Probabilistic Noninterference in a Concurrent Language †

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

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

September 22, 1999

Abstract

In previous work [16], we give a type system that guarantees that well-typed multi-threaded programs are possibilistically noninterfering. If thread scheduling is probabilistic, however, then well-typed programs may have probabilistic timing channels. We describe how they can be eliminated without making the type system more restrictive. We show that well-typed concurrent programs are probabilistically noninterfering if every total command with a guard containing high variables executes atomically. The proof uses the notion of a probabilistic state of a computation from Kozen's work in the denotational semantics of probabilistic programs [11].[1]

1 Introduction

This work is motivated by applications of mobile code where programs are downloaded, as needed, and executed on a trusted host. Here a host may have sensitive data that downloaded code may need, and we want assurance that they are not leaked by the code. In some cases, the best approach may simply be to forbid any access to the sensitive data, using some access control mechanism. But often the code will legitimately need to access the data to be useful, and in this case, we must be sure the code does not leak it.

Specifically, this paper is concerned with identifying conditions under which concurrent programs, involving high (private) and low (public) variables, can be proved free of information flows from high variables to low variables. In previous work [16], we developed a type system that ensures that well-typed multi-threaded programs have a possibilistic noninterference property. Possibilistic noninterference asserts that the set of possible final values of low variables is independent of the initial values of high variables. Hence, if we run such a program and observe some final values for its low variables, then we cannot conclude anything about the initial values of its high variables. However, that work relies on a purely nondeterministic thread scheduler, whose implementation requires what

†This is an expanded version of a paper that appeared in the Proceedings of the 11th IEEE Computer Security Foundations Workshop, Rockport, MA, pages 34–43, June 1998.

[1]This material is based upon activities supported by DARPA and by the National Science Foundation under Agreement Nos. CCR-9612176 and CCR-9612345.

Dijkstra calls an "erratic daemon" [3]. More realistically, we would expect a mechanically-implemented scheduler to be *probabilistic*. But with a probabilistic scheduler, a possibilistic noninterference property is no longer sufficient—indeed, it now becomes easy to construct well-typed programs with probabilistic timing channels.

We illustrate this point with an example. Suppose that x is a high variable whose value is either 0 or 1, y is a low variable, and c is a command that does not assign to y. Also assume that c takes many steps to complete. Consider the following multi-threaded program:

- Thread α:

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

- Thread β:

  c;
  $y := 0$

Assuming that thread scheduling is purely nondeterministic, this program satisfies possibilistic noninterference—we can see by inspection that the final value of y can be either 0 or 1, regardless of the initial value of x.

But suppose the two threads are actually scheduled probabilistically, by flipping a coin at each step to decide which thread to run. Then the threads run at roughly the same rate and, as a result, the value of x ends up being copied into y with high probability. That is, a change in the initial value of x changes the *probability distribution* of the final values of y. Hence this program exhibits a probabilistic timing channel.

Note that with c suitably chosen, the program is well typed in the type system of [16], and hence that system cannot guard against such channels. One approach would be to modify the type system so that all guards of conditionals are low. In this case, the program is no longer well typed. But such a restriction is likely too burdensome in practice.

Instead, our strategy is to allow the use of high variables in guards, but to require that the resulting timing variations be masked. We accomplish this by imposing a simple syntactic restriction: every conditional whose guard contains high variables must be executed atomically. This is accomplished by wrapping such a conditional with a new command, called **protect** [15], that guarantees that the conditional will be executed atomically in a multi-threaded environment; the result is that timing variations (based on which branch of the conditional is selected) will not be observable internally.[2] In general, we require that any total guarded command be protected if its guard contains high variables. In the remainder of the paper, we formally establish the soundness of our restriction by proving that protected well-typed programs satisfy a probabilistic noninterference property, which says that the joint probability distribution of final values of low variables is independent of the initial values of high variables.

[2] Later, in Sections 6 and 8, we make some remarks about *external* observations.

2 Syntax and semantics

Threads are commands in the following deterministic imperative language:

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

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 do not have side effects, nor do they contain partial operations like division. The command **for** e **do** c is executed by evaluating e, yielding an integer, and then executing c that many times.

A structural operational semantics for our language is given in Figure 1. Programs are executed with respect to a memory μ, which is a mapping from identifiers to integers. Because of our restrictions on expressions, we know that an expression e is always well defined in a memory μ provided that every free variable of e is in $dom(\mu)$; this will always be the case if e is well typed. Also, we assume for simplicity that expressions are evaluated atomically.[3] 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.

The semantics defines a sequential transition relation $\longrightarrow$ 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 define the reflexive transitive closure $\longrightarrow^*$ in the usual way. First $\kappa \longrightarrow^0 \kappa$, for any configuration κ, and $\kappa \longrightarrow^k \kappa''$, for $k > 0$, if there is a configuration κ' such that $\kappa \longrightarrow^{k-1} \kappa'$ and $\kappa' \longrightarrow \kappa''$. Then $\kappa \longrightarrow^* \kappa'$ if $\kappa \longrightarrow^k \kappa'$ for some $k \geq 0$.

Protected sections have a noninterleaving semantics, expressed by rule ATOMICITY. The rule says that a protected section can execute in one sequential step even though the command being protected may take more than one step to terminate successfully. In effect, this captures the idea of disabling scheduling events (interrupts) while a protected section executes and guarantees that at most one thread will be in a protected section at any time. For this reason, we prohibit **while** loops in protected sections. This eliminates any risk of a thread failing to terminate while in a protected section, causing all other threads to freeze, and also simplifies the semantics. Of course **for** loops can be protected and, in fact, must be protected in some situations as we shall see. We also assume that protected sections are not nested. This is not a practical limitation and it simplifies our proofs.

An interleaving semantics for multi-threaded programs is given by the GLOBAL rules in Figure 1. As in [16], we take a concurrent program to be a set O of commands that run concurrently. The set O is called the thread pool and it does not grow during execution. We represent O as a mapping from thread identifiers (α, β, ...) to commands. We assume that

[3] The noninterference property we prove does not depend on atomicity here unless the time it takes to evaluate an expression depends on the values of high variables.

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

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

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

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

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

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

(ITERATE)
$$\frac{\mu(e) \leq 0}{(\mathbf{for}\ e\ \mathbf{do}\ c, \mu) \longrightarrow \mu}$$

$$\frac{\mu(e) > 0}{(\mathbf{for}\ e\ \mathbf{do}\ c, \mu) \longrightarrow (c; \mathbf{for}\ \mu(e) - 1\ \mathbf{do}\ c, \mu)}$$

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

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

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

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

Figure 1: Sequential and concurrent transition semantics

all threads share a single global memory μ, through which the treads can communicate. A pair (O, μ), consisting of a thread pool and a shared memory, is called a *global configuration*.

The GLOBAL rules let us prove judgments of the form

$$(O, \mu) \stackrel{p}{\Longrightarrow} (O', \mu').$$

This asserts that the probability of going from (O, μ) to (O', μ') is p. The first two GLOBAL rules specify the global transitions that can be made by a nonempty thread pool. A scheduling event occurs after a single sequential step of execution. This coupled with the ATOMICITY rule ensures atomic execution of protected sections. 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'. Note that the rules prescribe a uniform probability distribution for the scheduling of threads. (Actually, we could use any fixed distribution for thread scheduling; we use a uniform distribution simply for simplicity.) The third GLOBAL rule, which deals with an empty thread pool, is introduced to accommodate our concurrent program execution model in which a concurrent program is represented by 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

The types of the system are stratified into data and phrase types:

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

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

The rules of the type system are given in Figure 2. They extend the system of [16] with rules for **protect** and **for**. 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 a *variable typing*, mapping variables to phrase types of the form $\tau\ var$. Note that guards of conditionals and **for** loops may contain high variables, unlike the guards of **while** loops.

The effect of these typing rules is to impose constraints on the various constructs of the language; these constraints can be summarized as follows:

- In an assignment $x := e$, if x is low, then e must contain no high variables.
- The guard of a **while** loop must contain no high variables.
- In a conditional **if** e **then** c **else** c', if the guard e contains any high variables, then the branches c and c' must not contain any **while** loops or assignments to low variables. A similar constraint applies to **for** loops.

