A New Type System for Secure Information Flow

Geoffrey Smith
School of Computer Science
Florida International University
Miami, Florida 33199, USA
smithg@cs.fiu.edu

Abstract

With the variables of a program classified as L (low, public) or H (high, private), we wish to prevent the program from leaking information about H variables into L variables. Given a multi-threaded imperative language with probabilistic scheduling, the goal can be formalized as a property called probabilistic noninterference. Previous work identified a type system sufficient to guarantee probabilistic noninterference, but at the cost of severe restrictions: to prevent timing leaks, H variables were disallowed from the guards of **while** *loops. Here we present a new type system that gives each command a type of the form τ_1 cmd τ_2; this type says that the command assigns only to variables of level τ_1 (or higher) and has running time that depends only on variables of level τ_2 (or lower). Also we use types of the form τ cmd n for commands that terminate in exactly n steps. With these typings, we can prevent timing leaks by demanding that no assignment to an L variable may sequentially follow a command whose running time depends on H variables. As a result, we can use H variables more flexibly; for example, under the new system a thread that involves only H variables is always well typed. The soundness of the type system is proved using the notion of probabilistic bisimulation.*

1 Introduction

In this paper, as in [10] and [13], we consider a simple multi-threaded imperative programming language in which each variable is classified either as L (low, public) or H (high, private). Our goal is to develop a static analysis that ensures that a program cannot "leak" the values of H variables. Of course, the possible ways of leaking information depend on what is observable. If we can observe the running program from the *outside*, seeing running time or the usage of various system resources, then controlling leaks is very difficult, because leaks can be based on very low-level implementation details, such as caching behavior. Hence our focus, as in previous work, is on controlling *internal* leaks, in which information about H variables is somehow transmitted to L variables. This makes the task more tractable, because we can control what is observable by the running program—for example, we can deny it access to a real-time clock.

More precisely, we wish to achieve *noninterference* properties, which assert that changing the initial values of H variables cannot affect the final values of L variables. Given the nondeterminism associated with multi-threading, and our assumption that thread scheduling is probabilistic, we require more precisely that changing the initial values of H variables cannot affect the joint probability distribution of the possible final values of L variables; this property is called *probabilistic noninterference*.

A type system that guarantees a weaker property, called *possibilistic noninterference*, was given in [10]. Building on that work, a type system for probabilistic noninterference was given in [13]. The restrictions imposed by that system can be summarized as follows:

1. An expression e is H if it contains any H variables; otherwise it is L.
2. Only L expressions can be assigned to L variables.
3. A guarded command with H guard cannot assign to L variables.
4. The guard of a **while** loop must be L.
5. An **if** with H guard must be *protected*, so that it executes atomically, and can contain no **while** loops within its branches.

Restrictions 2 and 3 prevent what Denning [3] long ago called *direct* and *indirect* flows, respectively. In a language without concurrency, restrictions 2 and 3 are sufficient to guarantee noninterference—see for example [15]. Restrictions 4 and 5 were introduced to prevent timing-based flows

in multi-threaded programs; unfortunately, they restrict the set of allowable programs quite severely.

A recent paper by Honda, Vasconcelos, and Yoshida [6] explores secure information flow in the π-calculus, showing in particular that the system of [10] can be embedded into their system. Most interestingly, they propose *enriching* the set of command types of [10] from

- H *cmd*, for commands that assign only to H variables and are guaranteed to terminate; and
- L *cmd*, for commands that assign to L variables or might not terminate,

to

- τ *cmd* $\Downarrow$, for commands that assign only to variables of type τ (or higher) and are guaranteed to terminate; and
- τ *cmd* $\Uparrow$, for commands that assign only to variables of type τ (or higher) and might not terminate.

They then argue that in some cases H variables can be used in the guards of **while** loops without sacrificing possibilistic noninterference.

Inspired by this suggestion, we can observe that the command typings used in [10] and [13] conflate two distinct issues: what does a command *assign to*, and what is the command's *running time*. This leads us to propose command types with *two* parameters to address these two issues separately. More precisely, our new system will make use of the following command types:

- τ_1 *cmd* τ_2, for commands that assign only to variables of type τ_1 (or higher) and whose running time depends only on variables of type τ_2 (or lower); and
- τ *cmd* n, for commands that assign only to variables of type τ (or higher) and which are guaranteed to terminate in exactly n steps.

With these typings, we can impose more accurate restrictions to prevent flows based on timing. In particular, we can replace restrictions 4 and 5 above with the following rule:

> A command whose running time depends on H variables cannot be followed sequentially by a command that assigns to L variables.

Let's now consider some examples informally, assuming that $x : H$ and $y : L$.

1. $x := 0$: H *cmd* 1
2. $y := 0$: L *cmd* 1
3. **if** $x = 0$ **then** $x := 5$ **else skip** : H *cmd* 2
4. **while** $y = 0$ **do skip** : H *cmd* L
5. **while** $y = 0$ **do** $y := y - 1$: L *cmd* L
6. **if** $x = 0$ **then**
   while $y = 0$ **do skip**
   else
   skip : H *cmd* H
7. $y := 5$; **while** $x + 1$ **do skip** : L *cmd* H
8. (**while** $x = 0$ **do skip**); $y := 5$: *illegal*

Example 3 shows that an **if** can have a H guard and nevertheless have a known running time; such a command can be sequentially followed by a L assignment without any problem.

Example 8, on the other hand, is illegal because the running time of the **while** loop depends on the H variable x, so we can't follow it sequentially with an assignment to the L variable y. This example would be dangerous, because another thread could reliably determine whether x is 0 or not, simply by waiting for a while (to give the thread scheduler a chance to run all threads) and then seeing whether y is 5.

Our typings also satisfy interesting subtyping rules. As usual, we have $L \subseteq H$. Furthermore, command types are contravariant in their first position, and covariant in their second position. Also, τ *cmd* $n \subseteq \tau$ *cmd* L, because if a command always halts in n steps, then its running time doesn't depend on the values of H variables. This subtyping rule implies that example 3 above also has type H *cmd* L.

