

Confinement Properties for Multi-Threaded Programs

Geoffrey Smith¹

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

Dennis Volpano¹

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

Abstract

Given a program that has access to some private information, how can we ensure that it does not improperly leak the information? We formalize the desired security property as a property called noninterference. We discuss versions of noninterference appropriate for multi-threaded programs with probabilistic scheduling and describe rules for ensuring noninterference.

1 Introduction

Ensuring the privacy of information is a major problem today, made both more pressing and more difficult by the enormous growth of the Internet. In this paper, we address one aspect of this problem: given a program P that has access to some private information, how can we prevent P from leaking the information? (This problem was called the *confinement problem* by Lampson [6], who first raised the issue in the early 1970s.) We will focus in particular on the case when P is multi-threaded.

The difficulty of preventing a program P from leaking private information depends greatly on what kinds of observations of P are possible. If we can

¹ This material is based upon activities supported by the National Science Foundation under Agreements Number CCR-9612176 and CCR-9612345.

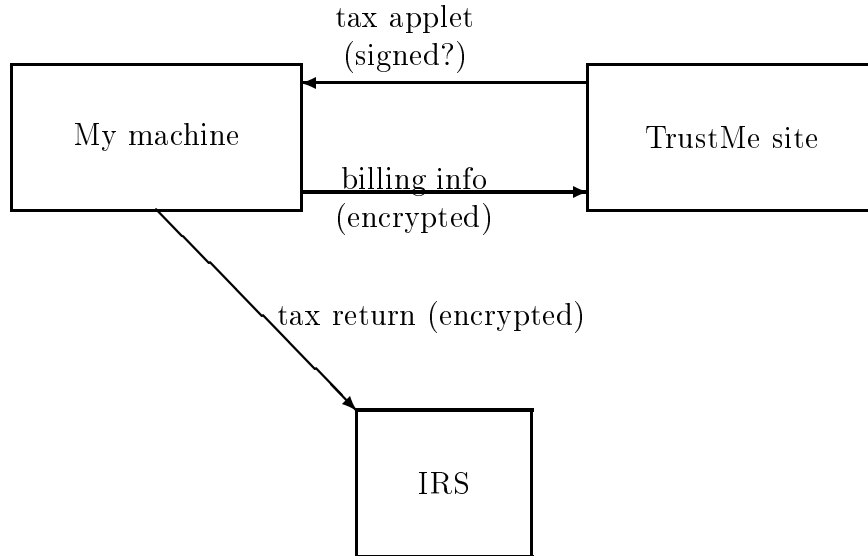


Fig. 1. A Tax Return Applet

make *external observations* of P 's running time, memory usage, and so forth, then preventing leaks becomes very difficult. For example, P could modulate its running time in order to encode the private information. Furthermore, these modulations might depend on low-level implementation details, such as paging and caching behavior. But this means that it is insufficient to prove confinement with respect to an abstract semantics—every implementation detail that affects running time must be addressed in the proof of confinement. For this reason, we will not consider such external observations further.

If, instead, we can only make *internal observations* of P 's behavior, the confinement problem becomes more tractable. Internal observations include the values of program variables, together with any system-provided functions that can be called by P . Of course, if the system provides a real-time clock, then running time is observable internally, and we are no better off than before. But in this case we can design the system with confinement in mind, excluding features (like real-time clocks) that are problematic. This situation is relevant to the case of mobile code, which runs under the control of a host machine that can limit what the code can observe.

When only internal observations are possible, we can formulate the confinement problem as follows [1,2]: if each program variable is classified as L (low, public) or H (high, private), then we wish to ensure that information cannot flow from H variables to L variables.

For example, Figure 1 suggests the behavior of a tax return applet which could be downloaded from a site called TrustMe. The applet runs on my machine, allowing me to complete my tax return. When I finish, the applet sends the completed tax return to the IRS and sends billing information back to TrustMe, using encryption to protect the privacy of these communications.

