Information Flow Analysis in Logical Form

Torben Amtoft and Anindya Banerjee*

Department of Computing and Information Sciences
Kansas State University, Manhattan KS 66506, USA
{tamtoft,ab}@cis.ksu.edu

Abstract. We specify an information flow analysis for a simple imperative language, using a Hoare-like logic. The logic facilitates static checking of a larger class of programs than can be checked by extant type-based approaches in which a program is deemed insecure when it contains an insecure subprogram. The logic is based on an abstract interpretation of program traces that makes independence between program variables explicit. Unlike other, more precise, approaches based on a Hoare-like logic, our approach does not require a theorem prover to generate invariants. We demonstrate the modularity of our approach by showing that a frame rule holds in our logic. Moreover, given an insecure but terminating program, we show how strongest postconditions can be employed to statically generate failure explanations.

1 Introduction

This paper specifies an information flow analysis using a Hoare-like logic and considers an application of the logic to explaining insecure flow of information in simple imperative programs.

Given a system with high, or secret (H), and low, or public (L) inputs and outputs, where $L \leq H$ is a security lattice, a classic security problem is how to enforce the following end-to-end *confidentiality* policy: protect secret data, i.e., prevent leaks of secrets at public output channels. An information flow analysis checks if a program satisfies the policy. Denning and Denning were the first to formulate an information flow analysis for confidentiality[11]. Subsequent advances have been comprehensively summarized in the recent survey by Sabelfeld and Myers [27]. An oft-used approach for specifying static analyses for information flow is *security type systems* [23, 29]. Security types are ordinary types of program variables and expressions annotated with security levels. Security typing rules prevent leaks of secret information to public channels. For example, the security typing rule for assignment prevents H data from being assigned to a L variable. A well-typed program "protects secrets", i.e., no information flows from H to L during program execution.

In the security literature, "protects secrets" is formalized as *noninterference* [13] and is described in terms of an "indistinguishability" relation on states.

* Supported by NSF grants CCR-0296182 and CCR-0209205

Two program states are indistinguishable for L if they agree on values of L variables. The noninterference property says that any two runs of a program starting from two initial states indistinguishable for L, yield two final states that are indistinguishable for L. The two initial states may differ on values of H variables but not on values of L variables; the two final states must agree on the current values of L variables. One reading of the noninterference property is as a form of (in)dependence [7]: L output is independent of H inputs. It is this notion that is made explicit in the information flow analysis specified in this paper.

A shortcoming of usual type-based approaches for information flow [4, 14, 29, 24] is that a type system can be too imprecise. Consider the sequential program $l := h; l := 0$, where l has type L and h has type H. This program is rejected by a security type system on account of the first assignment. But the program obviously satisfies noninterference – final states of any two runs of the program will always have the same value, 0, for l and are thus indistinguishable for L.

How can we admit such programs? Our inspiration comes from abstract interpretation [8], which can be viewed as a method for statically computing approximations of program invariants [9]. A benefit of this view is that the static abstraction of a program invariant can be used to annotate a program with pre- and postconditions and the annotated program can be checked against a Hoare-like logic. In information flow analysis, the invariant of interest is *independence of variables*, for which we use the notation $[x \# w]$ to denote that x is independent of w. The idea is that this holds provided any two runs (hereafter called *traces* and formalized in Section 2) which have the same initial[1] value for all variables *except for* w will at least agree on the current value of x. This is just a convenient restatement of noninterference but we tie it to the static notion of variable independence.

The set of program traces is potentially infinite, but our approach statically computes a finite abstraction, namely a set of independences, $T^{\#}$, that describes a set of traces, T. This is formalized in Section 3. We formulate (in Section 4) a Hoare-like logic for checking independences and show (Section 5) that a checked program satisfies noninterference. The assertion language of the logic is decidable since it is just the language of finite sets of independences with subset inclusion. Specifications in the logic have the form, $\{T^{\#}\}\ C\ \{T_1^{\#}\}$. Given precondition $T^{\#}$, we show in Section 6 how to compute strongest postconditions; for programs with loops, this necessitates a fixpoint computation[2]. We show that the logic deems the program $l := h; l := 0$ secure: the strongest postcondition of the program contains the independence $[l \# h]$.

Our approach falls in between type-based analysis and full verification where verification conditions for loops depend on loop invariants generated by a theorem prover. Instead, we approximate invariants using a fixpoint computation. Our approach is modular and we show that our logic satisfies a frame rule (Section 7). The frame rule permits local reasoning about a program: the relevant

[1] The initial value of a variable is its value before execution of the whole program.

[2] The set of independences is a finite lattice, hence the fixpoint computation will terminate.

independences for a program are only those $[x \# w]$ where x occurs in the program. Moreover, in a larger context, the frame rule allows the following inference (in analogy with [21]): start with a specification $\{T^{\#}\}\ C\ \{T_0^{\#}\}$ describing independences before and after store modifications; then, $\{T^{\#} \cup T_1^{\#}\}\ C\ \{T_0^{\#} \cup T_1^{\#}\}$ holds provided C does not modify any variable y, where $[y \# w]$ appears in $T_1^{\#}$. The initial specification, $\{T^{\#}\}\ C\ \{T_0^{\#}\}$ can reason with only the slice of store that C touches.