The rest of the paper is organized as follows. Section 2 reviews the definition of our multi-threaded language, which is the same as the language of [13], and Section 3 defines our type system precisely. The soundness of the type system is then proved in Section 4, which argues that every well-typed multi-threaded program satisfies probabilistic noninterference. Finally, Section 5 concludes and mentions some future directions.

2 The Multi-Threaded Language

Threads are written in the simple imperative language:

(*phrases*)	p	::=	$e \mid c$
(*expressions*)	e	::=	$x \mid n \mid e_1 + e_2 \mid$ $e_1 * e_2 \mid e_1 = e_2 \mid \ldots$
(*commands*)	c	::=	$x := e \mid$ **skip** $\mid$ **if** e **then** c_1 **else** $c_2 \mid$ **while** e **do** $c \mid$ $c_1; c_2 \mid$ **protect** c

In our syntax, metavariable x ranges over identifiers and n over integer literals. Integers are the only values; we use 0 for false and nonzero for true. We assume that expressions are *free of side effects* and are *total*. The command **protect** c causes c to be executed *atomically*; this is important only when concurrency is considered.

Programs are executed with respect to a memory μ, which is a mapping from identifiers to integers. Also, we assume for simplicity that expressions are evaluated atomically; thus we simply extend a memory μ in the obvious way to map expressions to integers, writing $\mu(e)$ to denote the value of expression e in memory μ.

We define the semantics of commands via a sequential transition relation $\longrightarrow$ on configurations. A *configuration* C is either a pair (c, μ) or simply a memory μ. In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ. The sequential transition relation is defined by the following (completely standard) structural operational semantics:

(UPDATE) $$\frac{x \in \mathit{dom}(\mu)}{(x := e, \mu) \longrightarrow \mu[x := \mu(e)]}$$

(NO-OP) $$(\textbf{skip}, \mu) \longrightarrow \mu$$

(BRANCH) $$\frac{\mu(e) \neq 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_2, \mu)}$$

(LOOP) $$\frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu) \longrightarrow \mu}$$

$$\frac{\mu(e) \neq 0}{(\textbf{while } e \textbf{ do } c, \mu) \longrightarrow (c; \textbf{while } e \textbf{ do } c, \mu)}$$

(SEQUENCE) $$\frac{(c_1, \mu) \longrightarrow \mu'}{(c_1; c_2, \mu) \longrightarrow (c_2, \mu')}$$

$$\frac{(c_1, \mu) \longrightarrow (c_1', \mu')}{(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')}$$

(ATOMICITY) $$\frac{(c, \mu) \longrightarrow^* \mu'}{(\textbf{protect } c, \mu) \longrightarrow \mu'}$$

In rule (ATOMICITY), note that (as usual) $\longrightarrow^*$ denotes the reflexive transitive closure of $\longrightarrow$.

Note that our sequential transition relation $\longrightarrow$ is *deterministic* and *total* (if some obvious restrictions are met):

Lemma 2.1 *If every identifier in c is in $dom(\mu)$ and no subcommand involving* **while** *is* **protect***ed in c, then there is a unique configuration C such that $(c, \mu) \longrightarrow C$.*

Also, the behavior of sequential composition is characterized by the following two lemmas:

Lemma 2.2 *If $(c_1, \mu) \longrightarrow^i \mu'$ and $(c_2, \mu') \longrightarrow^j \mu''$, then $(c_1; c_2, \mu) \longrightarrow^{i+j} \mu''$.*

Lemma 2.3 *If $(c_1; c_2, \mu) \longrightarrow^j \mu'$, then there exist i and μ'' such that $0 < i < j$, $(c_1, \mu) \longrightarrow^i \mu''$, and $(c_2, \mu'') \longrightarrow^{j-i} \mu'$.*

The multi-threaded programs that we consider here consist simply of a set of commands (the threads) running concurrently under a shared memory μ. We model this set as a *thread pool* O, which is a finite function from thread identifiers (α, β, ...) to commands. A pair (O, μ), consisting of a thread pool and a shared memory, is called a *global configuration*.

A multi-threaded program is executed in an interleaving manner, by repeatedly choosing a thread to run for a step. We assume that the choice is made probabilistically, with each thread having an equal probability of being chosen at each step—that is, we assume a *uniform thread scheduler*. We formalize this by defining a global transition relation $\stackrel{p}{\Longrightarrow}$ on global configurations:

(GLOBAL) $$\frac{O(\alpha) = c \quad (c, \mu) \longrightarrow \mu' \quad p = 1/|O|}{(O, \mu) \stackrel{p}{\Longrightarrow} (O - \alpha, \mu')}$$

$$\frac{O(\alpha) = c \quad (c, \mu) \longrightarrow (c', \mu') \quad p = 1/|O|}{(O, \mu) \stackrel{p}{\Longrightarrow} (O[\alpha := c'], \mu')}$$

$$(\{\,\}, \mu) \stackrel{1}{\Longrightarrow} (\{\,\}, \mu)$$

The judgment $(O, \mu) \stackrel{p}{\Longrightarrow} (O', \mu')$ asserts that the probability of going from global configuration (O, μ) to (O', μ') is p. Note that $O - \alpha$ denotes the thread pool obtained by removing thread α from O, and $O[\alpha := c']$ denotes the thread pool obtained by updating the command associated with α to c'. The third rule (GLOBAL), which deals with an empty thread pool, allows us to view a multi-threaded program as a discrete Markov chain [4]. The states of the Markov chain are global configurations and the transition matrix is governed by $\stackrel{p}{\Longrightarrow}$.