Definition 3.1 *We say that p is well typed under γ if $\gamma \vdash p : \rho$ for some ρ. Also, O is well typed under γ if $O(\alpha)$ is well typed under γ for every $\alpha \in dom(O)$.*

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

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

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

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

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

(SKIP) $\gamma \vdash \mathbf{skip} : H \ cmd$

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

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

(FOR) $$\frac{\gamma \vdash e : \tau, \ \ \gamma \vdash c : \tau \ cmd}{\gamma \vdash \mathbf{for} \ e \ \mathbf{do} \ c : \tau \ cmd}$$

(PROTECT) $$\frac{\gamma \vdash c : \tau \ cmd}{\gamma \vdash \mathbf{protect} \ c : \tau \ cmd}$$

(BASE) $L \subseteq H$

(REFLEX) $\rho \subseteq \rho$

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

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

Figure 2: Typing and subtyping rules

1)	$(\{\alpha := \textbf{while}\ l = 0\ \textbf{do skip},\ \ \beta := (l := 1)\},$	$[l := 0])$
2)	$(\{\alpha := \textbf{while}\ l = 0\ \textbf{do skip}\},$	$[l := 1])$
3)	$(\{\alpha := \textbf{skip}; \textbf{while}\ l = 0\ \textbf{do skip},\ \ \beta := (l := 1)\},$	$[l := 0])$
4)	$(\{\alpha := \textbf{skip}; \textbf{while}\ l = 0\ \textbf{do skip}\},$	$[l := 1])$
5)	$(\{\ \},$	$[l := 1])$

Figure 3: States of Markov chain

4 Probabilistic states

Informally, our formulation of probabilistic noninterference is a sort of probabilistic lock step execution statement. Under two memories that may differ on high variables, we want to know that the probability that a concurrent program can reach some global configuration under one of the memories is the same as the probability that it reaches an equivalent configuration under the other.

A concurrent program O executing in a memory μ can be viewed as a discrete Markov chain [4]. The states of the Markov chain are all the global configurations reachable from the initial state (O, μ) under $\overset{p}{\Longrightarrow}$, and the transition matrix T (sometimes called the stochastic matrix) is given by

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

For example, consider the following program:

$$O = \{\alpha := \textbf{while}\ l = 0\ \textbf{do skip},\ \ \beta := (l := 1)\}$$

Starting with memory $[l := 0]$, the program can get into at most five different configurations, and so its Markov chain has five states, given in Figure 3. For instance, starting in state 1 we might run thread α for a step, taking us to state 3. (This follows from the second GLOBAL rule and the second LOOP rule of Figure 1.) Alternatively, from state 1 we might run thread β for a step, taking us to state 2. (This follows from the first GLOBAL rule and the UPDATE rule of Figure 1.) Furthermore, the GLOBAL rules specify that the probability of each of these transitions is 1/2 since there are two threads in the thread pool. No other transitions are possible from state 1.

In this way, we can determine the probability of going from each of the five states to any other state. These probabilities are collected in the transition matrix T given in Figure 4. Note that from state 5 we go to state 5 with probability 1 because of the third GLOBAL rule of Figure 1. Since no other state is reachable from state 5, it is called an *absorbing state* [4].

The set of Markov states may be countably infinite—a simple example is a nonterminating loop that increments a variable. In this case, the transition matrix is also countably infinite. In general, if T is a transition matrix and $T((O, \mu), (O', \mu')) > 0$, for some global configurations (O, μ) and (O', μ'), then either O is nonempty and $T((O, \mu), (O', \mu')) = 1/|O|$, or else O and O' are empty, $\mu = \mu'$, and $T((O, \mu), (O', \mu')) = 1$.

	1	2	3	4	5
1	0	1/2	1/2	0	0
2	0	0	0	0	1
3	1/2	0	0	1/2	0
4	0	1	0	0	0
5	0	0	0	0	1

Figure 4: Transition matrix

Kozen gives a denotational semantics of probabilistic programs whereby a program denotes a mapping from one probability distribution to another [11]. The idea of transforming distributions is also useful in an operational setting such as ours. Using the Markov chain, we model the execution of a concurrent program deterministically as a sequence of *probabilistic states*:

Definition 4.1 *A probabilistic state is a probability measure on the set of global configurations.*

A probabilistic state can be represented as a row vector whose components must sum to 1. So if T is a transition matrix and s is a probabilistic state, then the next probabilistic state in the sequence of such states modeling a concurrent computation is simply the vector-matrix product sT. For instance, the initial probabilistic state for the program O in our preceding example is $(1\ \ 0\ \ 0\ \ 0\ \ 0)$. It indicates that the Markov chain begins in state 1 with certainty. The next state is given by taking the product of this state with the transition matrix of Figure 4, giving $(0\ \ 1/2\ \ 1/2\ \ 0\ \ 0)$. This state indicates the Markov chain can be in states 2 and 3, each with a probability of 1/2. Multiplying this vector by T, we get the third probabilistic state, $(1/4\ \ 0\ \ 0\ \ 1/4\ \ 1/2)$; we can determine the complete execution in this way. The first five probabilistic states in the sequence are depicted in Figure 5. The fifth probabilistic state tells us that the probability that O terminates under memory $[l := 0]$ in at most four steps is 7/8.

We remark that $(O, [l := 0])$ is an example of a concurrent program that is probabilistically total, since it halts with probability 1. But it is not nondeterministically total, because it has an infinite computation path.

Note that although there may be infinitely many states in the Markov chains corresponding to our programs, the probabilistic states that arise in our program executions will assign nonzero probability to only finitely many of them. This is because we begin execution in a single global configuration (O, μ), and we branch by at most a factor of k at each step, where k is the number of threads in O. If, however, we were to extend our language with a random number generator, which returns an arbitrary integer with respect to some probability distribution, then we would have to consider probabilistic states that give nonzero probabilities to an infinite number of global configurations.

With probabilistic states, we can now see how probability distributions can be sensitive to initial values of high variables, even for programs that have types in the system of Figure 2. Consider the example from Section 1, where c is instantiated to **skip**:

$$\left\{ \begin{array}{l} \alpha := (\textbf{if } x = 1 \textbf{ then skip}; \textbf{skip});\ y := 1, \\ \beta := (\textbf{skip};\ y := 0) \end{array} \right\}$$

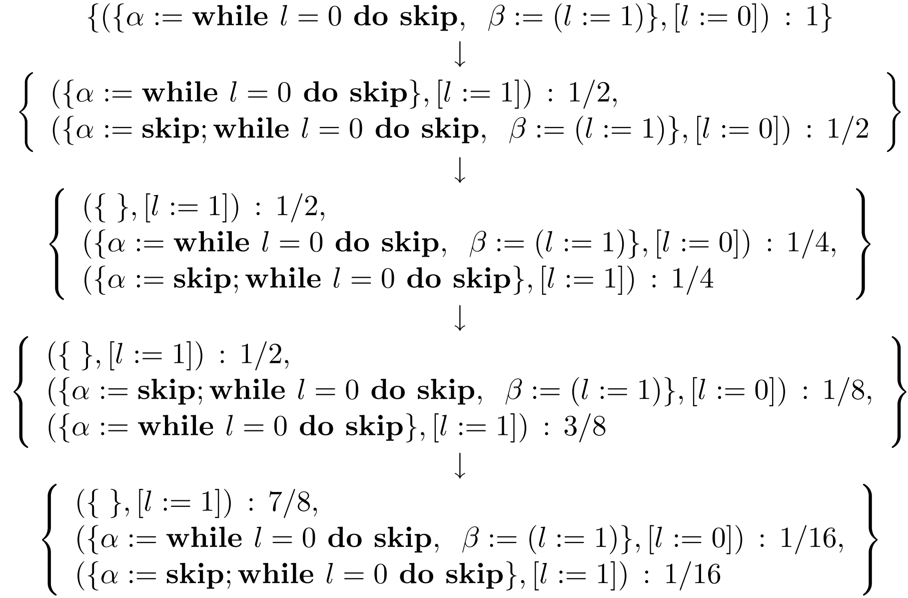

Figure 5: A probabilistic state sequence

Each thread is well typed. We give two sequences of state transitions, assuming the obvious transition semantics for **if** e **then** c. One begins with x equal to 0 (Figure 6) and the other with x equal to 1 (Figure 7). Notice the change in distribution for the final values of y when the initial value of the high variable x changes. For instance, the probability that y has final value 1 when x equals 1 is 13/16, and falls to 1/2 when x equals 0. What is going on here is that the initial value of x affects the amount of time required to execute the conditional; this in turn affects the likely order in which the two assignments to y are executed. Now suppose that we protect the conditional in this example. Then the conditional (in effect) executes in one step, regardless of the value of x, and so the sequence of transitions for $x = 0$ is equivalent, state by state, to the sequence of transitions for $x = 1$ (Figures 8 and 9).