We also show (Section 9) that strongest postconditions can be used to statically generate failure explanations for an insecure but terminating program. If there is a program fragment C whose precondition contains $[l \# h]$, but whose strongest postcondition does not contain $[l \# h]$, we know statically that C is an offending fragment. Thus we may expect to find two initial values of h which produce two different values of l. We consider two ways this may happen [11]; we do not consider termination, timing leaks and other covert channels. One reason for failure of $[l \# h]$ to be in the strongest postcondition, is that C assigns H data to a L variable. The other reason is that C is a conditional or a while loop whose guard depends on a high variable and which updates a low variable in its body. Consider, for example, `if` h `then` $l := 1$ `else` $l := 0$. Our failure explanation for the conditional will be modulo an *interpretation function*, that, for distinct variables h_1 and h_2 map h_1 to *true* and h_2 to *false*. Under this interpretation, the execution of the program produces two different values of l. This explains why l is not independent of h. Because we use a static analysis, false positives may be generated: consider `if` h `then` $l := 7$ `else` $l := 7$, a program that is deemed insecure when it is clearly not. However, such false positives can be ruled out by an instrumented semantics that tracks constant values more precisely.

Contributions. First and foremost, we formulate information flow analysis in a logical form via a Hoare-like logic. The approach deems more programs secure than extant type-based approaches. Secondly, we describe the relationship between information flow and program dependence, explored in [1, 16], in a more direct manner by computing independences between program variables. The independences themselves are static descriptions of the noninterference property. In Section 8, we show how our logic conservatively extends the security type system of Smith and Volpano [29], by showing that any well-typed program in their system satisfies the invariant $[l \# h]$. Thirdly, when a program is deemed insecure, the annotated derivation facilitates explanations on *why* the program is insecure by statically generating counterexamples. The development in this paper considers *termination-insensitive* noninterference only: we assume that an attacker cannot observe nontermination. Complete proofs of all theorems appear in the companion technical report [2].

2 Language: syntax, traces, semantics

This section gives the syntax of a simple imperative language, formalizes the notion of traces, and gives the language a semantics using sets of traces.

Syntax. We consider a simple imperative language with assignment, sequencing, conditionals and loops as formalized by the following BNF. Commands $C \in$ **Cmd** are given by the syntax

$$C ::= x := E \mid C_1 \,; C_2 \mid \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while}\ E\ \texttt{do}\ C$$

where **Var** is an infinite set of variables, $x, y, z, w \in$ **Var** range over variables and where $E \in$ **Exp** ranges over expressions. Expressions are left unspecified but we shall assume the existence of a function fv(E) that computes the free variables of expression E. For commands, fv(C) is defined in the obvious way. We also define a function *modified* : **Cmd** $\to \mathcal{P}($**Var**$)$ that given a command, returns the set of variables potentially assigned to by the command.

Traces. A trace $t \in$ **Trc** associates each variable with its initial value and its current value; here values $v \in$ **Val** are yet unspecified but we assume that there exists a predicate *true?* on **Val**. (For instance, we could have **Val** as the set of integers and let *true?*(v) be defined as $v \neq 0$). We shall use $T \in \mathcal{P}($**Trc**$)$ to range over sets of traces. Basic operations on traces include:

- ini-$t(x)$ which returns the initial value of x as recorded by t;
- cur-$t(x)$ which returns the current value of x as recorded by t;
- $t[y \mapsto v]$ which returns a trace t' with the property: for all $x \in$ **Var**, ini-$t'(x) =$ ini-$t(x)$ and if $x \neq y$ then cur-$t'(x) =$ cur-$t(x)$; but cur-$t'(y) = v$.
- The predicate *initial* T on sets of traces T holds iff for all traces $t \in T$, and for all variables x, we have ini-$t(x) =$ cur-$t(x)$.

For instance, we could represent a trace t as a mapping **Var** $\to$ **Val** $\times$ **Val**; with $t(x) = (v_i, v_c)$ we would then have ini-$t(x) = v_i$ and cur-$t(x) = v_c$.

We shall write $t_1 \overset{x}{=} t_2$ to denote that cur-$t_1(x) =$ cur-$t_2(x)$, and we shall write $\neg(t_1 \overset{x}{=} t_2)$ to denote that $t_1 \overset{x}{=} t_2$ does not hold. Also, we shall write $t_1 \underset{x}{=} t_2$ to denote that for $y \neq x$, ini-$t_1(y) =$ ini-$t_2(y)$ holds. That is, the initial values of all variables, *except for* x, are equal in t_1 and t_2.

Semantics. We assume that there exists a semantic function $[\![E]\!]$: **Trc** $\to$ **Val** which satisfies the following property: if for all $x \in$ fv(E) we have $t_1 \overset{x}{=} t_2$, then $[\![E]\!](t_1) = [\![E]\!](t_2)$. The definition of $[\![E]\!]$ would contain the clause $[\![x]\!](t) =$ cur-$t(x)$. For each T and E we define

$$E\text{-true}(T) = \{t \in T \mid \mathit{true?}([\![E]\!](t))\}$$
$$E\text{-false}(T) = T \setminus E\text{-true}(T).$$

The semantics of a command has functionality $[\![C]\!] : \mathcal{P}($**Trc**$) \to \mathcal{P}($**Trc**$)$, and is defined in Fig. 1. To see that the last clause in Fig. 1 is well-defined, notice that $\mathcal{F}^C$ is a monotone function on the complete lattice $\mathcal{P}($**Trc**$) \to \mathcal{P}($**Trc**$)$.

$$
\begin{aligned}
[\![x := E]\!] &= \lambda T.\{t' \mid \exists t \in T : t' = t[x \mapsto [\![E]\!](t)]\} \\
[\![C_1 \,; C_2]\!] &= \lambda T.[\![C_2]\!]([\![C_1]\!](T)) \\
[\![\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2]\!] &= \lambda T.[\![C_1]\!](E\text{-true}(T)) \cup [\![C_2]\!](E\text{-false}(T)) \\
[\![\texttt{while } E \texttt{ do } C_0]\!] &= \mathit{lfp}(\mathcal{F}^C) \text{ where } C = \texttt{while } E \texttt{ do } C_0 \text{ and} \\
\mathcal{F}^C &: (\mathcal{P}(\mathbf{Trc}) \to \mathcal{P}(\mathbf{Trc})) \to (\mathcal{P}(\mathbf{Trc}) \to \mathcal{P}(\mathbf{Trc})) \\
\mathcal{F}^C(f) &= \lambda T.f([\![C_0]\!](E\text{-true}(T))) \cup E\text{-false}(T)
\end{aligned}
$$

Fig. 1. The Trace Semantics.

