Secure Information Flow in a Multi-threaded Imperative Language

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

Dennis Volpano
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943, USA
volpano@cs.nps.navy.mil

Abstract

Previously, we developed a type system to ensure secure information flow in a sequential, imperative programming language [VSI96]. Program variables are classified as either *high* or *low* security; intuitively, we wish to prevent information from flowing from high variables to low variables. Here, we extend the analysis to deal with a multi-threaded language. We show that the previous type system is insufficient to ensure a desirable security property called *noninterference*. Noninterference basically means that the final values of low variables are independent of the initial values of high variables. By modifying the sequential type system, we are able to guarantee noninterference for concurrent programs. Crucial to this result, however, is the use of purely nondeterministic thread scheduling. Since implementing such scheduling is problematic, we also show how a more restrictive type system can guarantee noninterference, given a more deterministic (and easily implementable) scheduling policy, such as round-robin time slicing. Finally, we consider the consequences of adding a clock to the language.

1 Introduction

The success of mobile code technologies depends in large part on what kinds of security guarantees can be made for clients executing the code. Among the concerns here is ensuring that code respects a client's *privacy*, so that sensitive information is not improperly disclosed. Current software approaches to security address the issue of protecting privacy by introducing protection domains and access privileges. The basic idea is to specify, via a security policy, a set of privileges for a piece of code based on its digital signature. A check is then made for a certain access privilege when the code attempts to cross a domain boundary, say for example if it attempts to access the local file system. If the privilege has been granted, execution proceeds. Keep in mind that the decision is made here against a security policy for the code's signature, not the code itself. This is the approach taken in the security architecture of the Java Developer's Kit (JDK) 1.2 [GMPS97] and in the extended stack introspection proposal of [WBDF97].

But suppose we can prove that a program satisfies a secure information flow property that guarantees that the program respects private information. Then there is no need to check at runtime whether the code has permission to read private information; we can simply trust it, since the property guarantees that the information will not be improperly disclosed. This is the approach taken in this paper. We are interested in developing a type system for a concurrent programming language and exploring the secure-flow properties that can be shown to hold for all well-typed programs. With such a type system, code can be type checked for secure-flow violations just once. Code that type checks can be allowed to run and access private information without any further checks.[1] Type checking might be done by a client's security architecture. Another way it might be done is at a code certification site. For example, efforts are underway at some companies in the U.S. to "certify" the security of Java compilation units used in electronic commerce servers. (It is understandable why consumer confidence is low here given the rash of stolen credit card numbers despite the use of encryption.) Such a site might apply a type checker as an initial step in certifying code.

This paper continues our earlier work [VSI96, VS97b, VS97a] on the relationship between typing, security properties, and semantics, but now in a concurrent setting. The paper presents the following results:

1. We show that the type system of [VSI96] is no longer sufficient to guarantee a desirable security property, called *noninterference*, if we add threads to our language. The noninterference property is intended to assert that information cannot flow from high variables to low variables; basically, it says that the final values of low variables are independent of the initial values of high variables.

2. We show that the noninterference property can be restored in a multi-threaded language by requiring the guards of **while** loops to have type *low* and by requiring **while** loops themselves to have type *low cmd*. (Conditionals do not need to be restricted.) This is the main result of the paper.

3. Crucial to the above result, however, is the use of purely nondeterministic thread scheduling. It is not

To appear in the 25th ACM Symposium on Principles of Programming Languages, San Diego, California, January 19–21, 1998.

[1] We do not mean to suggest that such a type system would address *all* security concerns. Integrity properties, for instance, might well be best handled by code signing.

clear how such scheduling can be implemented in practice. We show that with more deterministic scheduling, such as round-robin time slicing (which is used in the implementation of Java threads in Windows NT 4.0), the noninterference property does not hold. We show that noninterference can be restored, regardless of the scheduling policy used, by also requiring the guards of conditionals to have type *low*.

4. We consider adding a clock to the language. We show that unless the clock is given type *high*, noninterference is not preserved.

The remainder of the paper is organized as follows. In Section 2, we give an example that shows that the type system of [VSI96] is insufficient to ensure noninterference in a multi-threaded language. In Sections 3 and 4, we formally define the semantics of our multi-threaded language and its type system. Then, in Section 5, we prove that the type system guarantees the noninterference property. In Section 6, we explore how adding a clock to the language affects the noninterference property. In Section 7, we consider the consequences of using a less nondeterministic (but more implementable) semantics of concurrency. In Section 8, we discuss some interactions among the noninterference property and language semantics. Finally, in Section 9, we discuss some related work.

2 The Effect of Threads on Noninterference

Recently, the authors showed that a Denning-style secure-flow analysis of imperative programs can be formulated as a *type system* [VSI96]. For example, suppose that we wish to support two security classes, L (low) and H (high). Then we can use these security classes as the types of program variables. Thus, for variables x and y, we can say $x : H$ to indicate that x holds high-security information and $y : L$ to indicate that y holds low-security information. And then an improper assignment like $y := x$ can be caught as a type error. Note, however, that the opposite assignment $x := y$ should be allowed; to deal with this we introduce *subtyping* into our type system and say that $L \subseteq H$. More subtly, the type system must also guard against *implicit* information flows, as seen in a program like

$$\textbf{if } even(x) \textbf{ then } y := 0 \textbf{ else } y := 1,$$

which indirectly copies the last bit of x into y. To deal with such implicit flows, the type system also classifies program *commands* as having either type $H\ cmd$ or $L\ cmd$; intuitively, a command of type $H\ cmd$ cannot transmit any information to L variables and hence can safely be used in the branches of a conditional whose guard has type H.