3 The Type System

Our type system is based upon the following types:

(*data types*)	τ	::=	$L \mid H$
(*phrase types*)	ρ	::=	$\tau \mid \tau$ *var* $\mid \tau_1$ *cmd* $\tau_2 \mid$
			τ *cmd* n

(R-VAL) $$\frac{\gamma(x) = \tau \ \textit{var}}{\gamma \vdash x : \tau}$$

(INT) $$\gamma \vdash n : L$$

(SUM) $$\frac{\gamma \vdash e_1 : \tau, \ \ \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 + e_2 : \tau}$$

(ASSIGN) $$\frac{\gamma(x) = \tau \ \textit{var}, \ \ \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau \ \textit{cmd} \ 1}$$

(SKIP) $$\gamma \vdash \textbf{skip} : H \ \textit{cmd} \ 1$$

(IF) $$\frac{\begin{array}{l}\gamma \vdash e : \tau \\ \gamma \vdash c_1 : \tau \ \textit{cmd} \ n \\ \gamma \vdash c_2 : \tau \ \textit{cmd} \ n\end{array}}{\gamma \vdash \textbf{if} \ e \ \textbf{then} \ c_1 \ \textbf{else} \ c_2 : \tau \ \textit{cmd} \ n + 1}$$

$$\frac{\begin{array}{l}\gamma \vdash e : \tau_1 \\ \tau_1 \subseteq \tau_2 \\ \gamma \vdash c_1 : \tau_2 \ \textit{cmd} \ \tau_3 \\ \gamma \vdash c_2 : \tau_2 \ \textit{cmd} \ \tau_3\end{array}}{\gamma \vdash \textbf{if} \ e \ \textbf{then} \ c_1 \ \textbf{else} \ c_2 : \tau_2 \ \textit{cmd} \ \tau_1 \vee \tau_3}$$

(WHILE) $$\frac{\begin{array}{l}\gamma \vdash e : \tau_1 \\ \tau_1 \subseteq \tau_2 \\ \tau_3 \subseteq \tau_2 \\ \gamma \vdash c : \tau_2 \ \textit{cmd} \ \tau_3\end{array}}{\gamma \vdash \textbf{while} \ e \ \textbf{do} \ c : \tau_2 \ \textit{cmd} \ \tau_1 \vee \tau_3}$$

(COMPOSE) $$\frac{\begin{array}{l}\gamma \vdash c_1 : \tau \ \textit{cmd} \ m \\ \gamma \vdash c_2 : \tau \ \textit{cmd} \ n\end{array}}{\gamma \vdash c_1; c_2 \ : \ \tau \ \textit{cmd} \ m + n}$$

$$\frac{\begin{array}{l}\gamma \vdash c_1 : \tau_1 \ \textit{cmd} \ \tau_2 \\ \tau_2 \subseteq \tau_3 \\ \gamma \vdash c_2 : \tau_3 \ \textit{cmd} \ \tau_4\end{array}}{\gamma \vdash c_1; c_2 \ : \ \tau_1 \wedge \tau_3 \ \textit{cmd} \ \tau_2 \vee \tau_4}$$

(PROTECT) $$\frac{\begin{array}{l}\gamma \vdash c : \tau_1 \ \textit{cmd} \ \tau_2 \\ c \text{ contains no } \textbf{while} \text{ loops}\end{array}}{\gamma \vdash \textbf{protect} \ c : \tau_1 \ \textit{cmd} \ 1}$$

Figure 1. Typing rules

(BASE) $L \subseteq H$

(CMD) $$\frac{\tau_1' \subseteq \tau_1, \quad \tau_2 \subseteq \tau_2'}{\tau_1 \ \mathit{cmd}\ \tau_2 \subseteq \tau_1' \ \mathit{cmd}\ \tau_2'}$$

$$\frac{\tau' \subseteq \tau}{\tau \ \mathit{cmd}\ n \subseteq \tau' \ \mathit{cmd}\ n}$$

$$\tau \ \mathit{cmd}\ n \subseteq \tau \ \mathit{cmd}\ L$$

(REFLEX) $\rho \subseteq \rho$

(TRANS) $$\frac{\rho_1 \subseteq \rho_2, \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

(SUBSUMP) $$\frac{\gamma \vdash p : \rho_1, \quad \rho_1 \subseteq \rho_2}{\gamma \vdash p : \rho_2}$$

Figure 2. Subtyping rules

The rules of the type system are given in Figures 1 and 2. In the rules (IF), (WHILE), and (COMPOSE), $\vee$ denotes *join* (*least upper bound*) and $\wedge$ denotes *meet* (*greatest lower bound*). The rules allow us to prove *typing judgments* of the form $\gamma \vdash p : \rho$ as well as *subtyping judgments* of the form $\rho_1 \subseteq \rho_2$. Here γ denotes an *identifier typing*, mapping identifiers to phrase types of the form τ *var*. As usual, we say that phrase p is *well typed* under γ if $\gamma \vdash p : \rho$ for some ρ. Similarly, thread pool O is well typed under γ if each thread in O is well typed under γ.

As an example, let's show the derivation of the typing of example 7 from the Introduction:

$$\gamma \vdash y := 5; \textbf{while}\ x + 1\ \textbf{do skip}\ :\ L\ \mathit{cmd}\ H,$$

assuming that $\gamma(x) = H$ *var* and $\gamma(y) = L$ *var*. We have

$$\gamma \vdash 5\ :\ L \tag{1}$$

by rule (INT). Then we get

$$\gamma \vdash y := 5\ :\ L\ \mathit{cmd}\ 1 \tag{2}$$

by rule (ASSIGN) on (1), and

$$\gamma \vdash y := 5\ :\ L\ \mathit{cmd}\ L \tag{3}$$

by rule (SUBSUMP) on (2) using the third rule (CMD). Next we have

$$\gamma \vdash x\ :\ H \tag{4}$$

from rule (R-VAL) and

$$\gamma \vdash 1\ :\ L \tag{5}$$

by rule (INT), which gives

$$\gamma \vdash 1\ :\ H \tag{6}$$

by rule (SUBSUMP) on (5) using rule (BASE), and

$$\gamma \vdash x + 1\ :\ H \tag{7}$$

by rule (SUM) on (4) and (6). Next

$$\gamma \vdash \textbf{skip}\ :\ H\ \mathit{cmd}\ 1 \tag{8}$$

by rule (SKIP), and hence

$$\gamma \vdash \textbf{skip}\ :\ H\ \mathit{cmd}\ L \tag{9}$$

by rule (SUBSUMP) on (8) using the third rule (CMD). Hence we get

$$\gamma \vdash \textbf{while}\ x + 1\ \textbf{do skip}\ :\ H\ \mathit{cmd}\ H \tag{10}$$

by rule (WHILE) on (7) and (9), since $H \subseteq H$ (by rule (REFLEX)) and $L \subseteq H$ (by rule (BASE)), and since $H \vee L = H$. And finally, we get

$$\gamma \vdash y := 5; \textbf{while}\ x + 1\ \textbf{do skip}\ :\ L\ \mathit{cmd}\ H \tag{11}$$

by the second rule (COMPOSE) on (3) and (10), since $L \subseteq H$, $L \wedge H = L$, and $L \vee H = H$.