3 Independences

We are interested in a finite abstraction of a (possibly infinite) set of concrete traces. The abstract values are termed *independences*: an independence $T^\# \in \mathbf{Independ} = \mathcal{P}(\mathbf{Var} \times \mathbf{Var})$ is a set of pairs of the form $[x \# w]$, denoting that the *current* value of x is independent of the *initial* value of w. This is formalized by the following definition of when an independence correctly describes a set of traces. The intuition is that x is independent of w iff any two traces which have the same initial values except on w must agree on the current value of x; in other words, the initial value of w does not influence the current value of x at all.

Definition 1. *$[x \# w] \models T$ holds iff for all $t_1, t_2 \in T$: $t_1 \underset{w}{=} t_2$ implies $t_1 \overset{x}{=} t_2$.*
$T^\# \models T$ holds iff for all $[x \# w] \in T^\#$ it holds that $[x \# w] \models T$.

Definition 2. *The ordering $T_1^\# \preceq T_2^\#$ holds iff $T_2^\# \subseteq T_1^\#$.*

This is motivated by the desire for a subtyping rule, stating that if $T_1^\# \preceq T_2^\#$ then $T_1^\#$ can be replaced by $T_2^\#$. Such a rule is sound provided $T_2^\#$ is a subset of $T_1^\#$ and therefore obtainable from $T_1^\#$ by removing information. Clearly, **Independ** forms a complete lattice wrt. the ordering; let $\sqcap_i {T^\#}_i$ denote the greatest lower bound (which is the set union). We have some expected properties:

- If $T^\# \models T$ and $T_1 \subseteq T$ then $T^\# \models T_1$;
- if $T_1^\# \models T$ and $T_1^\# \preceq T_2^\#$ then $T_2^\# \models T$;
- if for all $i \in I$ it holds that $T_i^\# \models T$, then $\sqcap_{i \in I} T_i^\# \models T$.

Moreover, we can write a concretization function $\gamma : \mathbf{Independ} \to \mathcal{P}(\mathcal{P}(\mathbf{Trc}))$: $\gamma(T^\#) = \{T \mid T^\# \models T\}$. It is easy to verify that γ is completely multiplicative. Therefore [20, p.237] there exists a Galois connection between $\mathcal{P}(\mathcal{P}(\mathbf{Trc}))$ and **Independ**, with γ the concretization function. Finally, we have the following fact about initial sets of traces.

Fact 1 *For all T, if initial T then $[x \# y] \models T$ for all $x \neq y$.*

4 Static Checking of Independences

To statically check independences we define, in Fig. 2, a Hoare-like Logic where judgements are of the form $G \vdash \{T_1^\#\}\ C\ \{T_2^\#\}$. The judgement is interpreted as saying that if the independences in $T_1^\#$ hold *before* execution of C then, provided C terminates, the independences in $T_2^\#$ will hold *after* execution of C. The context $G \in \mathbf{Context} = \mathcal{P}(\mathbf{Var})$ is a *control dependence*, denoting (a superset of) the variables that at least one test surrounding C depends on. For example, in $\texttt{if}\ x\ \texttt{then}\ y := 0\ \texttt{else}\ z := 1$, the static checking of $y := 0$ takes place in the context that contains all variables that x is dependent on. This is crucial, especially since x may depend on a high variable.

We now explain a few of the rules in Fig. 2. Checking an assignment, $x := E$, in context G, involves checking any $[y \# w]$ in the postcondition $T^\#$. There are two cases. If $x \neq y$, then $[y \# w]$ must also appear in the precondition $T_0^\#$. Otherwise, if $x = y$ then $[x \# w]$ appears in the postcondition provided all variables referenced in E are independent of w; moreover, w must not appear in G, as otherwise, x would be (control) dependent on w.

Checking a conditional, $\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2$, involves checking C_1 and C_2 in a context G_0 that includes not only the "old" context G but also the variables that E depends on (as variables modified in C_1 or C_2 will be control dependent on such). Equivalently, if w is not in G_0, then all free variables x in E must be independent of w, that is, $[x \# w]$ must appear in the precondition $T_0^\#$.

Checking a while loop is similar to checking a conditional. The only difference is that it requires guessing an "invariant" $T^\#$ that is both the precondition and the postcondition of the loop and its body.

In Section 6, when we define *strongest postcondition*, we will select $G_0 = G \cup \{w \mid \exists x \in \mathrm{fv(E)} \bullet [x \# w] \notin T_0^\#\}$ for the conditional and the while loop. Instead of guessing the invariant, we will show how to compute it using fixpoints.

Example 1. We have the derivations

$$\emptyset \vdash \{\{[l \# h], [h \# l]\}\}\ l := h\ \{\{[h \# l], [l \# l]\}\} \text{ and}$$
$$\emptyset \vdash \{\{[h \# l], [l \# l]\}\}\ l := 0\ \{\{[h \# l], [l \# l], [l \# h]\}\}$$

and therefore also

$$\emptyset \vdash \{\{[l \# h], [h \# l]\}\}\ l := h\ ; l := 0\ \{\{[h \# l], [l \# l], [l \# h]\}\}$$

With the intuition that l stands for "low" or "public" and h stands for "high" or "sensitive", the derivation asserts that if l is independent of h before execution, then provided the program halts, l is independent of h after execution. By Definition 1, any two traces of the program with different initial values for h, agree on the current value for l. Thus the program is secure, although it contains an insecure sub-program.