But how do I know that my private financial information is not somehow encoded in the billing information sent back to TrustMe? If I classify the tax return as H and the billing information as L , then I would like to know that no information can flow from H variables to L variables.

2 Possibilistic Noninterference

Formally, we want programs to satisfy a property called *noninterference* [10], which says that the final values of L variables don't depend on the initial values of H variables. In the case when programs are multi-threaded, and hence nondeterministic, we need a *possibilistic noninterference* property [8], which says that changing the initial values of H variables cannot change the *set* of possible final values of L variables.

Here's a non-example, similar to one in [8]. Suppose x is H , with value 0 or 1, and y is L . Also, assume that t is initially 0. Consider the following program, which consists of two threads:

Thread α : **if** $x = 1$ **then**
 while $t = 0$ **do** *skip*;
 $y := 1$;
 $t := 1$

Thread β : **if** $x = 0$ **then**
 while $t = 0$ **do** *skip*;
 $y := 0$;
 $t := 1$

Note that thread α always assigns 1 to y , and thread β always assigns 0 to y , but the *order* in which these assignments are done depends on the value of x . As a result, with any fair scheduler the value of x is copied to y .

Suppose that we adopt a formal semantics for our multi-threaded language that specifies a *purely nondeterministic scheduler*. Such a scheduler is characterized by the simple rule:

At each step, any thread can be selected to run for a step.

Suppose that we prove that a program P satisfies possibilistic noninterference with respect to this scheduler. Can we conclude that P remains secure if we implement something more deterministic, such as round-robin time slicing?

The answer is no. For suppose that x is H , with value 0 or 1, y is L , and c is a command that doesn't alter x or y , but that takes longer than a time slice. Consider the following program:

Thread α : **if** $x = 1$ **then** $(c; c)$;
 $y := 1$

Thread β : c ;
 $y := 0$

With respect to the purely nondeterministic scheduler, this program satisfies possibilistic noninterference: regardless of the initial value of x , the final value of y can be either 0 or 1. But under round-robin time slicing, the value of x is always copied to y . Thus we see that noninterference is not a safety property—it is not closed under trace subsetting.

3 Probabilistic Noninterference

A purely nondeterministic scheduler is convenient in a formal semantics, but it is unclear how such a scheduler might be implemented; it seems to require an “erratic daemon”.²

We might consider a probabilistic implementation that flips coins to select the thread to run in the next step. But note that this moves us from a *nondeterministic semantics*, in which events are either *possible* or *impossible*, to a *probabilistic semantics*, in which events have a *probability* of occurring. Still, we can say that this gives an implementation of the purely nondeterministic scheduler, if we are willing to equate “possible” with “occurs with nonzero probability”.

But now suppose that x is H , with value between 1 and 100, and y is L . Suppose that $random(100)$ returns a random number between 1 and 100. Consider the following program:

Thread α : $y := x$

Thread β : $y := random(100)$

This program satisfies possibilistic noninterference: regardless of the initial value of x , the final value of y can be any number between 1 and 100. But with a probabilistic semantics, this is not good enough, because the final values of y are not equally likely. In particular, if we can run the program repeatedly, we expect the final values of y to look something like

75, 22, 12, 22, 22, 93, 4, 22, . . .

allowing us to conclude (in this case) that x is probably 22. Thus we see that possibilistic noninterference is not sufficient to prevent *probabilistic* information flows.³ Instead, we now need a *probabilistic noninterference* property, which says that changing the initial values of H variables cannot change the *joint distribution* of possible final values of L variables [9]. In the next section, we develop this idea more formally.

² The term is due to Dijkstra [3].

³ This observation can be credited to McLean [7] and Wittbold and Johnson [11].

4 Multi-Threaded Programs as Markov Chains

We assume that threads are written in a simple imperative language:

$$\begin{array}{l}
 c ::= \textit{skip} \\
 | \quad x := e \\
 | \quad c_1; c_2 \\
 | \quad \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \\
 | \quad \mathbf{while} \ e \ \mathbf{do} \ c
 \end{array}$$

Integers are the only values; we use 0 for false and nonzero for true. We assume that all expressions are pure and total, and that expressions are executed atomically.