We now give some discussion of the typing rules.

The first (IF) rule says that an **if** statement takes $n + 1$ steps if both its branches take n steps. This rule can sometimes be used to "pad" a command to eliminate timing leaks, as in the transformation approach proposed by Johan Agat [1]. For example, if $x : H$ and $y : L$, then the thread

if $x = 0$ **then**
    $x := x * x;\ x := x * x$
else
    $x := x + 1;$
$y := 0$

is dangerous, because the time at which y is assigned 0 depends on the value of x. And this program is not well typed under our rules—the **then** branch of the **if** has type H *cmd* 2 and the **else** branch has type H *cmd* 1, which means that the first (IF) rule does not apply. Instead we must coerce the two branches to type H *cmd* L and use the second (IF) rule, which gives the **if** type H *cmd* H. But this makes it illegal (under the second rule (COMPOSE)) to sequentially compose the **if** with the assignment $y := 0$, which has type L *cmd* 1, and $H \not\subseteq L$. To make the program well typed, we can pad the **else** branch to $x := x + 1$; **skip**, which has type H *cmd* 2. Now we can type the **if** using the first (IF) rule, giving it type H *cmd* 3, and then we can give the thread type L *cmd* 4, using the first rule (COMPOSE). It should be noted, however, that Agat's transformation approach is more general that what we can achieve here.

The second rule (IF) is rather complex. One might hope that we could exploit subtyping to simplify the rule, but this is not possible here. We would not want to coerce the type of e up to τ_2, because then it would appear that the execution time of the **if** depends on τ_2 variables. Nor would we want to coerce the types of c_1 and c_2 to τ_1 *cmd* τ_3, because then it would appear that the **if** can assign to τ_1 variables.

We can, however, specialize the second rule (IF) to a pair of rules in the case where L and H are the only security levels; the same specialization can be done to rule (WHILE) and the second rule (COMPOSE). The specialized typing rules are shown in Figure 3.

The constraint $\tau_3 \subseteq \tau_2$ in rule (WHILE) is perhaps surprising,[1] but the typing rules are unsound without it. The problem is that **while** e **do** c implicitly involves sequential composition of c and the entire loop, as shown in the second rule (LOOP). As a result, if c's running time depends on H variables, then c must not assign to L variables. For example, if x is H and y is L, then without the constraint $\tau_3 \subseteq \tau_2$ in rule (WHILE), the following program would be well typed:

while 1 **do**
($y := y + 1$; **while** x **do skip**)

Note that $y := y+1$ has type L *cmd* L and **while** x **do skip** has type H *cmd* H, so the loop body has type L *cmd* H. Hence, without the constraint $\tau_3 \subseteq \tau_2$, the **while** loop can be given type L *cmd* H. But the loop is dangerous—if $x = 0$, then y is incremented only once, and if $x \neq 0$, then y is incremented repeatedly.

Finally, we note that **protect** c takes a command that is guaranteed to terminate and makes it appear to run in just one step. This gives another way of dealing with the example program discussed above; rather than padding the **else** branch, we can just **protect** the **if** (or just its **then** branch), thereby masking any timing differences resulting from different values of x.

[1] Indeed, I did not originally notice the need for it.

4 Properties of the Type System

In this section, we formally establish the properties of the type system. We begin with a lemma that shows that the type system does not restrict a thread at all unless the thread involves both L and H variables:

Lemma 4.1 *Any command involving only L variables has type L cmd L. Any command involving only H variables has type H cmd H.*

Note this is certainly not the case for the type system of [13], since (for example) that system disallows H variables in the guards of **while** loops.

Now we establish the soundness of the type system.

Lemma 4.2 (Simple Security) *If $\gamma \vdash e : L$, then contains only L variables.*

Proof. By induction on the structure of e. □

Lemma 4.3 (Confinement) *If $\gamma \vdash c : H$ cmd τ, then c does not assign to any L variables.*

Proof. By induction on the structure of c. □

Lemma 4.4 (Subject Reduction) *Suppose that $(c, \mu) \longrightarrow (c', \mu')$. If $\gamma \vdash c : \tau_1$ cmd τ_2, then $\gamma \vdash c' : \tau_1$ cmd τ_2. And if $\gamma \vdash c : \tau$ cmd n then $\gamma \vdash c' : \tau$ cmd $(n-1)$.*

Proof. By induction on the structure of c.

The result holds vacuously if c is of the form $x := e$, **skip**, or **protect** c'.

If c is of the form **if** e **then** c_1 **else** c_2, then c' is either c_1 or c_2. Now, if c has type τ *cmd* n, then it must be typed by the first rule (IF), which implies that both c_1 and c_2 have type τ *cmd* $(n-1)$. And if c has type τ_1 *cmd* τ_2, then it is typed either with the first rule (IF) (using the fact that τ_1 *cmd* $m \subseteq \tau_1$ *cmd* τ_2), or with the second rule (IF). In the first case, c_1 and c_2 have type τ_1 *cmd* m, which implies that they also have type τ_1 *cmd* τ_2. In the second case, c_1 and c_2 have type τ_1 *cmd* τ_3, for some τ_3 with $\tau_3 \subseteq \tau_2$. So they have type τ_1 *cmd* τ_2 as well, by rule (SUBSUMP).

If c is of the form **while** e **do** c_1, then c' is of the form $c_1; c$. In this case, c cannot have type τ *cmd* n; it must have type τ_1 *cmd* τ_2 by rule (WHILE). Hence c_1 has type τ_1 *cmd* τ_3 for some τ_3 with $\tau_3 \subseteq \tau_2$ and $\tau_3 \subseteq \tau_1$. Therefore, by the second rule (COMPOSE), $c_1; c$ has type τ_1 *cmd* τ_2.[2]

Finally, if c is of the form $c_1; c_2$, then c' is either c_2 (if $(c_1, \mu) \longrightarrow \mu'$) or $c_1'; c_2$ (if $(c_1, \mu) \longrightarrow (c_1', \mu')$). If c has type

[2] Note that this last step would fail without the constraint $\tau_3 \subseteq \tau_1$.