5 Probabilistic noninterference

Now we establish the main result, that a well-typed concurrent program is probabilistically noninterfering if every total command with a guard containing a high variable executes atomically. But first we need some properties of our deterministic thread language. The key property needed in the noninterference proof is a lockstep execution property for well-typed threads executed under equivalent memories (Lemma 5.7). This property in turn depends on other properties of well-typed threads in our system, specifically, Simple Security, Confinement and Mutual Termination. Together these properties assert that well-typed threads respect the privacy of high variables in the deterministic language.

First we need a notion of memory equivalence which basically requires agreement on

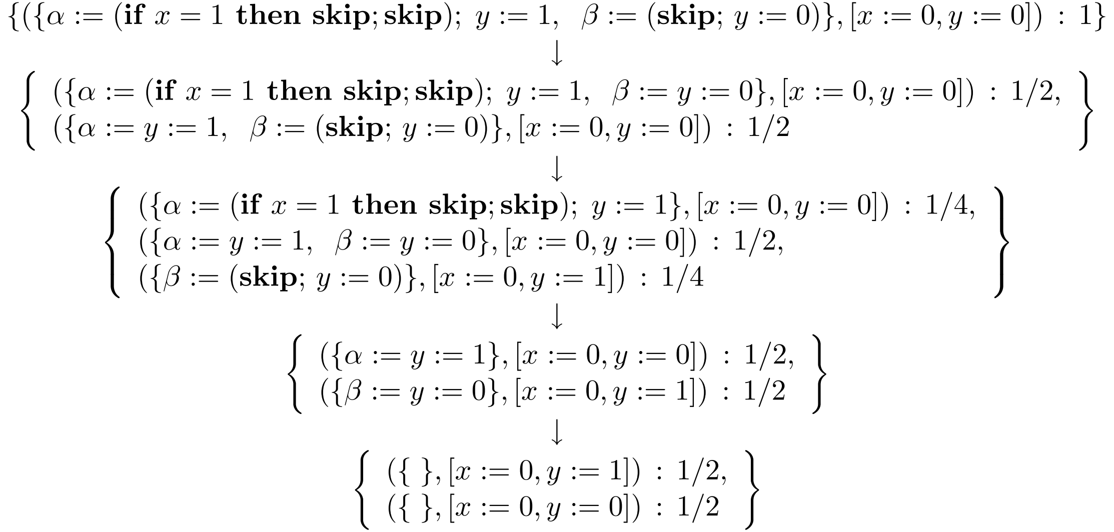

Figure 6: Probabilistic state sequence when $x = 0$

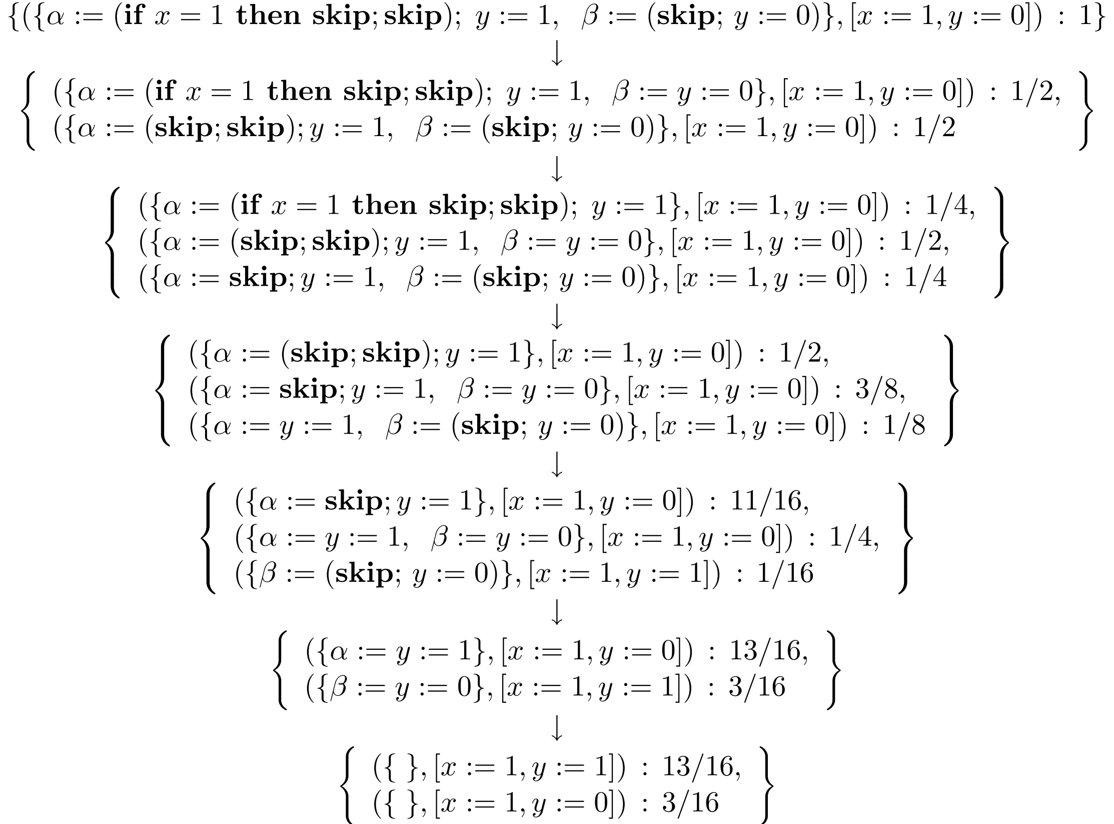

Figure 7: Probabilistic state sequence when $x = 1$

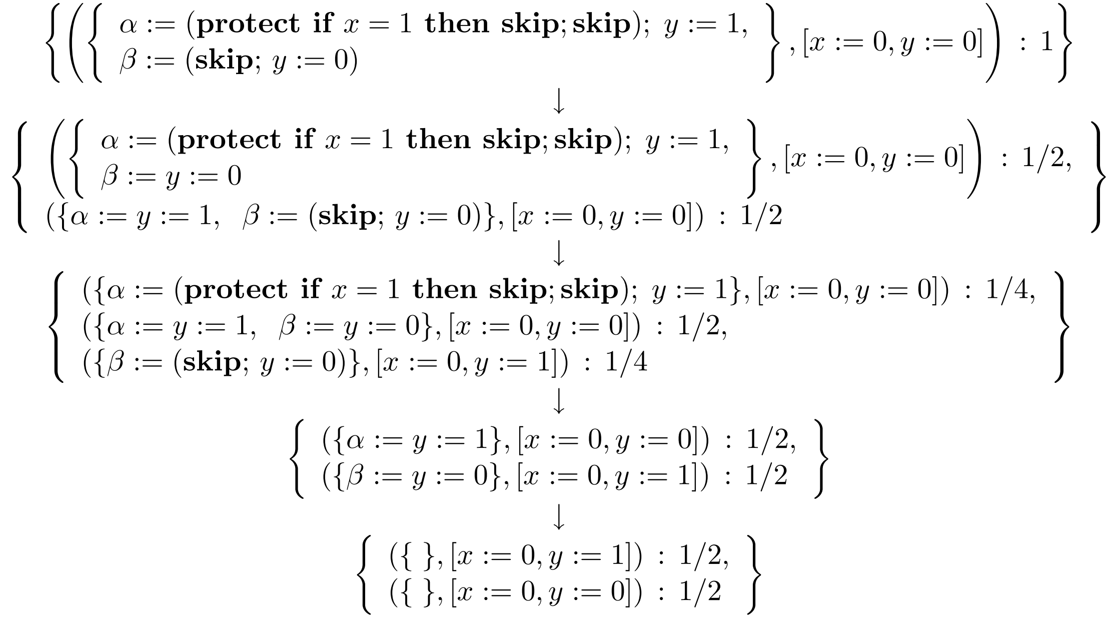

Figure 8: Probabilistic state sequence when $x = 0$

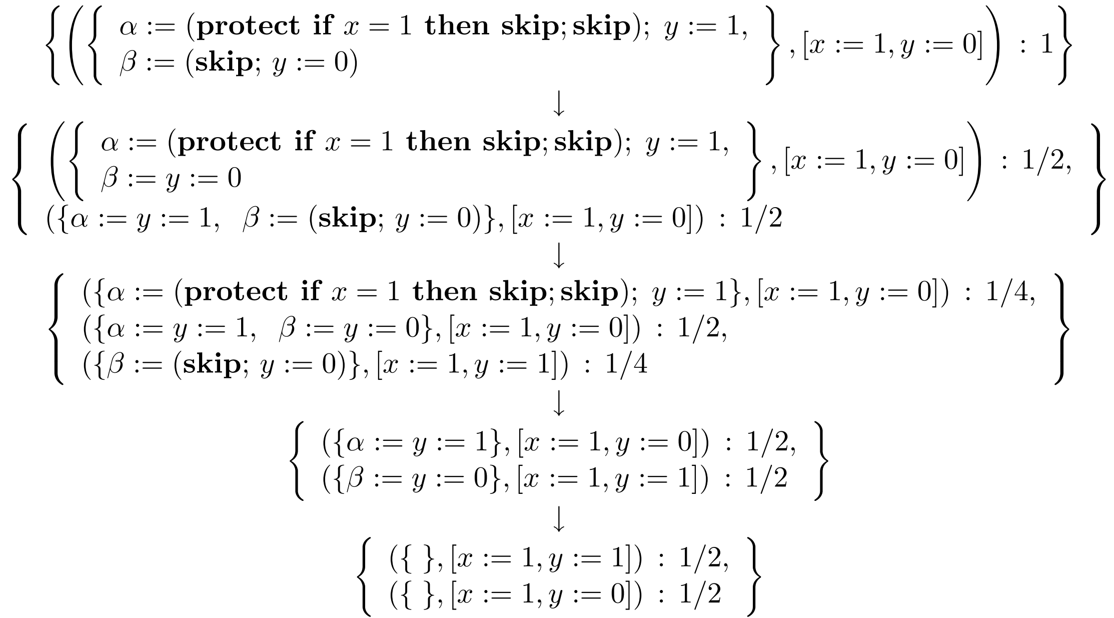

Figure 9: Probabilistic state sequence when $x = 1$

contents of low variables:

Definition 5.1 *Memories μ and ν are equivalent with respect to variable typing γ, written $\mu \sim_\gamma \nu$, if $dom(\mu) = dom(\nu) = dom(\gamma)$ and $\mu(x) = \nu(x)$ for all x such that $\gamma(x) = L\ var$.*

The proofs of Simple Security and Confinement (Lemmas 5.1 and 5.5) are complicated somewhat by subtyping. We shall assume, without loss of generality, that all typing derivations end with a single (perhaps trivial) application of the subsumption rule SUBTYPE.

Lemma 5.1 (Simple Security) *If $\gamma \vdash e : L$ then $\gamma(x) = L\ var$ for every variable x in e.*

Proof. By induction on the structure of e. Since $H \not\subseteq L$, the derivation of $\gamma \vdash e : L$ ends with a trivial application of rule SUBTYPE.

1. Case x. By rule R-VAL, we have $\gamma(x) = L\ var$.
2. Case n. The result holds vacuously.
3. Case $e_1 + e_2$. By rule SUM, we have $\gamma \vdash e_1 : L$ and $\gamma \vdash e_2 : L$. By induction, $\gamma(x) = L\ var$ for every variable x in e_1 and in e_2. The remaining binary operators are handled similarly.

□

Next we consider Confinement. Its proof depends on the next three lemmas. The first two treat the behavior of sequential composition,[4] and the third is a lemma about the termination of **for** loops.

Lemma 5.2 *If $(c_1; c_2, \mu) \longrightarrow^j \mu'$, then there exist k and μ'' such that $0 < k < j$, $(c_1, \mu) \longrightarrow^k \mu''$, and $(c_2, \mu'') \longrightarrow^{j-k} \mu'$.*

Proof. By induction on j. If the derivation begins with an application of the first SEQUENCE rule, then there exists μ'' such that $(c_1, \mu) \longrightarrow \mu''$ and $(c_1; c_2, \mu) \longrightarrow (c_2, \mu'') \longrightarrow^{j-1} \mu'$. So we can let $k = 1$. And, since $j - 1 \geq 1$, we have $k < j$.

