \text{ is universally conjunctive} \;\; \} \\
 & [f.(\forall.XS) \;\;\Rightarrow\;\; (\forall h \;::\; f.(\forall p \;:\; p \in XS \;:\; p\,))] \\
= & \quad \{ \;\; \text{Notation ``}\forall.\text{'' described above} \;\; \} \\
 & [f.(\forall.XS) \;\;\Rightarrow\;\; (\forall h \;::\; f.(\forall.XS)\,)] \\
= & \quad \{ \;\; \text{Lemma 0.iii} \;\; \} \\
 & f.(\forall.XS) \in Cyl \\
= & \quad \{ \;\; \text{By assumption, } [q \;\equiv\; \forall.XS] \;\; \} \\
 & f.q \in Cyl
\end{array}
$$

□

From the standpoint of mechanical verification, the usefulness of the result above is due to the fact that there are sets of predicates much smaller than Cyl whose conjunctive span is Cyl. In the following subsection, we show such a set of predicates whose cardinality is the same as the cardinality of the domain of k.

5.1 A lower-order quantification

Consider the following two sets of predicates, where K ranges over the set of possible values of the variables k.

$$PP \;=\; \{K \;::\; k = K\,\} \tag{18}$$

$$NN \;=\; \{K \;::\; k \neq K\,\} \tag{19}$$

It follows directly from these definitions that

$$PP \;=\; \neg NN \tag{20}$$

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; thus part (i) follows. Part (ii) follows directly from part (i), observation (20), Lemma 1, and Lemma 0.v. □

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 result.

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

Using the definition of NN (19), this may be further simplified to the following.

$$S \text{ is secure} \quad \equiv \quad (\forall K \;::\; wlp.S.(k \neq K) \in Cyl\,) \;\;\wedge\;\; wp.S.true \in Cyl \qquad (21)$$

Note that this is simpler than (16) since the range of the quantification is the domain of k. Thus, in particular, security according to (21) is stated as a first-order formula.

5.2 Deterministic programs

In the case that S is also known to be deterministic, we can simplify the security condition (21) even further. A deterministic program S satisfies the following properties [DS90]:

$$(\forall p \;::\; [wp.S.p \;\equiv\; \neg wlp.S.(\neg p)]\,) \qquad (22)$$

$$wp.S \text{ is universally disjunctive} \qquad (23)$$

Consequently, the term involving wp in (21) is subsumed by the term involving wlp:

$$\begin{array}{ll} & wp.S.true \in Cyl \\ = & \quad \{\ (22), \text{ with } p := true \ \} \\ & \neg wlp.S.false \in Cyl \\ = & \quad \{\ \text{Lemma 0.v}\ \} \\ & wlp.S.false \in Cyl \\ \Leftarrow & \quad \{\ false \in Cyl\ \} \\ & (\forall q \;:\; q \in Cyl \;:\; wlp.S.q \in Cyl\,) \end{array}$$

Thus, the security condition for deterministic S is given by

$$S \text{ is secure} \quad \equiv \quad (\forall K :: wlp.S.(k \neq K) \in Cyl) \tag{24}$$

Next, we show that condition (24) may also be expressed in terms of wp and PP instead of wlp and NN.

$$
\begin{array}{cl}
 & (\forall q : q \in Cyl : wlp.S.q \in Cyl) \\
= & \quad \{ \text{ Theorem 0 } \} \\
 & (\forall q : q \in NN : wlp.S.q \in Cyl) \\
= & \quad \{ \text{ Negation is its own inverse, so rename dummy } q \text{ to } \neg p \ \} \\
 & (\forall p : \neg p \in NN : wlp.S.(\neg p) \in Cyl) \\
= & \quad \{ \text{ Observation (20), and (22) } \} \\
 & (\forall p : p \in PP : \neg wp.S.p \in Cyl) \\
= & \quad \{ \text{ Lemma 0.v } \} \\
 & (\forall p : p \in PP : wp.S.p \in Cyl)
\end{array}
$$

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

$$S \text{ is secure} \quad \equiv \quad (\forall K :: wp.S.(k = K) \in Cyl) \tag{25}$$

5.3 Examples

We give some examples to show our formulae at work.