(IF)
$$\frac{\gamma \vdash e : L \quad \gamma \vdash c_1 : \tau_1 \textit{ cmd } \tau_2 \quad \gamma \vdash c_2 : \tau_1 \textit{ cmd } \tau_2}{\gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \tau_1 \textit{ cmd } \tau_2}$$

$$\frac{\gamma \vdash e : H \quad \gamma \vdash c_1 : H \textit{ cmd } H \quad \gamma \vdash c_2 : H \textit{ cmd } H}{\gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : H \textit{ cmd } H}$$

(WHILE)
$$\frac{\gamma \vdash e : L \quad \tau_2 \subseteq \tau_1 \quad \gamma \vdash c : \tau_1 \textit{ cmd } \tau_2}{\gamma \vdash \textbf{while } e \textbf{ do } c : \tau_1 \textit{ cmd } \tau_2}$$

$$\frac{\gamma \vdash e : H \quad \gamma \vdash c : H \textit{ cmd } H}{\gamma \vdash \textbf{while } e \textbf{ do } c : H \textit{ cmd } H}$$

(COMPOSE)
$$\frac{\gamma \vdash c_1 : \tau_1 \textit{ cmd } L \quad \gamma \vdash c_2 : \tau_1 \textit{ cmd } \tau_2}{\gamma \vdash c_1; c_2 \ : \ \tau_1 \textit{ cmd } \tau_2}$$

$$\frac{\gamma \vdash c_1 : \tau \textit{ cmd } H \quad \gamma \vdash c_2 : H \textit{ cmd } H}{\gamma \vdash c_1; c_2 \ : \ \tau \textit{ cmd } H}$$

Figure 3. Typing rules specialized to L and H

τ *cmd* n, then it must be typed by the first rule (COMPOSE) which means that c_1 has type τ *cmd* k and c_2 has type τ *cmd* l for some k and l with $k + l = n$. If c' is c_2, then we must have $k = 1$, so c_2 has type τ *cmd* $(n-1)$. If c' is $c_1'; c_2$, then by induction c_1' has type τ *cmd* $(k-1)$, and therefore c' has type τ *cmd* $(k-1+l) = \tau$ *cmd* $(n-1)$. And if c has type τ_1 *cmd* τ_2, then it must be typed by the second rule (COMPOSE) which means that c_1 has type τ_3 *cmd* τ_4 and c_2 has type τ_5 *cmd* τ_6, for some τ_3, τ_4, τ_5, and τ_6 satisfying the subtyping constraints $\tau_4 \subseteq \tau_5$, $\tau_4 \subseteq \tau_2$, $\tau_6 \subseteq \tau_2$, $\tau_1 \subseteq \tau_3$, and $\tau_1 \subseteq \tau_5$. Now, if c' is c_2, then by rule (SUBSUMP) it also has type τ_1 *cmd* τ_2, since τ_5 *cmd* $\tau_6 \subseteq \tau_1$ *cmd* τ_2. And if c' is $c_1'; c_2$, then by induction c_1' has type τ_3 *cmd* τ_4, and therefore c' has type τ_1 *cmd* τ_2. □

Lemma 4.5 *If* $\gamma \vdash c : \tau$ *cmd* 1 *and* $dom(\mu) = dom(\gamma)$, *then* $(c, \mu) \longrightarrow \mu'$ *for some* μ'.

Proof. The only commands with type τ *cmd* 1 are $x := e$, **skip**, and **protect** c_1. The result is immediate in the first two cases; in the case of **protect** c_1 we note that if c_1 is well typed and free of **while** loops, then c_1 is guaranteed to terminate. □

Definition 4.1 *Memories μ and ν are equivalent with respect to γ, written $\mu \sim_\gamma \nu$, if μ, ν, and γ have the same domain and μ and ν agree on all L variables.*

We now explore the behavior of a well-typed command c when run under two equivalent memories. We begin with a Mutual Termination lemma for **while**-free programs:

Lemma 4.6 (Mutual Termination) *Let c be a command containing no* **while** *loops. If c is well typed under γ, $\mu \sim_\gamma \nu$, and $(c, \mu) \longrightarrow^* \mu'$, then there is a ν' such that $(c, \nu) \longrightarrow^* \nu'$ and $\mu' \sim_\gamma \nu'$.*

Proof. Similar to Lemma 5.6 of [13]. □

In the context of multi-threaded programs, however, it is not enough to consider only the final memory resulting from the execution of c (as in the Mutual Termination lemma); we must also consider timing. The key property that lets us establish probabilistic noninterference is this: if a well-typed command c is run under two equivalent memories, it makes exactly the same sequence of assignments to L variables, *at the same times.* Hence the two memories will remain equivalent after every execution step. (Note however that c may terminate faster under one memory than the other; but then the slower execution will not make any more assignments to L variables.)

Here's an example that illustrates some of the working of the type system. Suppose that c is a well-typed command of the form

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2); c_3.$$

What happens when c is run under two equivalent memories μ and ν? If $e : L$, then $\mu(e) = \nu(e)$ by Simple Security, and hence both executions will choose the same branch. If, instead, $e : H$, then the two executions may choose different branches. But if the **if** is typed using the first rule (IF), then both branches have type H *cmd* n for some n. Therefore neither branch assigns to L variables, by Confinement, and both branches terminate after n steps, by Subject Reduction. Hence both executions will reach c_3 at the same time, and the memories will still be equivalent. And if the **if** is typed using the second rule (IF), then it gets type H *cmd* H, which is also the type given to each branch. Again, neither branch assigns to L variables, by Confinement. Now, in this case the two branches may not terminate at the same time—indeed, one may terminate and the other may not. But the entire command $(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2); c_3$ will have to be typed by the second rule (COMPOSE), which means that $c_3 : H$ *cmd* H. Hence c_3 makes no assignments to L variables, which means that, as far as L variables are concerned, it doesn't matter when (or even whether) c_3 is executed.