In [VSI96], it is shown that the type system ensures that every well-typed program c satisfies a *noninterference* property, which can be described as follows: suppose that μ and ν are two memories that agree on all L variables and that c can be run successfully starting from both μ and ν, yielding final memories μ' and ν'. Then μ' and ν' also agree on all L variables. Intuitively, this means that information cannot "leak" from H variables to L variables, since the final values of L variables are independent of the initial values of H variables.[2] Furthermore, programs can be checked automatically for type correctness, by doing *type inference* [VS97b].

[2] It is possible, however, for information about H variables to leak to an *outside* observer who can observe whether c halts, aborts, or fails to terminate, or how long c takes to terminate. See [VS97a] for some approaches to eliminating such covert information flows.

However, the language considered in [VSI96] is sequential, while mobile programs (such as Java applets) are often multi-threaded. For this reason, it is important to extend our analysis to deal with a multi-threaded language and to see how the noninterference property is affected by the presence of concurrency. This is the main goal of this paper.

We begin with an example that shows that the type system of [VSI96] is no longer sufficient to ensure noninterference if we extend our language with concurrent threads that communicate via a shared memory. The program, which consists of three threads, is given in Figure 1. Assume that `PIN` : $H\ var$ and `result` : $L\ var$. Then each of the threads in this program can be typed under the type system of [VSI96]. (The typing gives `trigger0` and `trigger1` type $H\ var$, and `maintrigger` and `mask` type $L\ var$.)

But, if the program is run in a memory where initially `maintrigger` $= 0$, `trigger0` $= 0$, `trigger1` $= 0$, `result` $= 0$, `mask` is a power a 2, and `PIN` is an arbitrary natural number less twice `mask`, then, assuming that scheduling is fair (i.e. each thread is scheduled infinitely often), the program eventually halts with the value of `PIN` copied into `result`. Thus the noninterference property is violated.

To restore the noninterference property in this concurrent setting, we impose two new restrictions on the typing of **while** loops: we require that the guard of a **while** loop have type L, and we require the **while** loop itself to get type $L\ cmd$. The new restrictions succeed in ruling out the above program—since `trigger0` and `trigger1` have type H, they cannot be used in the guards of the **while** loops in threads α and β.

In the next three sections, we develop these ideas precisely, proving that the new restrictions on **while** loops are sufficient to restore the noninterference property for multi-threaded programs.

3 Syntax and Semantics

Threads are written in the simple imperative language:

$$\begin{array}{lrcl}
(phrases) & p & ::= & e \mid c \\
(expressions) & e & ::= & x \mid n \mid e_1 + e_2 \mid \\
 & & & e_1 - e_2 \mid e_1 = e_2 \mid \ldots \\
(commands) & c & ::= & x := e \mid c_1; c_2 \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & & \textbf{while } e \textbf{ do } c
\end{array}$$

Metavariable x ranges over identifiers and n over integer literals. Integers are the only values; we use 0 for false and nonzero for true. Note that expressions are all pure (i.e. they do not cause side effects) and total (i.e. they do not contain partial operations like division).

The concurrent systems that we consider here consist simply of a set of commands (the threads) that run concurrently; we do not consider facilities for creating new threads. Following the approach taken in Cliff Jones's $\pi o \beta \lambda$ [Jon96], we model a system of threads with an *object map* O, which is simply a finite function from thread identifiers (α, β, ...) to commands. In addition, there is a single global memory μ, shared by all threads, that maps identifiers to integers. (Note that in this simple context, we don't need to distinguish identifiers from locations.) The only way that threads can interact is via the shared memory.

In this paper, 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. Note

- Thread α:

```
while mask != 0 do
    while trigger0 = 0 do
        ;
    result := result | mask;   // bitwise 'or'
    trigger0 := 0;
    maintrigger := maintrigger+1;
    if maintrigger = 1 then trigger1 := 1
```

- Thread β:

```
while mask != 0 do
    while trigger1 = 0 do
        ;
    result := result & ~mask;  // bitwise 'and' with the complement of mask
    trigger1 := 0;
    maintrigger := maintrigger+1;
    if maintrigger = 1 then trigger0 := 1
```

- Thread γ:

```
while mask != 0 do
    maintrigger := 0;
    if (PIN & mask) = 0 then
        trigger0 := 1
    else
        trigger1 := 1;
    while maintrigger != 2 do
        ;
    mask := mask / 2;
trigger0 := 1;
trigger1 := 1
```

Figure 1: A multi-threaded program that leaks information

that $\mu(e)$ is always defined, provided that every identifier occurring in e is in the domain of μ, which will always be the case if e is well typed.

As in Gunter [Gun92], we define the semantics of commands via transitions:

$$\text{(UPDATE)} \quad \frac{x \in dom(\mu)}{(x := e, \mu) \xrightarrow{s} \mu[x := \mu(e)]}$$

$$\text{(SEQUENCE)} \quad \frac{(c_1, \mu) \xrightarrow{s} \mu'}{(c_1; c_2, \mu) \xrightarrow{s} (c_2, \mu')} \qquad \frac{(c_1, \mu) \xrightarrow{s} (c_1', \mu')}{(c_1; c_2, \mu) \xrightarrow{s} (c_1'; c_2, \mu')}$$

$$\text{(BRANCH)} \quad \frac{\mu(e) \text{ nonzero}}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{s} (c_1, \mu)} \qquad \frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{s} (c_2, \mu)}$$

$$\text{(LOOP)} \quad \frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{s} \mu} \qquad \frac{\mu(e) \text{ nonzero}}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{s} (c; \textbf{while } e \textbf{ do } c, \mu)}$$

These rules define a transition relation $\xrightarrow{s}$ on *configurations*. A configuration 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. We write $\xrightarrow{s}^k$ for the k-fold self composition of $\xrightarrow{s}$, and $\xrightarrow{s}^*$ for the reflexive, transitive closure of $\xrightarrow{s}$.