If the derivation begins with an application of the second SEQUENCE rule, then there exists c_1' and μ_1 such that $(c_1, \mu) \longrightarrow (c_1', \mu_1)$ and $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu_1) \longrightarrow^{j-1} \mu'$. By induction, there exists k and μ'' such that $0 < k < j - 1$, $(c_1', \mu_1) \longrightarrow^k \mu''$, and $(c_2, \mu'') \longrightarrow^{j-1-k} \mu'$. Hence $(c_1, \mu) \longrightarrow^{k+1} \mu''$ and $(c_2, \mu'') \longrightarrow^{j-(k+1)} \mu'$. And $0 < k+1 < j$. □

Lemma 5.3 *If $(c_1, \mu) \longrightarrow^j \mu'$ and $(c_2, \mu') \longrightarrow^k \mu''$, then $(c_1; c_2, \mu) \longrightarrow^{j+k} \mu''$.*

[4]Lemmas 5.2 and 5.3 are necessitated by our use of a small-step transition semantics for threads. They would be unnecessary in an operational semantics that talks about complete evaluations as in, say, a natural semantics. But a natural semantics does not meet our needs, because we model concurrent program execution as a sequence of probabilistic states where configurations represent intermediate steps of an execution.

Proof. By induction on j. If $j = 1$ then by the first SEQUENCE rule, $(c_1; c_2, \mu) \longrightarrow (c_2, \mu') \longrightarrow^k \mu''$. Hence $(c_1; c_2, \mu) \longrightarrow^{1+k} \mu''$.

If $j > 1$ then there exist c_1' and μ_1 such that $(c_1, \mu) \longrightarrow (c_1', \mu_1) \longrightarrow^{j-1} \mu'$. By induction, $(c_1'; c_2, \mu_1) \longrightarrow^{j-1+k} \mu''$. And, by the second SEQUENCE rule, $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu_1)$. Hence $(c_1; c_2, \mu) \longrightarrow^{j+k} \mu''$. □

Definition 5.2 *A command c is total under γ if for all μ such that $dom(\mu) = dom(\gamma)$, there exists μ' such that $(c, \mu) \longrightarrow^* \mu'$.*

Lemma 5.4 *Suppose $\mu(e)$ is defined and $dom(\mu) = dom(\gamma)$. If c is total under γ then there exists μ' such that $(\textbf{for } e \textbf{ do } c, \mu) \longrightarrow^* \mu'$.*

Proof. Induction on $\mu(e)$. If $\mu(e) \leq 0$ then $(\textbf{for } e \textbf{ do } c, \mu) \longrightarrow \mu$ so let $\mu' = \mu$. Now suppose $\mu(e) > 0$. Since c is total under γ and $dom(\mu) = dom(\gamma)$, there exists μ'' such that $(c, \mu) \longrightarrow^* \mu''$. Further, $dom(\mu'') = dom(\mu) = dom(\gamma)$ and $\mu(\mu(e) - 1)$ is trivially defined since $\mu(e) - 1$ is an integer. So by induction, there exists μ' such that $(\textbf{for } \mu(e) - 1 \textbf{ do } c, \mu'') \longrightarrow^* \mu'$. Then $(c; \textbf{for } \mu(e) - 1 \textbf{ do } c, \mu) \longrightarrow^* \mu'$ by Lemma 5.3. And $(\textbf{for } e \textbf{ do } c, \mu) \longrightarrow (c; \textbf{for } \mu(e) - 1 \textbf{ do } c, \mu)$ by the second ITERATE rule since $\mu(e) > 0$. Hence $(\textbf{for } e \textbf{ do } c, \mu) \longrightarrow^* \mu'$. □

Lemma 5.5 (Confinement) *If $\gamma \vdash c : H\ cmd$, then $\gamma(x) = H\ var$, for every variable x assigned to in c, and c is total under γ.*

Proof. By induction on the structure of c. Since $L\ cmd \not\subseteq H\ cmd$, the derivation of $\gamma \vdash c : H\ cmd$ ends with a trivial application of rule SUBTYPE.