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

$$
\begin{array}{cl}
 & (k := h \,;\, k := k - h) \text{ is secure} \\
= & \quad \{ \text{ Security condition (21) } \} \\
 & \quad (\forall K :: wlp.(k := h \,;\, k := k - h).(k \neq K) \in Cyl) \\
 & \wedge \ wp.(k := h \,;\, k := k - h).true \in Cyl \\
= & \quad \{ \ wlp \text{ and } wp \text{ of } := \text{ and } ; \ \} \\
 & (\forall K :: (h - h \neq K) \in Cyl) \ \wedge \ true \in Cyl \\
= & \quad \{ \text{ Lemma 0.ii, and } true \in Cyl \ \} \\
 & (\forall K :: [h - h \neq K \ \equiv \ (\forall h :: h - h \neq K)]) \\
= & \quad \{ \text{ Arithmetic } \} \\
 & (\forall K :: [0 \neq K \ \equiv \ (\forall h :: 0 \neq K)]) \\
= & \quad \{ \text{ Predicate calculus } \} \\
 & true
\end{array}
$$

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.

$(\mathbf{if}\ h = 0 \longrightarrow skip\ \square\ true \longrightarrow loop\ \mathbf{fi})$ is secure

$=$ { Security condition (21) }

$(\forall K :: wlp.(\mathbf{if}\ h = 0 \longrightarrow skip\ \square\ true \longrightarrow loop\ \mathbf{fi}).(k \neq K) \in Cyl\ )$

$\wedge\ wp.(\mathbf{if}\ h = 0 \longrightarrow skip\ \square\ true \longrightarrow loop\ \mathbf{fi}).true \in Cyl$

$=$ { wlp and wp,
using $(\forall p :: [wlp.loop.p \equiv true]\ )$ and $[wp.loop.true \equiv false]$ }

$(\forall K :: ((h = 0 \Rightarrow k \neq K) \wedge (true \Rightarrow true)) \in Cyl\ )$

$\wedge\ ((h = 0 \vee true) \wedge (h = 0 \Rightarrow true) \wedge (true \Rightarrow false)) \in Cyl$

$=$ { Predicate calculus }

$(\forall K :: (h = 0 \Rightarrow k \neq K) \in Cyl\ ) \wedge false \in Cyl$

$=$ { Lemma 0.ii, and $false \in Cyl$ }

$(\forall K :: [h = 0 \Rightarrow k \neq K \equiv (\forall h :: h = 0 \Rightarrow k \neq K\ )]\ )$

$=$ { $[p]$ is shorthand for $(\forall k, h :: p\ )$ }

$(\forall K, k, h :: h = 0 \Rightarrow k \neq K \equiv (\forall h :: h = 0 \Rightarrow k \neq K\ ))$

$\Rightarrow$ { Instantiate with $K, k, h := 2, 2, 2$ }

$2 = 0 \Rightarrow 2 \neq 2 \equiv (\forall h :: h = 0 \Rightarrow 2 \neq 2\ )$

$=$ { Arithmetic and predicate calculus }

$(\forall h :: h \neq 0\ )$

$\Rightarrow$ { Instantiate $h := 0$ }

$false$

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

$(\mathbf{if}\ h = 0 \longrightarrow loop\ \square\ h \neq 0 \longrightarrow skip\ \mathbf{fi})$ is secure

$=$ { Security condition for deterministic programs (24) }

$(\forall K :: wlp.(\mathbf{if}\ h = 0 \longrightarrow loop\ \square\ h \neq 0 \longrightarrow skip\ \mathbf{fi}).(k \neq K) \in Cyl\ )$

$=$ { wlp }

$(\forall K :: ((h = 0 \Rightarrow true) \wedge (h \neq 0 \Rightarrow k \neq K)) \in Cyl\ )$

$=$ { Definition of Cyl, since $h \neq 0 \Rightarrow k \neq K$ is not independent of h }

$false$

6 Related work

The problem of secure information flow has been studied for several decades. A frequently 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]. Static certification mechanisms for secure information flow, an area pioneered by Denning and Denning [Den76, DD77], seem to fall into two general flavors: type systems and data flow analysis techniques. In this section, we discuss each of these and compare them to our work. A historical perspective on secure information flow can be found, for example, in a book by Gasser [Gas88].

6.0 Type systems approach

The static certification mechanism proposed by Denning and Denning [DD77] can be described as a type checker for secure information flow. Each variable x occurring in a program is declared to have a particular *security class*, or *type*, written $class.x$. These security classes are related by a given lattice ordering $\leq$. The type checker computes the type of an expression as the lattice join of the types of its subexpressions. For example,

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

Each program statement also has a security class, computed as the lattice meet of the security classes of the statement's l-expressions. For example,

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

The type checker certifies a program S as being secure only if:

0. for every assignment statement $x := E$ in S, $class.E \leq class.x$, and
1. for every conditional statement **if** E **then** T **else** U **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 proof of the *soundness* of their certification mechanism, that is, a proof that the mechanism certifies only secure programs. Recently, Volpano *et al.* have given a more rigorous proof thereof [VSI96, VS97b].