Next we have two rules specifying the global transitions that can be made by a system of threads:

$$\text{(GLOBAL)} \quad \frac{O(\alpha) = c \quad (c, \mu) \xrightarrow{s} \mu'}{(O, \mu) \xrightarrow{g} (O - \alpha, \mu')} \qquad \frac{O(\alpha) = c \quad (c, \mu) \xrightarrow{s} (c', \mu')}{(O, \mu) \xrightarrow{g} (O[\alpha := c'], \mu')}$$

The semantics, at the global level, is thus purely nondeterministic. (At this point, we don't even require that scheduling be fair.) How to implement this semantics is an open question; this will be discussed further in Sections 7 and 8.

4 The Type System

Here are the types used by our type system:

$$\begin{array}{lll} (\textit{data types}) & \tau & ::= L \mid H \\ (\textit{phrase types}) & \rho & ::= \tau \mid \tau \ var \mid \tau \ cmd \end{array}$$

For simplicity, we limit the security classes here to just L and H; it is possible to generalize to an arbitrary partial order of security classes.

Our type system, whose rules are given in Figure 2, allows 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*, which is a finite function from identifiers to phrase types.

If $\gamma \vdash c : \rho$ for some ρ, then we say that c is *well typed under* γ. Also, if $O(\alpha)$ is well typed under γ for every $\alpha \in dom(O)$, then we say that O is well typed under γ.

As compared with the type system of [VSI96], the typings of **while** loops are here restricted in two ways: first, the guard of a **while** loop must have type L, and second, the **while** loop itself can only get type L *cmd*.

5 Type Soundness

We begin with three lemmas that establish the key properties ensured by the type system; these lemmas are then used to prove that well-typed programs have the noninterference property.

Lemma 5.1 (Simple Security) *If $\gamma \vdash e : L$, then every identifier in e has class L.*

Proof. By induction on the structure of e. □

Lemma 5.2 (Confinement) *If $\gamma \vdash c : H$ cmd, then every identifier assigned to in c has class H, and c is guaranteed to terminate successfully from any memory μ where $dom(\mu) = dom(\gamma)$.*

Proof. By induction on the structure of c, and using the fact that c cannot contain any **while** loops. □

Lemma 5.3 (Subject Reduction) *If $\gamma \vdash c : \tau$ cmd and $(c, \mu) \xrightarrow{s} (c', \mu')$, then $\gamma \vdash c' : \tau$ cmd.*

Proof. By induction on the structure of c.

If c is of the form $c_1; c_2$, then it follows that $\gamma \vdash c_1 : \tau$ *cmd* and $\gamma \vdash c_2 : \tau$ *cmd*. (The argument for this is complicated somewhat by the presence of subtyping.) If the transition is by the second rule (SEQUENCE),

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

then by induction $\gamma \vdash c_1' : \tau$ *cmd*, and so by rule (COMPOSE) $\gamma \vdash c_1'; c2 : \tau$ *cmd*. If the transition is by the first rule (SEQUENCE), the argument is simpler.

If c is of the form **while** e **do** c_1, then τ must be L, and we must have $\gamma \vdash c_1 : L$ *cmd*, and so $\gamma \vdash c_1$; **while** e **do** $c_1 : L$ *cmd*.

The case of **if** e **then** c_1 **else** c_2 is similar. □

We also need a lemma about the execution of a sequential composition:

Lemma 5.4 *If $(c_1, \mu) \xrightarrow{s}^k (c_1', \mu')$, then $(c_1; c_2, \mu) \xrightarrow{s}^k (c_1'; c_2, \mu')$. If $(c_1, \mu) \xrightarrow{s}^k \mu'$, then $(c_1; c_2, \mu) \xrightarrow{s}^k (c_2, \mu')$.*

Proof. By induction on k. □

Definition 5.1 *Given an identifier typing γ, we say that memories μ and ν are equivalent, written $\mu \sim_\gamma \nu$, if μ, ν, and γ have the same domain and μ and ν agree on all L identifiers.*

We also say that two commands are equivalent if this can be shown from the following three rules:

1. *If $c = c'$, then $c \sim_\gamma c'$.*
2. *If c and d have type H cmd, then $c \sim_\gamma d$.*

$$\text{(IDENT)}\quad \frac{\gamma(x) = \rho}{\gamma \vdash x : \rho}$$

$$\text{(INT)}\quad \gamma \vdash n : \tau$$

$$\text{(R-VAL)}\quad \frac{\gamma \vdash e : \tau\ var}{\gamma \vdash e : \tau}$$

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

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

$$\text{(COMPOSE)}\quad \frac{\gamma \vdash c_1 : \tau\ cmd,\ \ \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash c_1; c_2 : \tau\ cmd}$$

$$\text{(IF)}\quad \frac{\gamma \vdash e : \tau,\ \ \gamma \vdash c_1 : \tau\ cmd,\ \ \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd}$$

$$\text{(WHILE)}\quad \frac{\gamma \vdash e : L,\ \ \gamma \vdash c : L\ cmd}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : L\ cmd}$$

$$\text{(BASE)}\quad L \subseteq H$$

$$\text{(REFLEX)}\quad \rho \subseteq \rho$$

$$\text{(CMD}^{-}\text{)}\quad \frac{\tau_1 \subseteq \tau_2}{\tau_2\ cmd \subseteq \tau_1\ cmd}$$

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

Figure 2: Typing and subtyping rules

3. If $c \sim_\gamma c'$ and $d \sim_\gamma d'$, then $c; d \sim_\gamma c'; d'$.

Finally, we extend equivalence to configurations by defining $(c, \mu) \sim_\gamma (d, \nu)$ if $c \sim_\gamma d$ and $\mu \sim_\gamma \nu$.