1. Case $x := e$. By rule ASSIGN, $\gamma(x) = H\ var$. If $dom(\mu) = dom(\gamma)$ then $x \in dom(\mu)$, and $\mu(e)$ is defined since e is well typed under γ by rule ASSIGN. So by rule UPDATE, we have $(x := e, \mu) \longrightarrow \mu[x := \mu(e)]$.

2. Case **skip**. The result follows immediately from rule NO-OP.

3. Case $c_1; c_2$. By rule COMPOSE, we have $\gamma \vdash c_1 : H\ cmd$ and $\gamma \vdash c_2 : H\ cmd$. By induction, we have $\gamma(x) = H\ var$ for every variable x assigned to in c_1 and in c_2, and c_1 and c_2 are total under γ. So if $dom(\mu) = dom(\gamma)$ then there exists μ'' such that $(c_1, \mu) \longrightarrow^* \mu''$. Now $dom(\mu'') = dom(\mu)$, so there exists μ' such that $(c_2, \mu'') \longrightarrow^* \mu'$. Hence $(c_1; c_2, \mu) \longrightarrow^* \mu'$ by Lemma 5.3.

4. Case **if** e **then** c_1 **else** c_2. By rule IF, we have $\gamma \vdash c_1 : H\ cmd$ and $\gamma \vdash c_2 : H\ cmd$. By induction, we have $\gamma(x) = H\ var$ for every variable x assigned to in c_1 and in c_2, and c_1 and c_2 are total under γ. If $dom(\mu) = dom(\gamma)$ then $\mu(e)$ is defined since e is well typed under γ. If $\mu(e) \neq 0$ then $(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu)$ by rule BRANCH. And since c_1 is total under γ, there exists μ' such that $(c_1, \mu) \longrightarrow^* \mu'$. So $(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow^* \mu'$. The case when $\mu(e) = 0$ is similar.

5. Case **for** e **do** c_1. By rule FOR, we have $\gamma \vdash c_1 : H\ cmd$. By induction, $\gamma(x) = H\ var$ for every variable x assigned to in c_1, and c_1 is total under γ. If $dom(\mu) = dom(\gamma)$ then $\mu(e)$ is defined since e is well typed under γ. Then there exists μ' such that $(\textbf{for } e \textbf{ do } c_1, \mu) \longrightarrow^* \mu'$ by Lemma 5.4.

6. Case **protect** c_1. By rule PROTECT, we have $\gamma \vdash c_1 : H\ cmd$. By induction, $\gamma(x) = H\ var$ for every variable x assigned to in c_1, and c_1 is total under γ. So if $dom(\mu) = dom(\gamma)$ then there exists μ' such that $(c_1, \mu) \longrightarrow^* \mu'$. Hence $(\textbf{protect}\ c_1, \mu) \longrightarrow \mu'$ by rule ATOMICITY.

□

The Confinement Lemma says that any thread that can be given type $H\ cmd$ under a typing γ, terminates successfully when run in a memory μ having the same domain as γ and does not change the contents of any low variables of μ.

Now we establish the Mutual Termination property. It states that it is impossible for a well-typed thread to terminate under one memory and run forever under an equivalent memory.

Lemma 5.6 (Mutual Termination) *Suppose c is well typed under γ and protect free, and that $\mu \sim_\gamma \nu$. If $(c, \mu) \longrightarrow^* \mu'$ then there is a ν' such that $(c, \nu) \longrightarrow^* \nu'$ and $\mu' \sim_\gamma \nu'$.*

Proof. By induction on the length of the derivation of $(c, \mu) \longrightarrow^* \mu'$. We consider the different forms of c:

1. Case $x := e$. Since c is well typed, we have $x \in dom(\gamma)$. Since $dom(\mu) = dom(\nu) = dom(\gamma)$, we have $(c, \mu) \longrightarrow \mu[x := \mu(e)]$ and $(c, \nu) \longrightarrow \nu[x := \nu(e)]$. If $\gamma(x) = L\ var$ then by rule ASSIGN, we have $\gamma \vdash e : L$. So by Simple Security, $\gamma(y) = L\ var$ for every variable y in e. Hence $\mu(e) = \nu(e)$, and so $\mu[x := \mu(e)] \sim_\gamma \nu[x := \nu(e)]$. If, instead, $\gamma(x) = H\ var$ then trivially $\mu[x := \mu(e)] \sim_\gamma \nu[x := \nu(e)]$.

2. Case **skip**. The result follows immediately from rule NO-OP.

3. Case $c_1; c_2$. If $(c_1; c_2, \mu) \longrightarrow^j \mu'$ then by Lemma 5.2 there exist k and μ'' such that $0 < k < j$, $(c_1, \mu) \longrightarrow^k \mu''$ and $(c_2, \mu'') \longrightarrow^{j-k} \mu'$. By induction, there exists ν'' such that $(c_1, \nu) \longrightarrow^* \nu''$ and $\mu'' \sim_\gamma \nu''$. So by induction again, there exists ν' such that $(c_2, \nu'') \longrightarrow^* \nu'$ and $\mu' \sim_\gamma \nu'$. Finally, $(c_1; c_2, \nu) \longrightarrow^* \nu'$ by Lemma 5.3.

4. Case **while** e **do** c_1. Since $\gamma \vdash e : L$, we know by Simple Security that $\mu(e) = \nu(e)$. Suppose $\mu(e) = 0$. Then $(\textbf{while}\ e\ \textbf{do}\ c_1, \mu) \longrightarrow \mu$ and also $(\textbf{while}\ e\ \textbf{do}\ c_1, \nu) \longrightarrow \nu$, since $\mu(e) = \nu(e)$.

   If $\mu(e) \neq 0$ then $(\textbf{while}\ e\ \textbf{do}\ c_1, \mu) \longrightarrow (c_1; \textbf{while}\ e\ \textbf{do}\ c_1, \mu) \longrightarrow^* \mu'$. By induction, there exists ν' such that $(c_1; \textbf{while}\ e\ \textbf{do}\ c_1, \nu) \longrightarrow^* \nu'$ and $\mu' \sim_\gamma \nu'$. And since $\nu(e) \neq 0$, $(\textbf{while}\ e\ \textbf{do}\ c_1, \nu) \longrightarrow (c_1; \textbf{while}\ e\ \textbf{do}\ c_1, \nu) \longrightarrow^* \nu'$.

5. Case **if** e **then** c_1 **else** c_2. If $\gamma \vdash e : L$ then $\mu(e) = \nu(e)$ by Simple Security. If $\mu(e) \neq 0$ then $(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \mu) \longrightarrow (c_1, \mu) \longrightarrow^* \mu'$. By induction, there exists ν' such that $(c_1, \nu) \longrightarrow^* \nu'$ and $\mu' \sim_\gamma \nu'$. Since $\nu(e) \neq 0$, we have $(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \nu) \longrightarrow (c_1, \nu) \longrightarrow^* \nu'$. The case when $\mu(e) = 0$ is similar.