Example 2. The reader may check that the following informally annotated program gives rise to a derivation in our logic. Initially, G is empty, and all variables

$$\text{[Assign]}\ G \vdash \{T_0^\#\}\ x := E\ \{T^\#\} \qquad \begin{array}{l}\text{if } \forall [y \# w] \in T^\# \bullet \\ \quad x \neq y \Rightarrow [y \# w] \in T_0^\# \\ \quad x = y \Rightarrow w \notin G \wedge \forall z \in \mathrm{fv(E)} \bullet [z \# w] \in T_0^\#\end{array}$$

$$\text{[Seq]}\quad \frac{G \vdash \{T_0^\#\}\ C_1\ \{T_1^\#\} \qquad G \vdash \{T_1^\#\}\ C_2\ \{T_2^\#\}}{G \vdash \{T_0^\#\}\ C_1\ ; C_2\ \{T_2^\#\}}$$

$$\text{[If]}\quad \frac{G_0 \vdash \{T_0^\#\}\ C_1\ \{T^\#\} \qquad G_0 \vdash \{T_0^\#\}\ C_2\ \{T^\#\}}{G \vdash \{T_0^\#\}\ \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{T^\#\}} \quad \begin{array}{l}\text{if} \quad G \subseteq G_0 \\ \text{and } w \notin G_0 \Rightarrow \forall x \in \mathrm{fv(E)} \bullet [x \# w] \in T_0^\#\end{array}$$

$$\text{[While]}\quad \frac{G_0 \vdash \{T^\#\}\ C\ \{T^\#\}}{G \vdash \{T^\#\}\ \texttt{while}\ E\ \texttt{do}\ C\ \{T^\#\}} \quad \begin{array}{l}\text{if} \quad G \subseteq G_0 \\ \text{and } w \notin G_0 \Rightarrow \forall x \in \mathrm{fv(E)} \bullet [x \# w] \in T^\#\end{array}$$

$$\text{[Sub]}\quad \frac{G_1 \vdash \{T_1^\#\}\ C\ \{T_2^\#\}}{G_0 \vdash \{T_0^\#\}\ C\ \{T_3^\#\}} \quad \text{if } T_0^\# \preceq T_1^\# \text{ and } T_2^\# \preceq T_3^\# \text{ and } G_0 \subseteq G_1$$

Fig. 2. The Hoare Logic.

are pairwise independent; we write $[x \# y, z]$ to abbreviate $[x \# y], [x \# z]$.

	$\{[l \# h, x], [h \# l, x], [x \# l, h]\}$
$x := h$	$\{[l \# h, x], [h \# l, x], [x \# l, x]\}$
`if` $x > 0$	(G is now $\{h\}$)
`then` $l := 7$	$\{[l \# x, l], [h \# l, x], [x \# l, x]\}$
`else` $x := 0$	$\{[l \# h, x], [h \# l, x], [x \# l, x]\}$
end of `if`	$\{[l \# x], [h \# l, x], [x \# l, x]\}$

A few remarks:

- in the preamble, only x is assigned, so the independences for l and h are carried through, but $[x \# l, x]$ holds afterwards, as $[h \# l, x]$ holds beforehand;
- the free variable in the guard is independent of l and x but not of h, implying that h has to be in G.

5 Correctness

We are now in a position to prove the correctness of the Hoare logic with respect to the trace semantics.

Theorem 2. *Assume that*

$$G \vdash \{T_0^\#\}\ C\ \{T^\#\} \text{ where for all } [x \# y] \in T_0^\#, \text{ it is the case that } x \neq y.$$

Then, initial T implies $T^\# \models [\![C]\!](T)$.

That is, if T is an *initial set*, then $T^\#$ correctly describes the set of concrete traces obtained by executing command C on T.

The correctness theorem can be seen as the noninterference theorem for information flow. Indeed, with l and h interpreted as "low" and "high" respectively, suppose $[l \# h]$ appears in $T^\#$. Then any two traces in $[\![C]\!](T)$ (the set of traces resulting from the execution of command C from initial set T) that have initial values that differ only on h, must agree on the current value of l.

Note that the correctness result deals with "terminating" traces only. For example, with $P = \texttt{while}\ h \neq 0\ \texttt{do}\ h := 7$ and $T^\# = \{[l \# h], [h \# l]\}$ we have the judgement $\emptyset \vdash \{T^\#\}\ P\ \{T^\#\}$ (since $\{h\} \vdash \{T^\#\}\ h := 7\ \{T^\#\}$) showing that P is deemed secure by our logic, yet an observer able to observe non-termination can detect whether h was initially 0 or not.

To prove Theorem 2, we claim the following, more general, lemma. Then the theorem follows by the lemma using Fact 1.

Lemma 1. *If* $G \vdash \{T_0^\#\}\ C\ \{T^\#\}$ *and* $T_0^\# \models T$ *then also* $T^\# \models [\![C]\!](T)$.

6 Computing Independences

In Fig. 3 we define a function

$$sp : \mathbf{Context} \times \mathbf{Cmd} \times \mathbf{Independ} \to \mathbf{Independ}$$

with the intuition (formalized below) that given a control dependence G, a command C and a precondition $T^\#$, $sp(G, C, T^\#)$ computes a postcondition $T_1^\#$ such that $G \vdash \{T^\#\}\ C\ \{T_1^\#\}$ holds, and $T_1^\#$ is the "largest" set (wrt. the subset ordering) that makes the judgement hold. Thus we compute the "strongest provable postcondition", which might differ[3] from the strongest *semantic* postcondition, that is, the largest set $T_1^\#$ such that for all T, if $T^\# \models T$ then $T_1^\# \models [\![C]\!](T)$.