The advantage of using a type system as the basis of a certification mechanism is that it caters for a simple implementation. However, because of the nature of type systems, this certification mechanism rejects any program that contains a subprogram that by itself would be insecure. But as we saw in examples (0)–(3) of section 1, an insecure program can occur as a subprogram of a secure program. In contrast, our approach does recognize programs like (0)–(3) as being secure.

Also, just like it would be too much to hope for a useful type system that would reject programs that may fail to terminate, it would be too much to hope for a useful type system that would reject programs that may leak information via their termination behavior. (Volpano and Smith [VS97a] have attempted such a type system; however, their type system rejects any program that mentions h in a loop guard, which can be construed as resigning the game before it starts.)

6.1 Data flow analysis approach

Just like there are data flow analysis techniques that let an optimizing compiler analyze a given program beyond type checking [ASU86], there are data flow analysis techniques for secure information flow. The idea is to conservatively find the program paths through which data may flow.

We choose to present the data flow analysis approach to secure information flow as a translation from a given program S to a program S' that captures and facilitates reasoning about the possible flows of S. For every variable x of S, program S' instead contains a variable $\underline{x}$, representing the security class of the value dynamically computed into x. To deal with implicit flows, S' also contains a special variable $\underline{local}$, representing the security class of the values used to compute the guards that led execution to the current instruction. We describe the translation by example. For every assignment statement of the form $x := y + z$ in S, S' contains a corresponding statement

$$\underline{x} := \underline{y} \uparrow \underline{z} \uparrow \underline{local}$$

where, as in the previous subsection, $\uparrow$ denotes the lattice join on the set of security classes. For every conditional statement

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

in S, S' contains a corresponding statement

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

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 $\bot \leq \top$), then " S is secure " can be expressed as the following Hoare triple on S':

$$\{\, \underline{k} \leq \bot \wedge \underline{h} \leq \top \wedge \underline{local} \leq \bot \,\} \quad S' \quad \{\, \underline{k} \leq \bot \,\} \tag{26}$$

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 this approach, akin to data flow analysis algorithms used in compilers, and wrestled with a proof of soundness for the approach. (Contrary to what our description of the approach above may suggest, Andrews and Reitman used an ordinary **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 programs that make use of the nondeterminism [VSI96].)

The data flow analysis approach can offer more precision than the type system approach. For example, the approach will certify the 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.

A way to reduce rejections of secure programs that result from the second of these reasons, one can combine the data flow analysis approach with a correctness logic, as mentioned by Andrews and Reitman [AR80]: Instead of rewriting program S into S', one can superimpose the new variables ($\underline{k}$, $\underline{h}$, $\underline{local}$) and their updates onto program S, and then reason about S using the Hoare triple (26) but on S instead of on S'. A consequence of this combination with a correctness logic is that one can rule out some impossible control paths, such as the one in program (3).

7 Summary

We have presented a simple and novel 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 channels. Unlike previous methods like type systems and compiler data flow analysis techniques, the characterization is in terms of the semantics of the program. Consequently, our method is more precise, and can therefore certify more secure programs, than previous methods.

We showed an application of our security characterization with a weakest precondition semantics, deriving a first-order predicate whose validity implies that an adversary cannot deduce any secure information from observing the public inputs, outputs, and termination behavior of the program. As far as we know, our approach is the only one that can analyze termination behavior in a useful way and is also the only one that handles nondeterminism correctly.

Future work includes employing our method in practice, and investigating the utility of our method in other areas, such as program slicing [Wei84].

Acknowledgments. We are grateful to the following colleagues for sharing their insights and comments on our work: Martín Abadi, Ernie Cohen, Mark Lillibridge, Jayadev Misra, Greg Nelson, Raymie Stata, and the participants at the September 1997 session of the IFIP WG 2.3 meeting in Alsace, France.

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.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[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.

[Cri84] Flaviu Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.

[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.

[Hoa72] C. A. R. Hoare. Notes on data structuring. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 83–174. Academic Press, 1972.

[Hoa96] C. A. R. Hoare. Unifying theories of programming. In *Marktoberdorf Summer School on Programming Methodology*, 1996.

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

[Lei97] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `http://www.cs.indiana.edu/hyplan/pierce/fool/`.

[LvdS94] K. Rustan M. Leino and Jan L. A. van de Snepscheut. Semantics of exceptions. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods, and Calculi*, pages 447–466. Elsevier, 1994.

[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.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.