   If instead $\gamma \not\vdash e : L$ then by rule IF, and the fact that c is well typed, we have $\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : H\ cmd$. Then by Confinement, $\mu' \sim_\gamma \mu$ and there exists ν' such that $(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \nu) \longrightarrow^* \nu'$ and $\nu' \sim_\gamma \nu$. So $\mu' \sim_\gamma \nu'$.

6. Case **for** e **do** c_1. If $\gamma \vdash e : L$ then $\mu(e) = \nu(e)$ by Simple Security. So if $\mu(e) \leq 0$ then $(\textbf{for } e \textbf{ do } c_1, \mu) \longrightarrow \mu$ and since $\mu(e) = \nu(e)$, $(\textbf{for } e \textbf{ do } c_1, \nu) \longrightarrow \nu$. If instead $\mu(e) > 0$, then $(\textbf{for } e \textbf{ do } c_1, \mu) \longrightarrow (c_1; \textbf{for } \mu(e) - 1 \textbf{ do } c_1, \mu) \longrightarrow^* \mu'$. By induction, there exists ν' such that $(c_1; \textbf{for } \mu(e) - 1 \textbf{ do } c_1, \nu) \longrightarrow^* \nu'$ and $\mu' \sim_\gamma \nu'$. And since $\mu(e) = \nu(e)$, $(\textbf{for } e \textbf{ do } c_1, \nu) \longrightarrow (c_1; \textbf{for } \nu(e) - 1 \textbf{ do } c_1, \nu) \longrightarrow^* \nu'$.

   If instead $\gamma \not\vdash e : L$ then since c is well typed, we have $\gamma \vdash \textbf{for } e \textbf{ do } c_1 : H\ cmd$ by rule FOR. By Confinement, there exists ν' such that $(\textbf{for } e \textbf{ do } c_1, \nu) \longrightarrow^* \nu'$ and $\nu' \sim_\gamma \nu$. Also, $\mu' \sim_\gamma \mu$ by Confinement, so $\mu' \sim_\gamma \nu'$.

□

Definition 5.3 *A command is protected under γ if every conditional and* **for** *loop is enclosed by a* **protect** *whenever the guard contains a variable x such that $\gamma(x) = H\ var$. Also, O is protected under γ if $O(\alpha)$ is protected under γ for every $\alpha \in dom(O)$.*

Now we can show that if we execute a well-typed, protected command c in two equivalent memories then the two executions proceed in lock step.

Lemma 5.7 (Lock Step Execution) *Suppose c is well typed under γ and protected, and that $\mu \sim_\gamma \nu$. If $(c, \mu) \longrightarrow (c', \mu')$, then there exists ν' such that $(c, \nu) \longrightarrow (c', \nu')$ and $\mu' \sim_\gamma \nu'$. And if $(c, \mu) \longrightarrow \mu'$, then there exists ν' such that $(c, \nu) \longrightarrow \nu'$ and $\mu' \sim_\gamma \nu'$.*

Proof. By induction on the structure of c.

1. Case $x := e$. Since c is well typed, $x \in dom(\gamma)$. Since $dom(\mu) = dom(\nu) = dom(\gamma)$, we have $(c, \mu) \longrightarrow \mu[x := \mu(e)]$ and $(c, \nu) \longrightarrow \nu[x := \nu(e)]$. If $\gamma(x) = L\ var$ then by rule ASSIGN, we have $\gamma \vdash e : L$. So by Simple Security, $\gamma(y) = L\ var$ for every variable y in e. Hence $\mu(e) = \nu(e)$, and so $\mu[x := \mu(e)] \sim_\gamma \nu[x := \nu(e)]$. If, instead, $\gamma(x) = H\ var$ then trivially $\mu[x := \mu(e)] \sim_\gamma \nu[x := \nu(e)]$.

2. Case **skip**. The result follows immediately from rule NO-OP.

3. Case $c_1; c_2$. If $(c_1; c_2, \mu) \longrightarrow (c_2, \mu')$ then by the first SEQUENCE rule, we have $(c_1, \mu) \longrightarrow \mu'$. By induction, there exists ν' such that $(c_1, \nu) \longrightarrow \nu'$ and $\mu' \sim_\gamma \nu'$. And therefore $(c_1; c_2, \nu) \longrightarrow (c_2, \nu')$ by the first SEQUENCE rule.

   If, instead, $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')$ then $(c_1, \mu) \longrightarrow (c_1', \mu')$ by the second SEQUENCE rule. By induction, there exists ν' such that $(c_1, \nu) \longrightarrow (c_1', \nu')$ and $\mu' \sim_\gamma \nu'$. Hence $(c_1; c_2, \nu) \longrightarrow (c_1'; c_2, \nu')$ by the second SEQUENCE rule.

4. Case **while** e **do** c_1. Since $\gamma \vdash e : L$, we know by Simple Security that $\mu(e) = \nu(e)$. Suppose $\mu(e) = 0$. Then $(\textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow \mu$ and so $(\textbf{while } e \textbf{ do } c_1, \nu) \longrightarrow \nu$ since $\mu(e) = \nu(e)$. If $\mu(e) \neq 0$ then $(\textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow (c_1; \textbf{while } e \textbf{ do } c_1, \mu)$. And since $\nu(e) \neq 0$, $(\textbf{while } e \textbf{ do } c_1, \nu) \longrightarrow (c_1; \textbf{while } e \textbf{ do } c_1, \nu)$.

5. Case **if** e **then** c_1 **else** c_2. Since c is protected, we have $\gamma \vdash e : L$. Therefore, $\mu(e) = \nu(e)$ by Simple Security. So if $\mu(e) \neq 0$ then $(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu)$. And since $\nu(e) \neq 0$, we have $(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \nu) \longrightarrow (c_1, \nu)$. The case when $\mu(e) = 0$ is similar.

6. Case **for** e **do** c_1. Since c is protected, $\gamma \vdash e : L$, and as above $\mu(e) = \nu(e)$. So if $\mu(e) \leq 0$ then (**for** e **do** $c_1, \mu) \longrightarrow \mu$ and since $\mu(e) = \nu(e)$, (**for** e **do** $c_1, \nu) \longrightarrow \nu$. If $\mu(e) > 0$, then (**for** e **do** $c_1, \mu) \longrightarrow (c_1;$ **for** $\mu(e) - 1$ **do** $c_1, \mu)$. And since $\nu(e) > 0$, (**for** e **do** $c_1, \nu) \longrightarrow (c_1;$ **for** $\nu(e) - 1$ **do** $c_1, \nu)$.

7. Case **protect** c_1. By rule ATOMICITY, $(c_1, \mu) \longrightarrow^* \mu'$. Since **protect** blocks are not nested, c_1 is protect free. Further, it is well typed under γ by rule PROTECT since c is well typed. So by Mutual Termination, there is a memory ν' such that $(c_1, \nu) \longrightarrow^* \nu'$ and $\mu' \sim_\gamma \nu'$. Thus, (**protect** $c_1, \nu) \longrightarrow \nu'$ by rule ATOMICITY.

□

Now we wish to extend the Lock Step Execution Lemma to probabilistic states. First, we need a notion of equivalence among probabilistic states. The basic idea is that two probabilistic states are equivalent under γ if they are the same after any high variables are projected out. Suppose, for example, that $x : H$ and $y : L$. Then

$$\left\{ \begin{array}{l} (O, [x := 0, y := 0]) : 1/3, \\ (O, [x := 1, y := 0]) : 1/3, \\ (O', [x := 0, y := 1]) : 1/3 \end{array} \right\}$$

is equivalent to

$$\{(O, [x := 2, y := 0]) : 2/3, (O', [x := 3, y := 1]) : 1/3\},$$

because in each case the result of projecting out the high variable x is

$$\{(O, [y := 0]) : 2/3, (O', [y := 1]) : 1/3\}.$$

Notice that projecting out high variables can cause several configurations to collapse into one, requiring summation of their probabilities. More formally, we define equivalence as follows:

Definition 5.4 *Given variable typing γ and memory μ, let μ_γ denote the result of erasing all high variables from μ. And given probabilistic state s, let the projection of s onto the low variables of γ, denoted s_γ, be defined by*

$$s_\gamma(O, \mu_\gamma) = \sum_{\nu \sim_\gamma \mu} s(O, \nu)$$

Finally, we say that probabilistic states s and s' are equivalent under γ, written $s \sim_\gamma s'$, if $s_\gamma = s'_\gamma$.