In the companion technical report [2], we show how to also compute "weakest precondition"; we conjecture that the developments in Sections 7 and 9 could also be carried out using weakest precondition instead of strongest postcondition.

We now explain two of the cases in Fig. 3. In an assignment, $x := E$, the postcondition carries over all independences $[y \# w]$ in the precondition if $y \neq x$; these independences are unaffected by the assignment to x. Suppose that w does not occur in context G. Then x is not control dependent on w. Moreover, if all variables referenced in E are independent of w, then $[x \# w]$ will be in the postcondition of the assignment.

The case for `while` is best explained by means of an example.

Example 3. Consider the program

$$C = \texttt{while}\ y\ \texttt{do}\ l := x\ ; x := y\ ; y := h.$$

[3] For example, let $C = l := h - h$ and $T^\# = \{[l \# h]\}$. Then $[l \# h]$ is in the strongest semantic postcondition, since for all T and all $t \in [\![C]\!](T)$ we have cur-$t(l) = 0$ and therefore $[l \# h] \models [\![C]\!]T$, but not in the strongest provable postcondition.

Let $T_0^\# \ldots T_8^\#$ be given by the following table. For example, the entry in the column for $T_4^\#$ and in the row for x shows that $[x \# h] \in T_4^\#$ and $[x \# l] \in T_4^\#$.

	$T_0^\#$	$T_1^\#$	$T_2^\#$	$T_3^\#$	$T_4^\#$	$T_5^\#$	$T_6^\#$	$T_7^\#$	$T_8^\#$
$h \#$	$\{l,x,y\}$	$\{l,x,y\}$	$\{l,x,y\}$	$\{l,x,y\}$	$\{l,x,y\}$	$\{l,x,y\}$	$\{l,x,y\}$	$\{l,x,y\}$	$\{l,x,y\}$
$l \#$	$\{h,x,y\}$	$\{h,l\}$	$\{h,l\}$	$\{h,l\}$	$\{h\}$	$\{l\}$	$\{l\}$	$\emptyset$	$\{l\}$
$x \#$	$\{h,l,y\}$	$\{h,l,y\}$	$\{h,l,x\}$	$\{h,l,x\}$	$\{h,l\}$	$\{h,l\}$	$\{l,x\}$	$\{l\}$	$\{l\}$
$y \#$	$\{h,l,x\}$	$\{h,l,x\}$	$\{h,l,x\}$	$\{l,x\}$	$\{l,x\}$	$\{l,x\}$	$\{l,x\}$	$\{l,x\}$	$\{l,x\}$

Our goal is to compute $sp(\emptyset, C, T_0^\#)$ and doing so involves the fixed point computation sketched below.

	Iteration		
	first	second	third
`while` y `do`	$T_0^\#$	$T_4^\# = T_3^\# \cap T_0^\#$	$T_7^\# = T_6^\# \cap T_0^\#$
G_0 :	$\{y\}$	$\{h,y\}$	$\{h,y\}$
$l := x$	$T_1^\#$	$T_5^\#$	$T_8^\#$
$x := y$	$T_2^\#$	$T_6^\#$	$T_6^\#$
$y := h$	$T_3^\#$	$T_6^\#$	$T_6^\#$

For example, the entry $T_6^\#$ in the column marked "second" and in the second row from the bottom, denotes that $sp(\{h,y\}, x := y, T_5^\#) = T_6^\#$.

Note that after the first iteration, $[l \# h]$ is still present; it takes a second iteration to filter it out and thus detect insecurity. The third iteration affirms that $T_7^\#$ is indeed a fixed point (of the functional $\mathcal{H}_C^{T_0^\#,\emptyset}$ defined in Fig. 3).

Theorem 3 states the correctness of the function *sp*, that it indeed computes a postcondition. Then, Theorem 4 states that the postcondition computed by *sp* is the strongest postcondition. We shall rely on the following property:

Lemma 2 (Monotonicity). *For all C, the following holds (for all G,G_1,$T^\#$,$T_1^\#$):*

1. *if $G \subseteq G_1$ then $sp(G, C, T^\#) \preceq sp(G_1, C, T^\#)$;*
2. *if $T^\# \preceq T_1^\#$ then $sp(G, C, T^\#) \preceq sp(G, C, T_1^\#)$.*

Theorem 3. *For all C, G, $T^\#$, it holds that $G \vdash \{T^\#\}\ C\ \{sp(G, C, T^\#)\}$.*

Theorem 4. *For all judgements $G \vdash \{T_1^\#\}\ C\ \{T^\#\}$, $sp(G, C, T_1^\#) \preceq T^\#$.*

The following result is useful for the developments in Sections 7 and 9:

Lemma 3. *Given y, C with $y \notin modified(C)$. Then for all $T^\#$, G, w: $[y \# w] \in T^\#$ implies $[y \# w] \in sp(G, C, T^\#)$.*

$$sp(G, x := E, T^{\#}) =$$
$$\quad \{[y \# w] \mid y \neq x \wedge [y \# w] \in T^{\#}\} \cup \{[x \# w] \mid w \notin G \wedge \forall y \in \mathrm{fv(E)} \bullet [y \# w] \in T^{\#}\}$$

$$sp(G, C_1 ; C_2, T^{\#}) = sp(G, C_2, sp(G, C_1, T^{\#}))$$

$$sp(G, \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2, T^{\#}) =$$
$$\quad \text{let } G_0 = G \cup \{w \mid \exists x \in \mathrm{fv(E)} \bullet [x \# w] \notin T^{\#}\}$$
$$\qquad T_1^{\#} = sp(G_0, C_1, T^{\#})$$
$$\qquad T_2^{\#} = sp(G_0, C_2, T^{\#})$$
$$\quad \text{in } T_1^{\#} \cap T_2^{\#}$$

