Probabilistic Noninterference through Weak Probabilistic Bisimulation

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

Abstract

To be practical, systems for ensuring secure information flow must be as permissive as possible. To this end, the author recently proposed a type system for multi-threaded programs running under a uniform probabilistic scheduler; it allows the running times of threads to depend on the values of H variables, provided that these timing variations cannot affect the values of L variables. But these timing variations preclude a proof of the soundness of the type system using the framework of probabilistic bisimulation, because probabilistic bisimulation is too strict regarding time. To address this difficulty, this paper proposes a notion of weak probabilistic bisimulation for Markov chains, allowing two Markov chains to be regarded as equivalent even when one "runs" more slowly than the other. The paper applies weak probabilistic bisimulation to prove that the type system guarantees the probabilistic noninterference property. Finally, the paper shows that the language can safely be extended with a **fork** *command that allows new threads to be spawned.*

1 Introduction

The *secure information flow* problem is concerned with finding techniques to ensure that programs do not leak sensitive data. It is a well-studied problem; see [14] for a comprehensive survey. Recently, the author proposed a new type system for secure information flow in a simple multi-threaded imperative programming language running under a uniform probabilistic scheduler [16]. The type system classifies program variables as either L (public) or H (private) and imposes restrictions to ensure that no information can leak from H variables to L variables; this is formalized as a property called *probabilistic noninterference* that asserts that the probability distribution on the final values of L variables is independent of the initial values of H variables. The type system aims to be as permissive as possible; for example it allows the running time of threads to depend on the values of H variables, so long as these timing variations do not affect the values of L variables.

Here's a simple example of the sort of multi-threaded program we are considering. Let thread α be

$$\textbf{while } x > 0 \textbf{ do } x := x - 1$$

let thread β be

$$y := 1$$

and let thread γ be

$$y := 2$$

where x is a H variable, which is assumed to be non-negative, and y is a L variable. We run this program under a *uniform probabilistic scheduler*, which at each computation step chooses either thread α, β, or γ, each with probability $1/3$. The state of the running program is given by a *global configuration* (O, μ) consisting of the thread pool O and the shared memory μ. For example, if we start in a memory with $x = 2$ and $y = 0$, the program might pass through the sequence of global configurations shown in Figure 1, where the annotation on $\Downarrow$ gives the probability of making that transition.

Does this program satisfy secure information flow? The answer depends on what observations are permitted. If we can observe the program from the *outside*, seeing when threads terminate, then the program is clearly dangerous—the running time of thread α reveals information about the initial value of the H variable x. But if we can only observe the program from the *inside*, seeing just the final values of L variables, then the program is safe, because it satisfies probabilistic noninterference—regardless of the initial value of x, the final value of y is either 1 or 2, each with probability 1/2. In this work (as in [16] and [18]) our concern is only with preventing internal leaks, by establishing probabilistic noninterference. Certainly there are applications where this approach is not good enough, but it does seem well-suited to the case of mobile code, since mobile code cannot be observed externally unless the host computer allows it to

$$(\{\alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1,\ \beta = y := 1,\ \gamma = y := 2\}, [x = 2, y = 0])$$
$$\Downarrow \tfrac{1}{3}$$
$$(\{\alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1,\ \beta = y := 1\}, [x = 2, y = 2])$$
$$\Downarrow \tfrac{1}{2}$$
$$(\{\alpha = x := x - 1; \textbf{while } x > 0 \textbf{ do } x := x - 1,\ \beta = y := 1\}, [x = 2, y = 2])$$
$$\Downarrow \tfrac{1}{2}$$
$$(\{\alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1,\ \beta = y := 1\}, [x = 1, y = 2])$$
$$\Downarrow \tfrac{1}{2}$$
$$(\{\alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1\}, [x = 1, y = 1])$$
$$\Downarrow 1$$
$$(\{\alpha = x := x - 1; \textbf{while } x > 0 \textbf{ do } x := x - 1\}, [x = 1, y = 1])$$
$$\Downarrow 1$$
$$(\{\alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1\}, [x = 0, y = 1])$$
$$\Downarrow 1$$
$$(\{\ \}, [x = 0, y = 1])$$

Figure 1. An example probabilistic execution

be. And, of course, preventing external leaks requires much more severe restrictions on programs—see for example [2].