Why do we need a notion of equivalence on commands? Well, we are trying to show that executing a command twice, from two equivalent memories, leads to equivalent memories. But to prove this property by induction on the number of transitions, it is necessary to deal with the fact that the two executions can proceed quite differently, because conditionals with H guards need not follow the same branches in the two executions. For this reason, we must prove a more general property: roughly speaking, equivalent configurations go to equivalent configurations.

Remark. The need for clause 3 in the above definition can be seen from the following example. Suppose $x : H$, $d : L\ cmd$, and $\mu \sim_\gamma \nu$. If μ and ν disagree about the value of x, then the command (**if** $x = 0$ **then** c_1 **else** c_2); d could go to $(c_1; d, \mu)$ under μ and go to $(c_2; d, \nu)$ under ν. Thus we need $c_1; d \sim_\gamma c_2; d$, but these don't have type $H\ cmd$. **End of Remark.**

Theorem 5.5 (Sequential Noninterference) *Suppose c and d are well typed under γ and $(c, \mu) \sim_\gamma (d, \nu)$. If $(c, \mu) \xrightarrow{s} (c', \mu')$, then there exists (d', ν') such that $(d, \nu) \xrightarrow{s}{}^{*} (d', \nu')$ and $(c', \mu') \sim_\gamma (d', \nu')$. And if $(c, \mu) \xrightarrow{s} \mu'$, then there exists ν' such that $(d, \nu) \xrightarrow{s}{}^{*} \nu'$ and $\mu' \sim_\gamma \nu'$.*

Proof. By induction on the structure of c.

We begin by dealing with the case when c and d both have type $H\ cmd$. In this case, by the Confinement Lemma c does not assign to any variables of type L. Therefore, if $(c, \mu) \xrightarrow{s} (c', \mu')$, then we can let (d', ν') be (d, ν) since $(d, \nu) \xrightarrow{s}{}^{0} (d, \nu)$ and $(c', \mu') \sim_\gamma (d, \nu)$. This is because c' must also have type $H\ cmd$ by the Subject Reduction Lemma. If, instead, the transition is $(c, \mu) \xrightarrow{s} \mu'$, then we can appeal again to the Confinement Lemma to get that neither c nor d assigns to any variables of type L, and that there exists ν' such that $(d, \nu) \xrightarrow{s}{}^{*} \nu'$.[3] Since $\mu' \sim_\gamma \nu'$, we're done.

We now deal with the case when c and d do not both have type $H\ cmd$ by considering in turn the possible forms of c:

Case $x := e$**.** Since c and d do not both have type $H\ cmd$, we must have $c = d$, and therefore c does not have type $H\ cmd$. Hence x must have type $L\ var$ and e must have type L. So, by the Simple Security Lemma, every identifier in e has class L. Therefore, since $\mu \sim_\gamma \nu$, we have $\mu(e) = \nu(e)$ and $\mu[x := \mu(e)] \sim_\gamma \nu[x := \nu(e)]$.

Case $c_1; c_2$**.** Since c and d do not both have type $H\ cmd$, d must have the form $d_1; d_2$, where $c_1 \sim_\gamma d_1$ and $c_2 \sim_\gamma d_2$. So if $(c_1, \mu) \xrightarrow{s} (c_1', \mu')$, then by induction there exists (d_1', ν') such that $(d_1, \nu) \xrightarrow{s}{}^{*} (d_1', \nu')$ and $(c_1', \mu') \sim_\gamma (d_1', \nu')$. So by Lemma 5.4, $(d_1; d_2, \nu) \xrightarrow{s}{}^{*} (d_1'; d_2, \nu')$. And, by clause 3 of the definition of $\sim_\gamma$, we have $(c_1'; c_2, \mu') \sim_\gamma (d_1'; d_2, \nu')$. Similarly, if $(c_1, \mu) \xrightarrow{s} \mu'$, then by induction there exists ν' such that $(d_1, \nu) \xrightarrow{s}{}^{*} \nu'$ and $\mu' \sim_\gamma \nu'$. Again by Lemma 5.4, $(d_1; d_2, \nu) \xrightarrow{s}{}^{*} (d_2, \nu')$.

[3] We remark that the proof would break down here if **while** loops were typed as in [VSI96]. Under those rules, d could contain **while** loops, and hence might not be assured of terminating.

Case if e then c_1 else c_2. Since c and d do not both have type $H\ cmd$, we must have $c = d$, and c does not have type $H\ cmd$. Hence $e : L$. As above, this implies that $\mu(e) = \nu(e)$. So if (**if** e **then** c_1 **else** $c_2, \mu) \xrightarrow{s} (c_1, \mu)$, then $\mu(e) = \nu(e)$ is nonzero, so

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \nu) \xrightarrow{s} (c_1, \nu).$$

And if (**if** e **then** c_1 **else** $c_2, \mu) \xrightarrow{s} (c_2, \mu)$, then

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \nu) \xrightarrow{s} (c_2, \nu).$$