$$sp(G, \texttt{while } E \texttt{ do } C_0, T^{\#}) =$$
$$\quad \text{let } \mathcal{H}_C^{T^{\#},G} : \textbf{Independ} \to \textbf{Independ} \text{ be given by } (C = \texttt{while } E \texttt{ do } C_0)$$
$$\qquad \mathcal{H}_C^{T^{\#},G}(T_0^{\#}) =$$
$$\qquad\quad \text{let } G_0 = G \cup \{w \mid \exists x \in \mathrm{fv(E)} \bullet [x \# w] \notin T_0^{\#}\}$$
$$\qquad\quad \text{in } sp(G_0, C_0, T_0^{\#}) \cap T^{\#}$$
$$\quad \text{in } lfp(\mathcal{H}_C^{T^{\#},G})$$

Fig. 3. Strongest Postcondition.

7 Modularity and the Frame Rule

Define $lhs(T^{\#}) = \{y \mid [y \# w] \in T^{\#}\}$. Then we have

Theorem 5 (Frame rule (I)). *Let $T_0^{\#}$ and C be given. Then for all $T^{\#}$, G:*

1. *If $lhs(T_0^{\#}) \cap modified(C) = \emptyset$ then $sp(G, C, T^{\#} \cup T_0^{\#}) \supseteq sp(G, C, T^{\#}) \cup T_0^{\#}$.*
2. *If $lhs(T_0^{\#}) \cap \mathrm{fv(C)} = \emptyset$ then $sp(G, C, T^{\#} \cup T_0^{\#}) = sp(G, C, T^{\#}) \cup T_0^{\#}$.*

Note that the weaker premise in 1 does not imply the stronger consequence in 2, since (with $[z \# w]$ playing the role of $T_0^{\#}$)

$$sp(\emptyset, x := y + z, \{[y \# w]\} \cup \{[z \# w]\}) = \{[y \# w], [z \# w], [x \# w]\}$$
$$sp(\emptyset, x := y + z, \{[y \# w]\}) \cup \{[z \# w]\} = \{[y \# w], [z \# w]\}.$$

In separation logic [17, 25], the frame rule is motivated by the desire for local reasoning: if C_1 and C_2 modify disjoint regions of a heap, reasoning about C_1 can be performed independently of the reasoning about C_2. In our setting, a consequence of the frame rule is that when analyzing a command C occurring in a larger context, the relevant independences are the ones whose left hand sides occur in C.

Theorem 5 is proved by observing that part (1) follows from Lemmas 3 and 2; then part (2) follows using the following result:

Lemma 4. *Let $T_0^{\#}$ and C be given, with $lhs(T_0^{\#}) \cap \mathrm{fv(C)} = \emptyset$. Then for all $T^{\#}$ and G, $sp(G, C, T^{\#} \cup T_0^{\#}) \subseteq sp(G, C, T^{\#}) \cup T_0^{\#}$.*

As a consequence of Theorem 5 we get the following result:

Corollary 1 (Frame rule (II)). *Assume that* $G \vdash \{T_1^\#\}\ C\ \{T_2^\#\}$ *and that* $lhs(T_0^\#) \cap modified(C) = \emptyset$. *Then* $G \vdash \{T_1^\# \cup T_0^\#\}\ C\ \{T_2^\# \cup T_0^\#\}$.

Proof. Using Theorems 5 and 4 we get $sp(G, C, T_1^\# \cup T_0^\#) \supseteq sp(G, C, T_1^\#) \cup T_0^\# \supseteq T_2^\# \cup T_0^\#$. Since by Theorem 3 we have $G \vdash \{T_1^\# \cup T_0^\#\}\ C\ \{sp(G, C, T_1^\# \cup T_0^\#)\}$, the result follows by [Sub].

A traditional view of modularity in the security literature is the "hook-up property" [19]: if two programs are secure then their composition is secure as well. Our logic satisfies the hook-up property for sequential composition; in our context, a secure program is one which has $[l \# h]$ as an invariant (if $[l \# h]$ is in the precondition, it is also in the strongest postcondition). With this interpretation, Sabelfeld and Sands's hook-up theorem holds [28, Theorem 5].

8 The Smith-Volpano Security Type System

In the Smith-Volpano type system [29], variables are labelled by security types; for example, $x : (T, \kappa)$ means that x has type T and security level κ. To handle implicit flows due to conditionals, the technical development requires commands to be typed $(\mathbf{com}\,\kappa)$ with the intention that all variables assigned to in such commands have level at least κ. The judgement $\Gamma \vdash C : (\mathbf{com}\ \kappa)$ says that in the security type context Γ, that binds free variables in C to security types, command C has type $(\mathbf{com}\ \kappa)$.

We now show a conservative extension: if a command is well-typed in the Smith-Volpano system, then for any two traces, the current values of low variables are independent of the initial values of high variables. For simplicity, we consider a command with only two variables, h with level H and l with level L.

Theorem 6. *Assume that* C *can be given a security type wrt. environment* $h : (\_, H), l : (\_, L)$. *Then for all* $T^\#$, *if* $[l \# h] \in T^\#$ *then* $[l \# h] \in sp(\emptyset, C, T^\#)$.

The upshot of the theorem is that a well-typed program has $[l \# h]$ as *invariant*: if $[l \# h]$ appears in the precondition, then it also appears in the strongest postcondition.