Programs are executed with respect to a single shared memory μ , which is a map from identifiers to integers. We extend this to a map from expressions to integers, writing $\mu(e)$ to denote the value of expression e in memory μ .

The semantics of commands is given by a standard transition semantics \longrightarrow on configurations (c, μ) or μ . The rules are given in Figure 2.

A multi-threaded program is modeled by an object map O that maps thread identifiers (α, β, \dots) to commands. The semantics of multi-threaded programs is given via global transitions \xrightarrow{p} on global configurations (O, μ) . The three rules are

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

The first and second rules deal with a nonempty set of threads; the third deals with an empty set of threads. Note that we are assuming a uniform scheduler, that selects each thread in O with equal probability.

With these definitions, a program O executing in memory μ is a *Markov chain* [4]. The states of the Markov chain are all the global configurations reachable from the initial state (O, μ) under \xrightarrow{p} , and the transition matrix T

(NOOP)	$(skip, \mu) \longrightarrow \mu$
(UPDATE)	$\frac{x \in dom(\mu)}{(x := e, \mu) \longrightarrow \mu[x := \mu(e)]}$
(SEQUENCE)	$\frac{(c_1, \mu) \longrightarrow \mu'}{(c_1; c_2, \mu) \longrightarrow (c_2, \mu')}$ $\frac{(c_1, \mu) \longrightarrow (c'_1, \mu')}{(c_1; c_2, \mu) \longrightarrow (c'_1; c_2, \mu')}$
(BRANCH)	$\frac{\mu(e) \neq 0}{(\mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2, \mu) \longrightarrow (c_1, \mu)}$ $\frac{\mu(e) = 0}{(\mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2, \mu) \longrightarrow (c_2, \mu)}$
(LOOP)	$\frac{\mu(e) = 0}{(\mathbf{while} \ e \ \mathbf{do} \ c, \mu) \longrightarrow \mu}$ $\frac{\mu(e) \text{ nonzero}}{(\mathbf{while} \ e \ \mathbf{do} \ c, \mu) \longrightarrow (c; \mathbf{while} \ e \ \mathbf{do} \ c, \mu)}$

Fig. 2. Sequential Transition Semantics

is given by

$$T((O_1, \mu_1), (O_2, \mu_2)) = \begin{cases} p, & \text{if } (O_1, \mu_1) \xrightarrow{p} (O_2, \mu_2) \\ 0, & \text{otherwise} \end{cases}$$

It is now useful to define a *probabilistic state* u to be a (discrete) probability distribution on the set of global configurations [5]. Concretely, u is a row vector with unit sum. With this viewpoint, we can model the execution of O under memory μ as a *deterministic* sequence of probabilistic states:

$$u_0, u_1, u_2, u_3, \dots$$

where u_0 is the distribution that assigns probability 1 to (O, μ) and 0 to all other configurations, and $u_{k+1} = u_k T$. We can now write a very simple expression for the k th probabilistic state: $u_k = u_0 T^k$.

For example, consider the program

$$O = \left\{ \begin{array}{l} \alpha : \mathbf{while} \ l = 0 \ \mathbf{do} \ skip \\ \beta : l := 1 \end{array} \right\}$$

executed in a memory that sets l to 0 initially. There are a total of five reachable states in the Markov chain:

$$q_1 : \left(\left\{ \begin{array}{l} \alpha : \mathbf{while} \ l = 0 \ \mathbf{do} \ skip \\ \beta : (l := 1) \end{array} \right\}, [l := 0] \right)$$

$$q_2 : (\{\alpha : \mathbf{while} \ l = 0 \ \mathbf{do} \ skip\}, [l := 1])$$

$$q_3 : \left(\left\{ \begin{array}{l} \alpha : skip; \mathbf{while} \ l = 0 \ \mathbf{do} \ skip \\ \beta : (l := 1) \end{array} \right\}, [l := 0] \right)$$

$$q_4 : (\{\alpha : skip; \mathbf{while} \ l = 0 \ \mathbf{do} \ skip\}, [l := 1])$$

$$q_5 : (\{\}, [l := 1])$$

The transition matrix T for this program is as follows:

	q_1	q_2	q_3	q_4	q_5
q_1	0	1/2	1/2	0	0
q_2	0	0	0	0	1
q_3	1/2	0	0	1/2	0
q_4	0	1	0	0	0
q_5	0	0	0	0	1