Case while e do c_1. Since **while** loops cannot have type $H\ cmd$, we must have $c = d$, and $e : L$. Again, this implies that $\mu(e) = \nu(e)$. So if (**while** e **do** $c_1, \mu) \xrightarrow{s} \mu$, then (**while** e **do** $c_1, \nu) \xrightarrow{s} \nu$. And if

$$(\textbf{while } e \textbf{ do } c_1, \mu) \xrightarrow{s} (c_1; \textbf{while } e \textbf{ do } c_1, \mu),$$

then

$$(\textbf{while } e \textbf{ do } c_1, \nu) \xrightarrow{s} (c_1; \textbf{while } e \textbf{ do } c_1, \nu).$$

□

Remark. The Sequential Noninterference theorem says that if $(c, \mu) \sim_\gamma (d, \nu)$ and (c, μ) reaches a configuration in one step, then (d, ν) reaches an equivalent configuration in *zero or more* steps. This bound cannot be strengthened. For example, suppose that c is $c_1; c_2$ and d is $d_1; c_2$, where c_1 and d_1 have type $H\ cmd$, but c_2 has type $L\ cmd$. Suppose further that (c_1, μ) goes to μ' in one step, but (d_1, ν) goes to ν' in 50 steps. Then $(c_1; c_2, \mu)$ goes in one step to (c_2, μ'). But $(d_1; c_2, \nu)$ takes 50 steps to get to (c_2, ν'). And we need d_1 to run to completion in order to get the required program equivalence, since c_2 is not equivalent to $d_1'; c_2$ for any d_1', under our definition of $\sim_\gamma$. **End of Remark.**

We now wish to apply the Sequential Noninterference Theorem to establish a Concurrent Noninterference Theorem. We begin with a lemma, which depends crucially on our nondeterministic scheduling, that shows that any execution of a thread can be "lifted" to an execution of the global system:

Lemma 5.6 (Global Execution) *Suppose $O(\alpha) = c$. If $(c, \mu) \xrightarrow{s}{}^{k} (c', \mu')$, then $(O, \mu) \xrightarrow{g}{}^{k} (O[\alpha := c'], \mu')$. And if $(c, \mu) \xrightarrow{s}{}^{k} \mu'$, then $(O, \mu) \xrightarrow{g}{}^{k} (O - \alpha, \mu')$.*

Proof. By induction on k.

If $k = 0$, then $(c, \mu) = (c', \mu')$, so $(O[\alpha := c'], \mu') = (O, \mu)$. Hence $(O, \mu) \xrightarrow{g}{}^{0} (O[\alpha := c'], \mu')$.

For the inductive step, if $(c, \mu) \xrightarrow{s}{}^{k+1} (c', \mu')$, then there exists (c'', μ'') such that $(c, \mu) \xrightarrow{s}{}^{k} (c'', \mu'') \xrightarrow{s} (c', \mu')$. By induction, $(O, \mu) \xrightarrow{g}{}^{k} (O[\alpha := c''], \mu'')$. And by the second rule (GLOBAL),

$$(O[\alpha := c''], \mu'') \xrightarrow{g} (O[\alpha := c''][\alpha := c'], \mu').$$

Since $O[\alpha := c''][\alpha := c'] = O[\alpha := c']$, it follows that $(O, \mu) \xrightarrow{g}{}^{k+1} (O[\alpha := c'], \mu')$.

The case where $(c, \mu) \xrightarrow{s}{}^{k+1} \mu'$ is similar. □

Remark. This lemma remains true if we assume that scheduling is *fair*, since we are dealing only with finite computations here. But if we assume *bounded fairness*, so that there is a fixed bound b on the number of transitions a thread can make before another thread gets a turn, then the lemma holds only for $k \leq b$. **End of Remark.**

Definition 5.2 *$O_1 \sim_\gamma O_2$ if $dom(O_1) = dom(O_2)$ and for all $\alpha \in dom(O_1)$, $O_1(\alpha) \sim_\gamma O_2(\alpha)$. Also, $(O_1, \mu) \sim_\gamma (O_2, \nu)$ if $O_1 \sim_\gamma O_2$ and $\mu \sim_\gamma \nu$.*

Corollary 5.7 (Concurrent Noninterference) *Suppose O_1 and O_2 are well typed under γ and $(O_1, \mu) \sim_\gamma (O_2, \nu)$. If $(O_1, \mu) \xrightarrow{g} (O_1', \mu')$, then there exists (O_2', ν') such that $(O_2, \nu) \xrightarrow{g}{}^{*} (O_2', \nu')$ and $(O_1', \mu') \sim_\gamma (O_2', \nu')$.*

Proof. If $(O_1, \mu) \xrightarrow{g} (O_1', \mu')$, then (by inspection of the rules (GLOBAL)) there exists α such that $O_1(\alpha) = c$ and either

1. $(c, \mu) \xrightarrow{s} (c', \mu')$ and $O_1' = O_1[\alpha := c']$, or else
2. $(c, \mu) \xrightarrow{s} \mu'$ and $O_1' = O_1 - \alpha$.

Let $d = O_2(\alpha)$. Then $(c, \mu) \sim_\gamma (d, \nu)$, since $(O_1, \mu) \sim_\gamma (O_2, \nu)$. So, in the first case, by the Sequential Noninterference Theorem there exists (d', ν') such that $(d, \nu) \xrightarrow{s}{}^{*} (d', \nu')$ and $(c', \mu') \sim_\gamma (d', \nu')$. Hence, by the Global Execution Lemma, $(O_2, \nu) \xrightarrow{g}{}^{*} (O_2[\alpha := d'], \nu')$. Finally,

$$(O_1[\alpha := c'], \mu') \sim_\gamma (O_2[\alpha := d'], \nu').$$

The second case is similar. □

Let $\{\}$ denote the empty object map. We can give a final corollary:

Corollary 5.8 *Suppose that O is well typed under γ and $\mu \sim_\gamma \nu$. If $(O, \mu) \xrightarrow{g}{}^{*} (\{\}, \mu')$, then there exists ν' such that $(O, \nu) \xrightarrow{g}{}^{*} (\{\}, \nu')$ and $\mu' \sim_\gamma \nu'$.*

Proof. By an easy generalization of the Concurrent Noninterference Corollary, it follows that there exists (O', ν') such that $(O, \nu) \xrightarrow{g}{}^{*} (O', \nu')$ and $(\{\}, \mu') \sim_\gamma (O', \nu')$. Then $O' = \{\}$ by definition of $\sim_\gamma$. □

6 Adding a Clock to the Language

Many languages include a system clock that can be read by a running program; for instance, Java includes a function `System.currentTimeMillis()`. One would expect that such a clock would have implications for secure information flow, since it makes timing information observable internally. In this section, we explore this issue.

To include a clock, we use a special identifier t, initially 0, which tells the number of transition steps that have been made in the current program execution. We can make t read-only by giving it either type L or H, rather than $L\ var$ or $H\ var$. We must modify the semantics of some commands to update t appropriately; the modified transitions are given in Figure 3.

Now, if we assume $t : L$, then we clearly run into trouble with the noninterference property. For example, suppose that $x : H$, $y : L$, and $c : H\ cmd$ is a command that takes 20 steps to finish. Consider the following program, which has just one thread:

$$\text{(UPDATE)}\quad \frac{x \in dom(\mu)}{(x := e, \mu) \xrightarrow{s} \mu[x := \mu(e), t := \mu(t) + 1]}$$

$$\text{(BRANCH)}\quad \frac{\mu(e) \text{ nonzero}}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{s} (c_1, \mu[t := \mu(t) + 1])}$$

$$\frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{s} (c_2, \mu[t := \mu(t) + 1])}$$

$$\text{(LOOP)}\quad \frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{s} \mu[t := \mu(t) + 1]}$$

$$\frac{\mu(e) \text{ nonzero}}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{s} (c; \textbf{while } e \textbf{ do } c, \mu[t := \mu(t) + 1])}$$

Figure 3: Modified transitions to maintain a clock t

```
if x = 1 then c;
if t < 10 then y := 0 else y := 1
```

Assuming that x is either 0 or 1 initially, this program copies x into y. By checking the value of t, the program can determine whether c was executed or not, which in turn tells whether $x = 1$ or not.

But if we assume, instead, that $t : H$, then the above program is ill-typed, because the branches of the second conditional do not have type $H\ cmd$. And, indeed, if $t : H$, then the proof of Theorem 5.5 still goes through, and so the noninterference property is preserved.

7 Other Scheduling Policies

The semantics of concurrency given by rule (GLOBAL) is purely nondeterministic; the rule simply says that at every step, *any* thread can be selected to run for a step. It is important to understand that the noninterference results of the last section depend crucially on this nondeterminism. For example, Corollary 5.8 says that if $\mu \sim_\gamma \nu$ and there is some way of scheduling the threads of (O, μ) that leads to termination, then there is some way of scheduling the threads of (O, ν) that leads to an equivalent result. But the two schedules can be very different! In particular, even if the first schedule treats all threads equally (in the sense that it gives each thread a roughly equal amount of CPU time), the second schedule might have to greatly favor one thread over the others. Therefore, if we impose additional constraints on the way scheduling is done, we may falsify the Global Execution Lemma and hence the noninterference property.[4]

For example, suppose that scheduling is done by round-robin time slicing, with a time-slice of b steps. Let $x : H$ and $y : L$ and consider the following two threads:

- Thread α:

```
if x = 1 then c;
y := 1
```

- Thread β:

```
y := 0
```

[4]Thus our situation is quite different from the usual one in which one proves the correctness of a concurrent program with respect to a nondeterministic scheduler. There, one can immediately say that the program is correct with respect to *any* scheduler that one might care to implement, because any schedule produced by an implemented scheduler could have been produced by the nondeterministic scheduler.

Suppose further that $c : H\ cmd$ is a command that takes longer than b steps to finish. If $\mu(x) = 0$, $\nu(x) = 1$, and $\mu(y) = \nu(y) = 0$, then $\mu \sim_\gamma \nu$. And from μ, we can terminate in a state where $y = 0$, but from ν we cannot; from ν, we can only terminate in a state where $y = 1$.

In terms of our proofs, here's what is going on:

$$((\textbf{if } x = 1 \textbf{ then } c); y := 1, \mu) \xrightarrow{s} (y := 1, \mu),$$

so by the Sequential Noninterference theorem there exists ν' such that

$$((\textbf{if } x = 1 \textbf{ then } c); y := 1, \nu) \xrightarrow{s}{}^{*} (y := 1, \nu'),$$

and $\mu \sim_\gamma \nu'$. But, although $(O, \mu) \xrightarrow{g} (O[\alpha := (y := 1)], \mu)$, it is *not* the case that $(O, \nu) \xrightarrow{g}{}^{*} (O[\alpha := (y := 1)], \nu')$, because the time-slicing scheduler will not let thread α run for such a long time without giving a turn to thread β.

Another approach to scheduling is *probabilistic*. One might attempt to approximate the effect of rule (GLOBAL) by flipping coins at each step to select the next thread to run. While such an implementation is in some ways faithful to rule (GLOBAL), the adoption of a probabilistic semantics makes it possible to create *probabilistic covert channels* [Gra90], which cannot be addressed without refining the notion of noninterference. This point is discussed in more detail in Section 8.