To formalize these ideas, we need to define a notion of equivalence on configurations; then we can argue that $\longrightarrow$ takes equivalent configurations to equivalent configurations. But first we make some observations about sequential composition. Any command c can be written in the *standard form*

$$(\ldots((c_1; c_2); c_3); \ldots); c_k$$

for some $k \geq 1$, where c_1 is *not* a sequential composition (but c_2 through c_k might be sequential compositions). If we adopt the convention that sequential composition associates to the left, then we can write this more simply as

$$c_1; c_2; c_3; \ldots; c_k.$$

Now, if c is executed, it follows from the (SEQUENCE) rules that the first execution step touches only c_1; that is, we have either

$$(c_1; c_2; c_3; \ldots; c_k, \mu) \longrightarrow (c_2; c_3; \ldots; c_k, \mu'),$$

if $(c_1, \mu) \longrightarrow \mu'$, or else

$$(c_1; c_2; c_3; \ldots; c_k, \mu) \longrightarrow (c_1'; c_2; c_3; \ldots; c_k, \mu'),$$

if $(c_1, \mu) \longrightarrow (c_1', \mu')$. Now we define our notion of equivalence on commands:

Definition 4.2 *We say that commands c and d are equivalent with respect to γ, written $c \sim_\gamma d$, if c and d are both well typed under γ and either*

- $c = d$,
- *c and d both have type H cmd τ, or*
- *c has standard form $c_1; c_2; c_3; \ldots; c_k$, d has standard form $d_1; c_2; c_3; \ldots; c_k$, for some k, and c_1 and d_1 both have type H cmd n for some n.*

We extend the notion of equivalence to configurations by saying that configurations C and D are equivalent, written $C \sim_\gamma D$, if any of the following four cases applies:

- C is of the form (c, μ), D is of the form (d, ν), $c \sim_\gamma d$, and $\mu \sim_\gamma \nu$.
- C is of the form (c, μ), D is of the form ν, c has type H *cmd* τ, and $\mu \sim_\gamma \nu$.
- C is of the form μ, D is of the form (d, ν), d has type H *cmd* τ, and $\mu \sim_\gamma \nu$.
- C is of the form μ, D is of the form ν, and $\mu \sim_\gamma \nu$.

(In effect, we are saying that a command of type H *cmd* τ is equivalent to a terminated command.)

Theorem 4.7 (Sequential Noninterference) *Suppose that $(c, \mu) \sim_\gamma (d, \nu)$, $(c, \mu) \longrightarrow C'$, and $(d, \nu) \longrightarrow D'$. Then $C' \sim_\gamma D'$.*

Proof. We begin by dealing with the case when c and d both have type H *cmd* τ, for some τ. In this case, by the Confinement Lemma, neither c nor d assigns to any L variables. Hence the memories of C' and D' remain equivalent. Now, if C' is of the form (c', μ') and D' is of the form (d', ν'), then by the Subject Reduction Lemma, c' and d' both have type H *cmd* τ, which gives $c' \sim_\gamma d'$. The cases when C' is of the form μ' and/or D' is of the form ν' are similar.

Now consider the case where c and d do not both have type H *cmd* τ. Let c have standard form $c_1; c_2; c_3; \ldots; c_k$. We can see from the definition of $\sim_\gamma$ that either $c = d$ or d has standard form $d_1; c_2; c_3; \ldots; c_k$, where c_1 and d_1 both have type H *cmd* n for some n.

In the latter case, we have by the Confinement Lemma that neither c_1 nor d_1 assigns to any L variables. Hence the memories of C' and D' are equivalent. And if $n > 1$, then by the Subject Reduction Lemma, C' and D' are of the form $(c_1'; c_2; c_3; \ldots; c_k, \mu')$ and $(d_1'; c_2; c_3; \ldots; c_k, \nu')$, respectively, where c_1' and d_1' both have type H *cmd* $(n-1)$. Thus $C' \sim_\gamma D'$. And if $n = 1$, then C' and D' are of the form $(c_2; c_3; \ldots; c_k, \mu')$ and $(c_2; c_3; \ldots; c_k, \nu')$, respectively.[3] So again $C' \sim_\gamma D'$.

We are left, finally, with the case where $c = d$. Let them have standard form $c_1; c_2; c_3; \ldots; c_k$ and consider in turn each of the possible forms of c_1:

[3] Actually, if $k = 1$ then C' and D' are just μ' and ν' here. We'll ignore this point in the rest of this proof.

Case $x := e$**.** In this case, we have that C' is

$$(c_2; c_3; \ldots; c_k, \mu[x := \mu(e)])$$

and D' is

$$(c_2; c_3; \ldots; c_k, \nu[x := \nu(e)]).$$

Now if x is H, then $\mu[x := \mu(e)] \sim_\gamma \nu[x := \nu(e)]$. And if x is L, then by rule (ASSIGN) $e : L$. Hence $\mu(e) = \nu(e)$ by Simple Security, so again

$$\mu[x := \mu(e)] \sim_\gamma \nu[x := \nu(e)].$$

Therefore $C' \sim_\gamma D'$.

Case skip. In this case C' is $(c_2; c_3; \ldots; c_k, \mu)$ and D' is $(c_2; c_3; \ldots; c_k, \nu)$. So $C' \sim_\gamma D'$.

Case if e **then** c_{11} **else** c_{12}**.** If $e : L$, then by Simple Security $\mu(e) = \nu(e)$. Hence C' and D' both choose the same branch. That is, either

$$C' = (c_{11}; c_2; c_3; \ldots; c_k, \mu)$$

and

$$D' = (c_{11}; c_2; c_3; \ldots; c_k, \nu),$$

or else

$$C' = (c_{12}; c_2; c_3; \ldots; c_k, \mu)$$

and

$$D' = (c_{12}; c_2; c_3; \ldots; c_k, \nu).$$

So $C' \sim_\gamma D'$.

If e doesn't have type L, then if c_1 is typed by the first rule (IF), then we have that c_{11} and c_{12} both have type H *cmd* n for some n. Therefore, whether or not C' and D' take the same branch, we have $C' \sim_\gamma D'$.