A probabilistic state s is well typed and protected under γ if for every global configuration (O, μ) where $s(O, \mu) > 0$, O is well typed and protected under γ, and $dom(\mu) = dom(\gamma)$.

For any global configuration (O, μ), the *point mass on* (O, μ), denoted $\iota_{(O,\mu)}$, is the probabilistic state that gives probability 1 to (O, μ) and probability 0 to all other global configurations.

Now we can show, as a corollary to the Lock Step Execution Lemma, that $\sim_\gamma$ is a congruence with respect to the transition matrix T on well-typed, protected point masses.

Lemma 5.8 (Congruence on Point Masses) *If ι and ι' are well-typed, protected point masses such that $\iota \sim_\gamma \iota'$, then $\iota T \sim_\gamma \iota' T$.*

Proof. Since $\iota \sim_\gamma \iota'$, there must exist a thread pool O and memories μ and ν such that $\iota = \iota_{(O,\mu)}$, $\iota' = \iota_{(O,\nu)}$, and $\mu \sim_\gamma \nu$.

If $O = \{\ \}$, then by third (GLOBAL) rule, we see that $\iota T = \iota$ and $\iota' T = \iota'$. So $\iota T \sim_\gamma \iota' T$.

Now suppose that O is nonempty. We show that for every (O', μ') where $(\iota T)(O', \mu') > 0$, there is a ν' such that $\mu' \sim_\gamma \nu'$ and $(\iota T)(O', \mu') = (\iota' T)(O', \nu')$. So suppose (O', μ') is a global configuration and $(\iota T)(O', \mu') > 0$. Since ι is a point mass,

$$(\iota T)(O', \mu') = T((O, \mu), (O', \mu'))$$

Therefore, $T((O, \mu), (O', \mu')) > 0$. By the definition of T, then, $T((O, \mu), (O', \mu')) = 1/|O|$ and there is a thread α and command c such that $O(\alpha) = c$ and either

1. $(c, \mu) \longrightarrow (c', \mu')$ and $O' = O[\alpha := c']$, or else
2. $(c, \mu) \longrightarrow \mu'$ and $O' = O - \alpha$.

In the first case, we have, by the Lock Step Execution Lemma, that there exists ν' such that $(c, \nu) \longrightarrow (c', \nu')$ and $\mu' \sim_\gamma \nu'$. Then, by rule (GLOBAL), $(O, \nu) \overset{1/|O|}{\Longrightarrow} (O[\alpha := c'], \nu')$, so by definition of T,

$$T((O, \nu), (O', \nu')) = 1/|O|$$

But ι' is also a point mass, therefore

$$(\iota' T)(O', \nu') = T((O, \nu), (O', \nu'))$$

Thus, $(\iota T)(O', \mu') = 1/|O| = (\iota' T)(O', \nu')$. The second case above is similar.

So for a given configuration (O, μ), if $\mu \sim_\gamma \nu$ and $(\iota T)(O, \nu) > 0$, then there exists ν' such that $\nu' \sim_\gamma \nu$ and $(\iota' T)(O, \nu') = (\iota T)(O, \nu)$ from above. Since $\nu' \sim_\gamma \mu$, $(\iota' T)(O, \nu')$ must be in the sum $(\iota' T)_\gamma(O, \mu_\gamma)$. Therefore, $(\iota T)_\gamma(O, \mu_\gamma) \leq (\iota' T)_\gamma(O, \mu_\gamma)$. Symmetrically, we have $(\iota T)_\gamma(O, \mu_\gamma) \geq (\iota' T)_\gamma(O, \mu_\gamma)$ and so $(\iota T)_\gamma = (\iota' T)_\gamma$, or $\iota T \sim_\gamma \iota' T$. □

Now we wish to generalize the above Congruence Lemma from point masses to arbitrary probabilistic states; this generalization is a direct consequence of the linearity of T. More precisely, the set of all measures forms a vector space if we define

- $(s + s')(O, \mu) = s(O, \mu) + s'(O, \mu)$, for measures s and s', and
- $(as)(O, \mu) = a(s(O, \mu))$, for real a and measure s.

With respect to this vector space, T is a linear transformation. Furthermore, $\sim_\gamma$ is a congruence with respect to the vector space operations:

Lemma 5.9 *If $s_i \sim_\gamma s'_i$ for all i, then*

$$a_1 s_1 + a_2 s_2 + a_3 s_3 + \cdots \ \sim_\gamma \ a_1 s'_1 + a_2 s'_2 + a_3 s'_3 + \cdots$$

Proof. First, we have that projections are homomorphic,

$$(a_1 s_1 + a_2 s_2 + \cdots)_\gamma = (a_1 s_1)_\gamma + (a_2 s_2)_\gamma + \cdots$$

since for all (O, μ),

$$\begin{aligned}
(a_1 s_1 + a_2 s_2 + \cdots)_\gamma (O, \mu_\gamma) &= \sum_{\nu \sim_\gamma \mu} a_1 s_1(O, \nu) + a_2 s_2(O, \nu) + \cdots \\
&= \sum_{\nu_1 \sim_\gamma \mu} a_1 s_1(O, \nu_1) + \sum_{\nu_2 \sim_\gamma \mu} a_2 s_2(O, \nu_2) + \cdots \\
&= (a_1 s_1)_\gamma (O, \mu_\gamma) + (a_2 s_2)_\gamma (O, \mu_\gamma) + \cdots \\
&= ((a_1 s_1)_\gamma + (a_2 s_2)_\gamma + \cdots)(O, \mu_\gamma)
\end{aligned}$$

Further, $(as)_\gamma = a s_\gamma$ since for all (O, μ),

$$(as)_\gamma (O, \mu_\gamma) = \sum_{\nu \sim_\gamma \mu} (as)(O, \nu) = \sum_{\nu \sim_\gamma \mu} a(s(O, \nu)) = a \sum_{\nu \sim_\gamma \mu} s(O, \nu) = a s_\gamma (O, \mu_\gamma)$$

Finally, we have

$$\begin{aligned}
(a_1 s_1 + a_2 s_2 + \cdots)_\gamma &= (a_1 s_1)_\gamma + (a_2 s_2)_\gamma + \cdots \\
&= a_1 s_{1_\gamma} + a_2 s_{2_\gamma} + \cdots \\
&= a_1 s'_{1_\gamma} + a_2 s'_{2_\gamma} + \cdots \\
&= (a_1 s'_1)_\gamma + (a_2 s'_2)_\gamma + \cdots \\
&= (a_1 s'_1 + a_2 s'_2 + \cdots)_\gamma
\end{aligned}$$

So $a_1 s_1 + a_2 s_2 + \cdots \ \sim_\gamma \ a_1 s'_1 + a_2 s'_2 + \cdots$. □

Theorem 5.10 (Probabilistic Noninterference) *If s and s' are well-typed, protected probabilistic states such that $s \sim_\gamma s'$, then $sT \sim_\gamma s'T$.*

Proof. To begin with, we argue that s and s' can be expressed as (possibly countably infinite) linear combinations of (not necessarily distinct) point masses such that the corresponding coefficients are the same, and the corresponding point masses are equivalent.

Now, we know that we can express s and s' as linear combinations of point masses:

$$s = a_1 \iota_1 + a_2 \iota_2 + a_3 \iota_3 + \cdots$$

and

$$s' = b_1 \iota'_1 + b_2 \iota'_2 + b_3 \iota'_3 + \cdots$$

Assume, for now, that s_γ (and s'_γ) is a point mass. That is, $\iota_i \sim_\gamma \iota_j \sim_\gamma \iota'_i \sim_\gamma \iota'_j$ for all i and j.