For instance, we get the first row of T by noting that running thread α from state q_1 takes us to state q_3 , and running thread β takes us to state q_2 . Thus, under our uniform scheduling assumption, we go from q_1 either to q_2 or to q_3 , each with probability 1/2.

In terms of probabilistic states, the initial distribution u_0 is $(1\ 0\ 0\ 0\ 0)$. And we can trace the probabilistic states that O passes through:

	q_1	q_2	q_3	q_4	q_5
u_0	1	0	0	0	0
u_0T	0	1/2	1/2	0	0
u_0T^2	1/4	0	0	1/4	1/2
u_0T^3	0	3/8	1/8	0	1/2
u_0T^4	1/16	0	0	1/16	7/8
u_0T^5	0	3/32	1/32	0	7/8
u_0T^6	1/64	0	0	1/64	31/32
\vdots			\vdots		

Of course, O will converge to the probabilistic state $(0\ 0\ 0\ 0\ 1)$.

Now, to formalize the probabilistic noninterference property, we need to define a notion of equivalence on probabilistic states. To this end, we say that probabilistic states u and u' are equivalent, written $u \sim u'$, if they are equal after H variables are projected out. Intuitively, u and u' agree about everything except the values of H variables. For example, if x is H and y is L , then

$$\left\{ \begin{array}{l} (O, [x := 0, y := 0]) : 1/2, \\ (O, [x := 1, y := 0]) : 1/2 \end{array} \right\}$$

is equivalent to

$$\{(O, [x := 2, y := 0]) : 1\},$$

since in both cases the result of projecting out x is

$$\{(O, [y := 0]) : 1\}.$$

Finally, we can give the formal definition of probabilistic noninterference:

Definition 4.1 Program O satisfies probabilistic noninterference if for all probabilistic states u and u' , $u \sim u'$ implies $uT \sim u'T$.

This definition gives us what we want, for suppose that we execute a program O under two memories μ and μ' that agree on the values of L variables. Then

$$\{(O, \mu) : 1\} \sim \{(O, \mu') : 1\}$$

and hence

$$\{(O, \mu) : 1\}T^k \sim \{(O, \mu') : 1\}T^k$$

for all k . That is, the two executions proceed in probabilistic lockstep.

5 Ensuring Probabilistic Noninterference

We can perform a static analysis that ensures that a program O satisfies probabilistic noninterference. The analysis is described formally (as a type system) in [9]; here we give an intuitive presentation as a set of rules. The rules impose constraints on assignments, **while** loops, and **if** statements:

- For an assignment, $y := e$, the rule is that if y is L , then e must contain no H variables.
- For a **while** loop, **while** e **do** c , the rule is that e must contain no H variables.
- For an **if** statement, **if** e **then** c **else** c' , the rule is that if e contains any H variables, then
 - (i) c and c' must contain no assignments to L variables,
 - (ii) c and c' must contain no **while** loops, and
 - (iii) the entire **if** statement must be *protected*, so that it executes atomically.

For the last rule, we introduce a new command, **protect** c , whose semantics is given by

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

That is, if (c, μ) can reach μ' in one or more steps, then $(\mathbf{protect} \ c, \mu)$ can reach μ' in exactly one step.

Applying these rules to the first program of Section 2, we see that the program is illegal, because both threads have **while** loops within the bodies of **if** statements whose guards are H . And for the second program of Section 2 to be legal, the **if** statement of thread α needs to be protected; this will mask the amount of time needed to execute it, thereby eliminating the timing channel. Of course, our rules are necessarily conservative. More experience is needed to determine how burdensome they are in practice.