And if c_1 is typed by the second rule (IF), than it gets type H *cmd* H. But, by rule (COMPOSE), this means that $c_1; c_2$ also has type H *cmd* H. This in turn implies that $c_1; c_2; c_3$ has type H *cmd* H, and so on, until we get that c has type H *cmd* H. So, since $c = d$ here, this case has already been handled.

Case while e **do** c_{11}**.** As in the case of **if**, if $e : L$, then by Simple Security $\mu(e) = \nu(e)$. Hence the two computations stay together. That is, either $C' = (c_2; c_3; \ldots; c_k, \mu)$ and $D' = (c_2; c_3; \ldots; c_k, \nu)$, or else

$$C' = (c_{11}; \textbf{while } e \textbf{ do } c_{11}; c_2; c_3; \ldots; c_k, \mu)$$

and

$$D' = (c_{11}; \textbf{while } e \textbf{ do } c_{11}; c_2; c_3; \ldots; c_k, \nu).$$

So $C' \sim_\gamma D'$.

If e doesn't have type L, then by rule (WHILE) we have that $c_1 : H$ *cmd* H. As in the case of **if**, this implies that $c : H$ *cmd* H, so this case has again already been handled.

Case protect c_{11}**.** By rule (PROTECT), c_{11} contains no **while** loops. Hence this case follows from the Mutual Termination lemma.

□

We remark here that if c is well typed and $\mu \sim_\gamma \nu$, then $(c, \mu) \sim_\gamma (c, \nu)$. Hence, by applying the Sequential Noninterference Theorem repeatedly, we see that, for all k, the configuration reached from (c, μ) after k steps will be equivalent to the configuration reached from (c, ν) after k steps. Hence, as we claimed above, the two executions make exactly the same assignments to L variables, at the same times.

Now we change our focus from the execution of an *individual* thread c, which is deterministic, to the execution of a *pool* of threads O, which is a Markov chain. Our first thought, given the Sequential Noninterference Theorem, may be that if O is well typed and $\mu \sim_\gamma \nu$, then (O, μ) and (O, ν) give rise to the *same* (i.e. isomorphic) Markov chains. But this isn't quite right. For example, suppose that O is $\{\alpha : (x := x * x), \quad \beta : (x := x + 1)\}$. If x is H, then memories $\{x = 0\}$ and $\{x = 1\}$ are equivalent. But running O from $\{x = 0\}$ gives a Markov chain with 4 states: $(O, \{x = 0\})$, $(\{\beta : (x := x + 1)\}, \{x = 0\})$, $(\{\alpha : (x := x * x)\}, \{x = 1\})$, and $(\{\}, \{x = 1\})$; and running O from $\{x = 1\}$ gives a Markov chain with 5 states: $(O, \{x = 1\})$, $(\{\beta : (x := x + 1)\}, \{x = 1\})$, $(\{\alpha : (x := x * x)\}, \{x = 2\})$, $(\{\}, \{x = 2\})$, and $(\{\}, \{x = 4\})$. Note however that the last two states, $(\{\}, \{x = 2\})$ and $(\{\}, \{x = 4\})$ should be considered equivalent; thus we might feel that the two Markov chains are basically the same after all.

Formally, what we need is to construct a *quotient Markov chain*. That is, given a Markov chain with state set S and an equivalence relation $\sim$ on S, we'd like to form a new Markov chain $S/\sim$ whose states are the *equivalence classes* of S under $\sim$. But when is this possible? Kemeny and Snell, who refer to the issue as "lumpability", identified the needed condition long ago [7, p. 124]:

> If $s_1 \sim s_2$, then for each equivalence class A, the probability of going from s_1 to a state in A is the same as the probability of going from s_2 to a state in A:
>
> $$\sum_{a \in A} p_{s_1 a} = \sum_{a \in A} p_{s_2 a}$$

(Here $p_{s_1 a}$ denotes the probability of going in one step from s_1 to a in the original Markov chain.) In this case, we can define the transition probabilities for $S/\sim$ as follows: the probability of going from equivalence class A to equivalence class B, denoted p_{AB}, is $\sum_{b \in B} p_{ab}$, where a is any element of A; the above condition says exactly that the choice of a makes no difference. Within computer science, this

idea later appeared in the work of Larsen and Skou [8], who (noting the analogy to bisimulation) introduced the name *probabilistic bisimulation* for such an equivalence relation $\sim$. In more recent work, Hermanns [5] includes a lengthy discussion of probabilistic bisimulation, and Sabelfeld and Sands [9] apply it to probabilistic noninterference.

The key property of $S/\sim$ is this [5, p. 49]:

Theorem 4.8 *Suppose that $\sim$ is a probabilistic bisimulation. Then for each starting state s, equivalence class A, and integer k, the probability that S, starting from s, will end up in a state in A after k steps is equal to the probability that $S/\sim$, starting from $[s]$ (the equivalence class of s), will end up in state A after k steps.*

Now, returning to our specific system, we want to define a probabilistic bisimulation, which we'll still call $\sim_\gamma$, on the set of global configurations (O, μ). Under the (more restrictive) type system of [13], it would suffice to define $(O_1, \mu) \sim_\gamma (O_2, \nu)$ iff $O_1 = O_2$ and $\mu \sim_\gamma \nu$. But here we need a looser notion, since (as discussed above) the executions of a well-typed command c under two equivalent memories μ and ν can be quite different. Roughly, we want to have $(O_1, \mu) \sim_\gamma (O_2, \nu)$ if $\mu \sim_\gamma \nu$ and $O_1(\alpha) \sim_\gamma O_2(\alpha)$ for all α. But actually O_1 and O_2 could have different domains, since changing the values of H variables can affect the running time of well-typed commands. So we'd like to say that O_1 can have extra threads in it, so long as they have type H *cmd* τ, (and similarly for O_2). Unfortunately, this won't work within our framework of probabilistic bisimulation, because the transition probabilities will not be the same in this case. For example, if $O_1 = \{\alpha : (y := 1), \quad \beta : \textbf{skip}\}$ and $O_2 = \{\alpha : (y := 1)\}$, where y is L, then $(O_2, \{y = 0\})$ goes to $(\{\}, \{y = 1\})$ with probability 1, but $(O_1, \{y = 0\})$ goes to the equivalent configuration $(\{\beta : \textbf{skip}\}, \{y = 1\})$ with probability only $1/2$.[4]

So to get a probabilistic bisimulation, we need a stronger definition that requires equality of domains:

Definition 4.3 $(O_1, \mu) \sim_\gamma (O_2, \nu)$ *if* $\mathit{dom}(O_1) = \mathit{dom}(O_2)$, $O_1(\alpha) \sim_\gamma O_2(\alpha)$ *for all* $\alpha \in \mathit{dom}(O_1)$, *and* $\mu \sim_\gamma \nu$.