9 Counter-example Generation

Assume that a program C cannot be deemed secure by our logic, that is, $[l \# h] \notin sp(\emptyset, C, T^\#)$ (where $T^\# \supseteq \{[l \# h]\}$). Then we might expect that we can find a "witness": two different initial values of h that produce two different final values of l. However, below we shall see three examples of false positives: programs which, while deemed insecure by our logic, do not immediately satisfy that property. Ideally, we would like to strengthen our analysis so as to rule out such false positives; this does not seem immediately feasible and instead, in order to arrive at a suitable result, we shall modify our semantics so the false positives become genuine positives. The programs in question are:

$$l := h - h. \tag{1}$$

$$\texttt{if } h \texttt{ then } l := 7 \texttt{ else } l := 7 \tag{2}$$

$$\texttt{while } h \texttt{ do } l := 7 \tag{3}$$

To deal with (1), a program where writing a high expression to a low variable does not reveal anything about the high variable, we shall assume that expressions are unevaluated (kept as symbolic trees); formally we demand that if there exists $z \in \mathrm{fv(E)}$ with $\neg(t_1 \stackrel{z}{=} t_2)$, then $[\![E]\!](t_1) \neq [\![E]\!](t_2)$.

To deal with (2), a program where writing to a low variable under high guard does not immediately enable an observer to determine the value of the high variable, we *tag* each assignment statement so that an observer can detect which branch is taken.

Finally, we must deal with (3), a program where there cannot be two different final values of l. There seems to be no simple way to fix this, except to *rule out loops*, thus in effect considering only programs with a fixed bound on run-time (since for such, a loop can be unfolded repeatedly and eventually replaced by a sequence of conditionals; this is how we handle loops with low guard). Remember (cf. Section 5) that a program deemed *secure* by our logic may not be really secure if non-termination can be observed; similarly a program deemed *insecure* may not be really insecure if non-termination cannot be observed.

Even with the above modifications, the existence of a witness is not amenable to a compositional proof. For consider the program $x := E_1(h)\,; l := E_2(x)$ where E_1 and E_2 are some expressions. Inductively, on the assignment to l, we can find two different values for x, v_1 and v_2, such that the resulting values of l are different. But we then need an extremely strong property concerning the assignment to x: that there exists two different values of h such that evaluating $E_1(h)$ wrt. these values produces v_1, respectively v_2.

Instead, we shall settle for a result which says that *all* pairs of different initial values for h are witnesses, in that the resulting values of l are different. Of course, we need to introduce some extra assumptions to establish this stronger property. For example, consider the program $\texttt{if } h = 0 \texttt{ then } l := 17 \texttt{ else } l := 7$ where two different values of h, say 3 and 4, may cause the same branch to be taken. To deal with that, our result must say that for every two values of h there exists an interpretation of *true?* such that wrt. that interpretation, different values of l result. In the above, we might stipulate that *true?*$(3 = 0)$ but not *true?*$(4 = 0)$. It turns out to be convenient to let that interpretation depend on the guard in question; hence we shall also tag guards so as to distinguish between different occurrences of the same guard.

We thus end up with a semantics $[\![C]\!]_{\mathcal{I}}$ parametrized wrt. an interpretation $\mathcal{I}$; the full development is in [2] where the following result is proved:

Theorem 7. *Assume that* $sp(\emptyset, C, T^{\#}) = T_1^{\#}$, *with* $[x \,\#\, h] \in T^{\#}$ *for* $x \neq h$ *and with* $[l \,\#\, h] \notin T_1^{\#}$. *Further assume that* $\neg(t_1 \stackrel{h}{=} t_2)$, *with the tags of* t_1 *and* t_2 *being disjoint from the tags in* C.

Then there exists an interpretation $\mathcal{I}$ *such that* $\neg([\![C]\!]_{\mathcal{I}}(t_1) \stackrel{l}{=} [\![C]\!]_{\mathcal{I}}(t_2))$.

10 Discussion

Perspective. This paper specifies an information flow analysis for confidentiality using a Hoare-like logic and considers an application of the logic to explaining *insecurity* in simple imperative programs. Program traces, potentially infinitely many, are abstracted by finite sets of variable independences. These variable independences can be statically computed using strongest postconditions, and can be statically checked against the logic.

Giacobazzi and Mastroeni [12] consider attackers as abstract interpretations and generalize the notion of noninterference by parameterizing it wrt. what an attacker can analyze about the input/output information flow. For instance, assume an attacker can only analyze the *parity* (odd/even) of values. Then

$$\texttt{while}\ h\ \texttt{do}\ l := l + 2\,; h := h - 1$$

is secure, although it contains an update of a low variable under a high guard. We might try to model this approach in our framework by parameterizing Definition 1 wrt. parity, but it is not clear how to alter the proof rules accordingly. Instead, we envision our logic to be put on top of abstract interpretations. In the above example, the program would be abstracted to $\texttt{while}\ h\ \texttt{do}\ h := h - 1$ which our logic already deems secure.

Related work. Perhaps the most closely related work is the one of Clark, Hankin, and Hunt [6], who consider a language similar to ours and then extend it to Idealized Algol, requiring distinguishing between identifiers and locations. The analysis for Idealized Algol is split in two stages: the first stage does a control-flow analysis, specified using a flow logic [20]. The second stage specifies what is an acceptable information flow analysis with respect to the control-flow analysis. The precision of the control-flow analysis influences the precision of the information flow analysis. Flow logics usually do not come with a frame rule so it is unclear what modularity properties their analysis satisfies. For each statement S in the program, they compute the set of dependences introduced by S; a pair (x, y) is in that set if different values for y prior to execution of S may result in different values for x after execution of S. For a complete program, they thus, as expected, compute essentially the same information as we do, but the information computed *locally* is different from ours: we estimate if different *initial* values of y, i.e., values of y prior to execution of *the whole program*, may result in different values for x after execution of S. Unlike our approach, their analysis is termination-sensitive.