It can be shown that any program O that satisfies the above rules satisfies probabilistic noninterference. Details can be found in [9]; here we sketch part of the argument.

First, we can show probabilistic noninterference for *point masses*; that is, for distributions in which some configuration has probability 1 and all others have probability 0:

Theorem 5.1 *If O satisfies the above rules and $\iota \sim \iota'$, where ι and ι' are point masses, then $\iota T \sim \iota' T$.*

Then we can extend the result to arbitrary distributions by exploiting the linearity of T :

Lemma 5.2 *If $u_i \sim u'_i$, for all i , then*

$$a_1 u_1 + a_2 u_2 + \dots \sim a_1 u'_1 + a_2 u'_2 + \dots$$

Lemma 5.3 *If $u \sim u'$, then there exist coefficients c_1, c_2, c_3, \dots and point masses $\iota_1, \iota_2, \iota_3, \dots$ and $\iota'_1, \iota'_2, \iota'_3, \dots$ with $\iota_i \sim \iota'_i$ for all i , such that*

$$u = c_1\iota_1 + c_2\iota_2 + c_3\iota_3 + \dots$$

and

$$u' = c_1\iota'_1 + c_2\iota'_2 + c_3\iota'_3 + \dots$$

Corollary 5.4 *If O satisfies the above rules and $u \sim u'$, then $uT \sim u'T$.*

Proof. Since T is a continuous linear transformation,

$$\begin{aligned} uT &= (c_1\iota_1 + c_2\iota_2 + c_3\iota_3 + \dots)T \\ &= c_1(\iota_1T) + c_2(\iota_2T) + c_3(\iota_3T) + \dots \\ &\sim c_1(\iota'_1T) + c_2(\iota'_2T) + c_3(\iota'_3T) + \dots \\ &= (c_1\iota'_1 + c_2\iota'_2 + c_3\iota'_3)T \\ &= u'T \end{aligned}$$

□

6 Conclusion

To develop secure computer systems, it is first necessary to identify the precise security properties of interest. We have presented one such property, probabilistic noninterference, aimed at protecting information privacy and we have described rules sufficient to guarantee it; our hope is that such rules provide a basis for constructing provably-secure systems in practice.

References

- [1] Dorothy Denning, “Secure Information Flow in Computer Systems,” PhD thesis, Purdue University, West Lafayette, IN, May 1975.
- [2] Dorothy Denning and Peter Denning, *Certification of programs for secure information flow*, Communications of the ACM **20** (1977), pp. 504–513.
- [3] Edsger Dijkstra, “A Discipline of Programming,” Prentice Hall, 1976.
- [4] William Feller, “An Introduction to Probability Theory and Its Applications,” Volume I, John Wiley & Sons, Inc., third edition, 1968.
- [5] Dexter Kozen, *Semantics of probabilistic programs*, Journal of Computer and System Sciences **22** (1981), pp. 328–350.
- [6] Butler W. Lampson, *A note on the confinement problem*, Communications of the ACM **16** (1973), pp. 613–615.
- [7] John McLean, *Security models and information flow*, In: *Proceedings 1990 IEEE Symposium on Security and Privacy*, Oakland, CA, 1990, pp. 180–187.

- [8] Geoffrey Smith and Dennis Volpano, *Secure information flow in a multi-threaded imperative language*, In: *Proceedings 25th Symposium on Principles of Programming Languages*, San Diego, CA, January 1998, pp. 355–364.
- [9] Dennis Volpano and Geoffrey Smith, *Probabilistic noninterference in a concurrent language*, *Journal of Computer Security*, 1999, To appear.
- [10] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine, *A sound type system for secure flow analysis*, *Journal of Computer Security* **4** (1996), pp. 167–187.
- [11] J. Todd Wittbold and Dale M. Johnson, *Information flow in nondeterministic systems*, In: *Proceedings 1990 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1990, pp. 144–161.