And, regrettably, we have to change our rule (GLOBAL) to prevent the thread pool domain from ever changing:

$$\text{(GLOBAL)} \qquad \frac{\begin{array}{l} O(\alpha) = c \\ (c, \mu) \longrightarrow \mu' \\ p = 1/|O| \end{array}}{(O, \mu) \xrightarrow{p} (O[\alpha := \textbf{skip}], \mu')}$$

$$\frac{\begin{array}{l} O(\alpha) = c \\ (c, \mu) \longrightarrow (c', \mu') \\ p = 1/|O| \end{array}}{(O, \mu) \xrightarrow{p} (O[\alpha := c'], \mu')}$$

[4]Note that intuitively there is no information leakage here; it's just a question of one program running more slowly than the other, which is not observable internally.

The new semantics says, in effect, that a completed thread remains alive, wasting processor time.[5] With this change, we can now prove what we want:

Theorem 4.9 $\sim_\gamma$ *is a probabilistic bisimulation on the set of global configurations.*

Proof. Suppose that $(O, \mu) \sim_\gamma (O', \nu)$ and suppose that $\mathit{dom}(O)$ and $\mathit{dom}(O')$ are $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Then (O, μ) and (O', ν) can each go to n (not necessarily distinct) global configurations, (O_1, μ_1), (O_2, μ_2), $\ldots$, (O_n, μ_n), and (O'_1, ν_1), (O'_2, ν_2), $\ldots$, (O'_n, ν_n), each with probability $1/n$, where (O_i, μ_i) and (O'_i, ν_i) denote the global configurations reached by choosing thread α_i. Now, since $O(\alpha_i) \sim_\gamma O'(\alpha_i)$, we have by the Sequential Noninterference Theorem that $(O_i, \mu_i) \sim_\gamma (O'_i, \nu_i)$ for all i. This implies that the probabilities of reaching any equivalence class from (O, μ) or from (O', ν) are the same. □

Finally we can argue that well-typed programs satisfy probabilistic noninterference: if O is well typed and $\mu \sim_\gamma \nu$, then $(O, \mu) \sim_\gamma (O, \nu)$. Hence, by Theorem 4.8, the probability that the L variables have certain values after k steps is the same when starting from (O, μ) as when starting from (O, ν).

5 Conclusion

The new type system allows probabilistic noninterference to be guaranteed for a much larger class of programs than previously permitted. In particular, there seems to be hope that the system might not be too hard to accommodate in practice, since any threads that involve only H variables or only L variables are automatically well typed.

As in previous work, we have assumed that program execution is observable only internally; with external observations of running time, timing leaks are certainly possible, because **protect**ed commands won't really execute in one time step. However, if a program is well typed without the use of **protect**, then it seems possible in principle to allow external observations, except that our simple timing model is unrealistic. For example, our semantics specifies that $x := x * x$ and **skip** each execute in 1 step, which seems to require an implementation that wastes a lot of time. Agat [2] has recently attempted to tackle external timing leaks in the realistic setting of Java byte-code programs, which requires the consideration of a host of delicate timing issues, such as caching.

On the other hand, if we are concerned only with internal timing leaks, then we don't really need such precise

[5]Also note that we actually have to do some summing in our $\xrightarrow{p}$ relation, since now it may be possible for (O, μ) to reach (O', μ') in more than one way; see [9, p. 202].

timings. In particular, if doesn't matter to us how much time $x := x * x$ and **skip** actually take, so long as rule (GLOBAL) is implemented faithfully, which means simply that the scheduler randomly picks a new thread after each computation step. Of course, doing scheduling with such fine granularity would appear to involve high overhead; it remains to be seen whether acceptable performance could be achieved with such a scheduler. This needs more study.

In other future work, it would be desirable to find a weaker notion of probabilistic bisimulation that would allow our original rule (GLOBAL) to be used. Also, it would be useful to investigate type inference for the new system, presumably using the approach of [12]. Finally, it would be valuable to integrate cryptography into the framework of this paper by using a weaker notion of noninterference based on computational complexity; some preliminary steps in this direction have been made [14, 11].

6 Acknowledgments

This work was partially supported by the National Science Foundation under grant CCR-9900951. I am grateful to Andrei Sabelfeld, David Sands, Dennis Volpano, and the anonymous reviewers for their useful comments on this work.

References

[1] J. Agat. Transforming out timing leaks. In *Proceedings 27th Symposium on Principles of Programming Languages*, pages 40–53, Boston, MA, Jan. 2000.

[2] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, Dec. 2000.

[3] D. Denning and P. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[4] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley & Sons, Inc., third edition, 1968.

[5] H. Hermanns. *Interactive Markov Chains*. PhD thesis, University of Erlangen-Nürnberg, July 1998.

[6] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proceedings 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199, Apr. 2000.

[7] J. Kemeny and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand, 1960.

[8] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[9] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, UK, July 2000.

[10] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings 25th Symposium on Principles of Programming Languages*, pages 355–364, San Diego, CA, Jan. 1998.

[11] D. Volpano. Secure introduction of one-way functions. In *Proceedings 13th IEEE Computer Security Foundations Workshop*, pages 246–254, Cambridge, UK, June 2000.

[12] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, Apr. 1997.

[13] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.

[14] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proceedings 27th Symposium on Principles of Programming Languages*, pages 268–276, Boston, MA, Jan. 2000.

[15] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.