To preserve noninterference in the face of an arbitrary scheduler, it appears necessary to require the guards of conditionals to have type L. If this is done, we can strengthen the Sequential Noninterference Theorem to the following form:

Theorem 7.1 (Lockstep Execution) *Suppose c is well typed under γ and $\mu \sim_\gamma \nu$. If $(c, \mu) \xrightarrow{s} (c', \mu')$, then there exists ν' such that $(c, \nu) \xrightarrow{s} (c', \nu')$ and $\mu' \sim_\gamma \nu'$. And if $(c, \mu) \xrightarrow{s} \mu'$, then there exists ν' such that $(c, \nu) \xrightarrow{s} \nu'$ and $\mu' \sim_\gamma \nu'$.*

This Lockstep Execution result is strong enough to establish Concurrent Noninterference, regardless of how scheduling is done. Anything done under μ can now be exactly mirrored under ν. Also, Lockstep Execution implies that we can add a clock t and even give it type L, since program timing now cannot depend on the values of H variables. Unfortunately, restricting conditionals in this way is likely to be quite burdensome in practice.

On the other hand, it can be useful for guarding against timing attacks. Kocher, for example, describes a timing

attack on RSA modular exponentiation to learn a private key [Koc96]. Such attacks are possible by merely knowing the source code for an algorithm. Typing conditionals as restrictively as **while** loops rejects code susceptible to this kind of attack.

8 A Closer Look at Noninterference

Our noninterference property basically says that the final values of low variables are independent of the initial values of high variables. More precisely, it says that changing the initial values of high variables cannot affect the set of possible final values of low variables. Hence, observing the final values of low variables cannot reveal anything about the initial values of high variables.

But consider the following example, which is given by McLean [McL90]. Let `x` be a high variable whose value is between 1 and 100 and let `y` be a low variable. Consider the following two threads:

- Thread α:

  ```
  y := x
  ```

- Thread β:

  ```
  y := rand(100)
  ```

where `rand(100)` returns a random integer between 1 and 100.

Now, this program satisfies our noninterference property: regardless of the value of `x`, the final value of `y` can be any integer between 1 and 100. But this program doesn't seem to be secure! If we were to run the program repeatedly, we would expect a sequence of final values for `y` something like

$$75, 22, 12, 22, 22, 93, 4, 22, \ldots$$

and we would feel quite confident that (in this case) the value of `x` is 22.

How can this be explained? The answer is that we have implicitly changed the semantics of our language from the purely nondeterministic semantics of rule (GLOBAL) to some kind of *probabilistic semantics*. In a nondeterministic semantics, outcomes are either *possible* or *impossible*, with no further distinction. But in a probabilistic semantics, outcomes occur according to some *probability distribution*, which makes it possible to make probabilistic inferences.

In our example, if we assume that each thread has an equal probability of being scheduled at each step and that `rand(100)` generates all numbers in the range 1 to 100 with equal probability, then we can see that the final value of `y` will be the initial value of `x` with probability 101/200, and will be any other number between 1 and 100 with probability 1/200. Hence we can be confident of correctly guessing the initial value of `x` by running the program repeatedly and picking the most common final value of `y`. To rule out such probabilistic inferences, we would need a more refined notion of noninterference that requires that the *probability distribution* of the final value of `y` be independent of the initial value of `x`. The program would not satisfy such a probabilistic notion of noninterference, because changes to the initial value of `x` do change the distribution of the final value of `y`.

Thus we can see that the appropriate formulation of the noninterference property depends on the kind of language being considered. In all cases, the idea is that the final values of low variables are independent of the initial values of high variables. For a deterministic language, this means that changing the initial values of high variables cannot change the final values of low variables. For a nondeterministic language, as considered in this paper, it means that changing the initial values of high variables cannot change the *set* of possible final values of low variables. And for a probabilistic language, it means that changing the initial values of high variables cannot change the *distribution* of possible final values of low variables.[5]

It can, of course, be argued that a nondeterministic semantics as used in this paper is unrealistic, because any real implementation would display probabilistic behavior. It is perhaps worth remarking that a nondeterministic semantics can be regarded as an abstraction of a probabilistic semantics in which one equates "possible" with "occurs with nonzero probability". For instance, an implementation of rule (GLOBAL) that flips coins at each step to decide which thread to run has the property that each thread has a nonzero probability of being selected at each step. Indeed, any terminating execution possible under rule (GLOBAL) has a nonzero probability of occurring in the implementation. Therefore, Corollary 5.8 does hold for this implementation. However, one has to be careful with this view of possibility. Though the corollary assures us that, under such an implementation, one can never be *certain* of the initial values of high variables based on observing the final values of low variables, it does not mean that one cannot guess the initial values with high probability.

It is also worth remarking that thread α in the example above is rejected by our type system. This suggests that well-typed programs in our system, if given a probabilistic semantics, might perhaps satisfy some sort of probabilistic noninterference property. But it is easy to see that our type system would not rule out probabilistic timing channels. For example, suppose `x` is a high varible whose value is either 0 or 1, `y` is a low variable, and `c` is a high command that takes a long time to execute. Consider the following two threads:

- Thread α:

  ```
  if x = 1 then (c;c);
  y := 1
  ```

- Thread β:

  ```
  c;
  y := 0
  ```

If thread scheduling works by flipping a coin at each step to decide which thread to run, then with high probability the two threads run at about the same rate. Hence, with high probability the value of `x` ends up being copied into `y`. Extending our type system to deal with a probabilistic language remains an area for future study.