To make our logic termination-sentitive, we could (analogous in spirit to [6]) define $[\bot\ \#\, w]$ to mean that if two tuples of initial values are equal except for on w, then either both tuples give rise to terminating computations, or both tuples give rise to infinite computations. For instance, if

$$\vdash \{T_0^\#\}\ \texttt{while}\ x > 7\ \texttt{do}\ x := x + 1\ \{T^\#\}$$

and $[x \,\#\, h]$ does not belong to $T_0^\#$ then $[\bot\ \#\, h]$ should not belong to $T^\#$ (neither of any subsequent assertion), since different values of h may result in different

values of x and hence of different termination properties. To prove semantic correctness for the revised logic we would need to also revise our semantics, since currently it does not facilitate reasoning about infinite computations.

Joshi and Leino [18] provide an elegant semantic characterization of noninterference that allows handling both termination-sensitive and termination-insensitive noninterference. Their notion of security for a command C is equationally characterized by $C\,;HH = HH\,;C\,;HH$, where HH means that an arbitrary value is assigned to a high variable. They show how to express their notion of security in Dijkstra's weakest precondition calculus. Although they do not consider synthesizing loop invariants, this can certainly be done via a fixpoint computation with weakest preconditions. However, their work is not concerned with computing dependences, nor do they consider generating counterexamples.

Darvas, Hähnle and Sands [10] use dynamic logic to express secure information flow in JavaCard. They discuss several ways that noninterference can be expressed in a program logic, one of which is as follows: consider a program with variables l and h. Consider another copy of the program with l, h relabeled to fresh variables l', h' respectively. Then, noninterference holds in the following situation: running the original program and the copy sequentially such that the initial state satisfies $l = l'$ should yield a final state satisfying $l = l'$. Like us, they are interested in showing insecurity by exhibiting distinct initial values for high variables that give distinct current values of low variables; unlike us, they look at actual runtime values. To achieve this accuracy, they need the power of a general purpose theorem prover, which is also helpful in that they can express declassification, as well as treat exceptions (which most approaches based on static analysis cannot easily be extended to deal with).

Barthe, D'Argenio and Rezk [5] use the same idea of self-composition (i.e., composing a program with a copy of itself) as Darvas et alii and investigate "abstract" noninterference [12] for several languages. By parameterizing noninterference with a property, they are able to handle more general information flow policies, including a form of declassification known as delimited information release [26]. They show how self-composition can be formulated in logics describing these languages, namely, Hoare logic, separation logic, linear temporal logic, etc. They also discuss how to use their results for model checking programs with finite state spaces to check satisfaction of their generalized definition of noninterference.

The first work that used a Hoare-style semantics to reason about information flow was by Andrews and Reitman [3]. Their assertions keep track of the security level of variables, and are able to deal even with parallel programs. However, no formal correctness result is stated.

Conclusion. This paper was inspired in part by presentations by Roberto Giacobazzi and Reiner Hähnle at the Dagstuhl Seminar on Language-based Security in October 2003. The reported work is only the first step in our goal to formulate more general definitions of noninterference in terms of program (in)dependence, such that the definitions support modular reasoning. One direction to consider is to repeat the work in this paper for a richer language, with methods, pointers,

objects and dynamic memory allocation; an obvious goal here is interprocedural reasoning about variable independences perhaps using a higher-order version of the frame rule [22]. Hähnle's Dagstuhl presentation inspired us to look at explaining insecurity by showing counterexamples. We plan to experiment with model checkers supporting linear arithmetic, for example BLAST [15], to *(i)* establish independences that our logic cannot find (cf. the false positives from Sect. 9); *(ii)* provide "genuine" counterexamples that are counterexamples wrt. the original semantics.

Acknowledgements. We would like to thank Reiner Hähnle, Peter O'Hearn, Tamara Rezk, David Sands, and Hongseok Yang, as well as the participants of the *Open Software Quality* meeting in Santa Cruz, May 2004, and the anonymous reviewers, for useful comments on a draft of this paper.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 147–160, 1999.
2. Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. Technical Report CIS TR 2004-3, Kansas State University, April 2004.
3. G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–75, January 1980.
4. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–270. IEEE Computer Society Press, 2002.
5. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2004. To appear.
6. David Clark, Chris Hankin, and Sebastian Hunt. Information flow for Algol-like languages. *Computer Languages*, 28(1):3–28, 2002.
7. Ellis S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, New York, NY, 1977.
9. Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices*, volume 12, pages 1–12. ACM Press, August 1977.
10. Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. Technical Report 2004-01, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004. A fuller version of a paper appearing in Workshop on Issues in the Theory of Security, 2003.

11. Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
12. Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 186–197, 2004.
13. J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.
14. Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–377, 1998.
15. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.
16. Sebastian Hunt and David Sands. Binding time analysis: A new PERspective. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*, volume 26 (9) of *Sigplan Notices*, pages 154–165, 1991.
17. Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–26, 2001.
18. Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37:113–138, 2000.
19. Daryl McCullough. Specifications for multi-level security and a hook-up. In *IEEE Symposium on Security and Privacy, April 27-29, 1987*, pages 161–166, 1987.
20. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. Web page at `www.imm.dtu.dk/~riis/PPA/ppa.html`.
21. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
22. Peter O'Hearn, Hongseok Yang, and John Reynolds. Separation and information hiding. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 268–280, 2004.
23. Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.
24. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
25. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society Press, 2002.
26. Andrei Sabelfeld and Andrew Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security (ISSS'03)*, 2004. To appear.
27. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
28. Andrei Sabelfeld and David Sands. A Per model of secure information flow in sequential programs. *Higher-order and Symbolic Computation*, 14(1):59–91, 2001.
29. Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, number 1214 in Lecture Notes in Computer Science, pages 607–621. Springer-Verlag, 1997.