Note that the a_i's and b_i's both sum to 1; hence they both can be understood as partitioning the unit interval $[0, 1]$:

a_1	a_2	a_3	$\cdots$

b_1	b_2	b_3	b_4	$\cdots$

0 1

To unify the coefficients in the two linear combinations, we must take the *union* of the two partitions, splitting up any terms that cross partition boundaries. For example, based on the picture above we would split the term $a_1\iota_1$ of s into $b_1\iota_1 + (a_1 - b_1)\iota_1$. And we would split the term $b_2\iota'_2$ of s' into $(a_1 - b_1)\iota'_2 + (b_2 - (a_1 - b_1))\iota'_2$. Continuing in this way, we can unify the coefficients of s and s'.

We can describe the splitting process more precisely as follows. We simultaneously traverse s and s', splitting terms as we go. Let $a\iota$ and $b\iota'$ be the next terms to be unified. If $a = b$, then keep both these terms unchanged. If $a < b$, then keep term $a\iota$ in s, but split $b\iota'$ into $a\iota'$ and $(b - a)\iota'$ in s'. Handle the case $a > b$ symmetrically. If one or both of the sums are infinite, then of course the algorithm gives an infinite sum. But each term of s and of s' is split only finitely often (otherwise the a_i's and b_i's would not have the same sum) with one exception—if s is a finite sum and s' is an infinite sum, then the last term of s is split into an infinite sum.

So far, we have shown how to unify the coefficients of s and s' in the case where s_γ (and s'_γ) is a point mass. In the general case, s and s' must first be rearranged into sums of sums of equivalent point masses:

$$s = (a_{11}\iota_{11} + a_{12}\iota_{12} + \cdots) + (a_{21}\iota_{21} + a_{22}\iota_{22} + \cdots) + \cdots$$

and

$$s' = (b_{11}\iota'_{11} + b_{12}\iota'_{12} + \cdots) + (b_{21}\iota'_{21} + b_{22}\iota'_{22} + \cdots) + \cdots$$

where $\iota_{ij} \sim_\gamma \iota_{ik} \sim_\gamma \iota'_{ij} \sim_\gamma \iota'_{ik}$ for all i, j, and k. Also, for each i, $\sum_j a_{ij} = \sum_j b_{ij}$. Hence we can apply the algorithm above to unify the a_{1j}'s with the b_{1j}'s, the a_{2j}'s with the b_{2j}'s, and so forth. Then we can form a single sum for s and for s' by interleaving these sums in a standard way.

The final result of all this effort is that we can express s and s' as

$$s = c_1\iota''_1 + c_2\iota''_2 + c_3\iota''_3 + \cdots$$

and

$$s' = c_1\iota'''_1 + c_2\iota'''_2 + c_3\iota'''_3 + \cdots$$

where $\iota''_i \sim_\gamma \iota'''_i$ for all i. Now, since T is a linear transformation, we have

$$sT = c_1(\iota''_1 T) + c_2(\iota''_2 T) + c_3(\iota''_3 T) + \cdots$$

and

$$s'T = c_1(\iota'''_1 T) + c_2(\iota'''_2 T) + c_3(\iota'''_3 T) + \cdots$$

By the Congruence on Point Masses Lemma, we have $\iota''_i T \sim_\gamma \iota'''_i T$, for all i. So, by Lemma 5.9, $sT \sim_\gamma s'T$. □

6 Discussion

The need for a probabilistic view of security in nondeterministic computer systems has been understood for some time [17, 13]. Security properties (models) to treat probabilistic channels in nondeterministic systems have been formulated by McLean[12] and Gray [7, 8].

It is important, however, to recognize that these efforts address a different problem than what we consider in this paper. They consider a computer system with a number of *users*, classified as high or low, who send inputs to and receive outputs from the system. The problem is to prevent high users, who have access to high information, from communicating with low users, who should have access only to low information. What makes treating privacy in this setting especially difficult is that users need not be processes under control of the system—they may be *people*, who are *external* to the system and who can observe the system's behavior from the outside. As a result, a high user may be able to communicate covertly by modulating system performance to encode high information that a low user in turn decodes using a real-time clock outside the system. Furthermore, because the low user is measuring *real time*, the modulations can depend on low-level system implementation details, such as the paging and caching behavior of the underlying hardware. This implies that it is not enough to prove privacy with respect to a high-level, abstract system semantics (like the semantics of Figure 1). To guarantee privacy, it is necessary for the system model to address all the implementation details.

In a mobile-code framework, where hosts are trusted, ensuring privacy is more tractable. A key assumption here is that any attempt to compromise privacy must arise from within the mobile code, which is *internal* to the system. As a result, the system can control what the mobile code can do and what it can observe. For example, if mobile-code threads are not allowed to see a real-time clock, then they can measure the timing of other threads only by observing variations in thread interleavings. Hence, assuming a correct implementation of our semantics, threads will not be able to detect any variations in the running time of a protected command, nor will they be able to detect timing variations due to low-level implementation details. Consequently, timing attacks are impossible in well-typed, protected programs in our language. For instance, Kocher describes a timing attack on RSA [10]. Basically, he argues that an attacker can discover a private key x by observing the amount of time required by several modular exponentiations $y^x \bmod n$. In our framework, one would use a protected **for** loop to implement modular exponentiation, which means that no useful timing information about exponentiation would be available to other threads—it would always appear to execute in exactly one step.

7 Other related research

Other work in secure information flow, in a parallel setting, includes that of Andrews and Reitman [1], Melliar-Smith and Moser [14], Focardi and Gorrieri [5, 6], and Banatre and Bryce [2]. Melliar-Smith and Moser consider covert channels in Ada. They describe a data dependency analysis to find places where Ada programs depend on the relative timing of operations within a system. Andrews and Reitman give an axiomatic flow logic for treating information flow in the presence of process synchronization. They also sketch how one might treat timing channels in the logic. Banatre and Bryce give an axiomatic flow logic for CSP processes, also treating information flow arising from synchronization. None of these efforts, though, gives a satisfactory account of the security properties that they guarantee. Focardi and Gorrieri identify trace-based and bisimulation-based security properties for systems expressed in an extension of Milner's CCS, which they call the Security Process Algebra. These properties, however, are possibilistic in nature (e.g. a system is *SNNI* [6] if the set of

traces that a low observer can see of a system is possible regardless of whether high-level actions are enabled or disabled in the system).

8 Conclusion

So what is the significance of our result? It depends on what can be observed. With respect to internal program behavior, our Probabilistic Noninterference result rules out all covert flows from high variables to low variables. But if *external* observation of the running program is allowed, then of course covert channels of the kind discussed in Section 6 remain possible. Note, however, that the mobile code setting affords us more control over external observations than would normally be possible. When we execute some mobile code on our machine, we can limit communication with the outside world, preventing precise observations of a program's behavior such as its running time. Depending on the application, one can build enough noise into the mobile code's interface with the outside in various ways to significantly reduce the capacity of an externally-observable timing channel. See, for example, the NRL Pump for secure acknowledgment [9].

References

[1] G. Andrews and R. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[2] J. Banâtre and C. Bryce. Information flow control in a parallel language framework. In *Proceedings 6th IEEE Computer Security Foundations Workshop*, pages 39–52, June 1993.

[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] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1994/1995.

[6] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997.

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

[8] James W. Gray, III. Toward a mathematical foundation for information flow security. In *Proceedings 1991 IEEE Symposium on Security and Privacy*, pages 21–34, Oakland, CA, May 1991.

[9] Myong H. Kang and Ira S. Moskowitz. A pump for rapid, reliable secure communication. In *Proceedings of the 1st ACM Conference on Computer & Communications Security*, pages 119–129, November 1993.

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

[11] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

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

[13] John McLean. Security models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.

[14] P.M. Melliar-Smith and L. Moser. Protection against covert storage and timing channels. In *Proceedings 4th IEEE Computer Security Foundations Workshop*, pages 209–214, June 1991.

[15] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, A Formal Introduction*. Wiley, 1992.

[16] Geoffrey Smith and Dennis 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, January 1998.

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