Finally, it is well known that in some cases noninterference is too restrictive. In particular, noninterference cannot accommodate *information downgrading*. For example, information is effectively downgraded when it is encrypted. The

[5] In the security literature, there have been many noninterference-like properties proposed. Noninterference was first proposed by Goguen and Meseguer [GM82] for deterministic systems. Later, McCullough [McC88] proposed Generalized Noninterference and Restrictiveness for nondeterministic systems, and Gray [Gra90, Gra91] proposed P-Restrictiveness and Information Flow Security for probabilistic systems. See also McLean [McL90] for a comparison of some of these properties, and Wittbold and Johnson [WJ90] for an information-theoretic account of possibilistic and probabilistic noninterference.

problem is that ciphertext is sensitive to changes in high cleartext, yet we would often like to treat the ciphertext as low. This is a clear violation of noninterference [McL90].

9 Related Work

Analyzing code for various security properties has a long history. Denning [Den75, Den76, DD77] developed a form of program certification for detecting secure flow violations in code. It was inspired by the work of Bell and LaPadula [BL73], Fenton [Fen73], and Lampson [Lam73], among others. There is also the classic operating systems protection work of Harrison, Ruzzo, and Ullman who showed that the problem of determining whether a program, comprised of simple primitives for updating an access matrix, leaks an access right is undecidable [HRU76]. See also [DDG+76] for an excellent discussion about solvability and complexity issues associated with formal systems for reasoning about program security.

More recently, there is the work of He and Gligor [HG92] who describe ways to eliminate timing channels in the source code of trusted computing bases using an automated tool. Banâtre, Bryce, and Le Métayer [BBLM94] attempt to treat secure information flow in a nondeterministic setting; they give a compile-time technique for detecting flow violations in sequential programs.

Other more recent efforts are more closely related to our work in that they too attempt to characterize some sort of security analysis as a formal system of types. Palsberg and Ørbæk [PØ95] have developed a system to manage trust in the lambda calculus. It is not clear what an appropriate notion of type soundness is for their trust system, given that explicit coercions between trusted and untrusted entities are available in the core calculus. Any suitable notion should speak to security in some way. Abadi [Aba97] has developed a system of typing rules for ensuring secrecy in cryptographic protocols. These protocols are expressed in an extension of the pi calculus called spi. Type soundness is that of testing equivalence between two terms $P\sigma$ and $P\sigma'$, where σ and σ' are substitutions of values for variables and P is a well-typed spi term. In other words, no other spi term, called an observer, can distinguish $P\sigma$ from $P\sigma'$. Heintze and Riecke [HR98] attempt to refine Denning's analysis using more detailed type structure. They also extend their type system for a concurrent language but do not treat type soundness in this case. Finally, Myers and Liskov [ML97] describe a decentralized approach to downgrading information in a secure information flow setting, but its soundness also is not addressed. Some sort of formal justification for downgrading is needed.

10 Conclusion

It is clear that with just ordinary thread implementations, users can exploit seemingly innocuous features like thread priorities and scheduling to easily build reliable covert channels. An off-the-shelf implementation of Java is more than enough here. Furthermore, the bandwidth of such channels is not an issue, for private keys and credit card numbers require little bandwidth. A truly secure programming language demands fundamental changes in language design and an understanding of the relationship between semantics and security.

11 Acknowledgments

This material is based upon activities supported by the National Science Foundation under grants CCR-9612176 and CCR-9612345. We are grateful to Martín Abadi, Paul Attie, Nevin Heintze, John McLean, and Jon Riecke for helpful discussions, and to Scott Smith for shepherding this paper.

References

[Aba97] Martín Abadi. Secrecy by typing in cryptographic protocols. In *Proceedings TACS '97*, September 1997.

[BBLM94] J. Banâtre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *Proceedings 3rd European Symposium on Research in Computer Security*, pages 55–73, Brighton, UK, November 1994. Lecture Notes in Computer Science 875.

[BL73] David Bell and Leonard LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp., Bedford, MA, 1973.

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

[DDG+76] D. Denning, P. Denning, S. Garland, M. Harrison, and W. Ruzzo. Proving protection systems safe. Technical Report CSD TR 209, Purdue University, November 1976.

[Den75] Dorothy Denning. *Secure Information Flow in Computer Systems.* PhD thesis, Purdue University, West Lafayette, IN, May 1975.

[Den76] Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[Fen73] J. Fenton. *Information Protection Systems.* PhD thesis, University of Cambridge, 1973.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[Gra90] James W. Gray, III. Probabilistic interference. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, CA, May 1990.

[Gra91] James W. Gray, III. Toward a mathematical foundation for information flow security. In *Proc. 1991 IEEE Symp. on Research in Security and Privacy*, pages 21–34, Oakland, CA, May 1991.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.

[HG92] J. He and V. Gligor. Formal methods and automated tool for timing-channel identification in TCB source code. In *Proceedings 2nd European Symposium on Research in Computer Security*, pages 57–75, November 1992.

[HR98] Nevin Heintze and Jon Riecke. The SLam Calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, 1998.

[HRU76] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[Jon96] Cliff B. Jones. Some practical problems and their influence on semantics. In *Proceedings of the 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, April 1996. Springer-Verlag.

[Koc96] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings 16th Annual Crypto Conference*, August 1996.

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

[McC88] Daryl McCullough. Noninterference and the Composability of Security Properties. In *Proceedings 1988 IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, CA, 1988.

[McL90] John McLean. Security models and information flow. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, 1990.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[PØ95] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proceedings 1995 Static Analysis Symposium*. Lecture Notes in Computer Science 983, 1995.

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

[VS97b] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. TAPSOFT '97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, April 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for java. Technical Report 546–97, Department of Computer Science, Princeton University, April 1997.

[WJ90] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proceedings 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 144–161, Oakland, CA, May 1990.