Returning to our example program, suppose that we combine threads α and γ sequentially, so that thread α becomes

$$\textbf{while } x > 0 \textbf{ do } x := x - 1;$$
$$y := 2$$

and thread β remains $y := 1$. In this case, the program is unsafe, even with respect to internal observations, because now the likely outcome of the race between the two assignments $y := 1$ and $y := 2$ depends on the intial value of x. More precisely, the larger the initial value of x, the greater the probability that the final value of y is 2. For example, a direct simulation shows that if the initial value of x is 0, then the final value of y is 1 with probability $1/4$ and 2 with probability $3/4$; but if the initial value of x is 5, then the final value of y is 1 with probability $1/4096$ and 2 with probability $4095/4096$. Hence this program does not satisfy probabilistic noninterference. (Note that it does, however, satisfy *possibilistic noninterference* [17] because, regardless of the initial value of x, it is *possible* for the final value of y to be either 1 or 2.)

The type system of [16] works by *classifying* and *restricting* the expressions and commands of a program. The classifications are as follows:

- An expression e is classified as H if it contains any H variables; otherwise it is classified as L.
- A command c is classified as τ_1 *cmd* τ_2 if it assigns only to variables of type τ_1 (or higher) and its running time depends only on variables of type τ_2 (or lower).
- A command c is classified as τ *cmd* n if it assigns only to variables of type τ (or higher) and it is guaranteed to terminate in exactly n steps.

Using these classifications, the type system enforces the following restrictions:

- Only L expressions can be assigned to L variables.
- A guarded command with H guard cannot assign to L variables.
- A command whose running time depends on H variables cannot be followed sequentially by a command that assigns to L variables.

Notice that this last restriction disallows the modified thread α considered above—because the running time of **while** $x > 0$ **do** $x := x - 1$ depends on the H variable x, it cannot be followed sequentially by an assignment to the L variable y.

In [16], it is argued that any multi-threaded program that is well typed under the above rules satisfies the probabilistic noninterference property. We sketch the approach here. First, we say that two memories μ and ν are equivalent, written $\mu \sim_\Gamma \nu$, if they agree on the values of L variables.

(Here Γ denotes the *identifier typing* that classifies the program variables as L or H.) The key property ensured by the type system is that if a well-typed command c is run twice, under two equivalent memories, then in each execution it will make exactly the same sequence of assignments to L variables, at the same times.

More precisely, we can define an equivalence relation $\sim_\Gamma$ on well-typed commands such that if equivalent commands c and d are run for a single step under equivalent memories μ and ν, then the results are equivalent in the sense that the resulting commands and memories are still equivalent. (This is the Sequential Noninterference Theorem of [16].) A subtle point, however, is that (c, μ) might terminate in one step, going to a terminal configuration μ', while (d, ν) might not, going to a non-terminal configuration (d', ν'). In this case, however, it is guaranteed that d' will have a type of the form H *cmd* _ , which means that it will make no further assignments to L variables.[1]

Now consider a pool O of threads running concurrently. Our semantics specifies that at each step, we pick a thread at random and run it for a step. This makes our program into a Markov chain whose states are global configurations (O, μ) consisting of a thread pool and a shared memory. Hence if we have a well-typed thread pool O and equivalent memories μ and ν, we would like to argue some sort of equivalence between the Markov chains starting from (O, μ) and from (O, ν).

In [16], the notion of equivalence used is *probabilistic bisimulation*, which we review in Section 3. Unfortunately, this equivalence is not quite suited to our type system, because it is too strict with respect to timing; in particular, probabilistic bisimulation cannot accommodate threads whose running time depends on H variables. This mismatch is handled clumsily in [16] by adopting a strange semantics in which threads never terminate—completed threads remain alive in the thread pool, performing **skip**s. The main goal of this paper is to correct this deficiency, allowing us to show probabilistic noninterference with respect to the usual semantics, which removes completed threads from the thread pool. To do this, we introduce a notion of *weak probabilistic bisimulation*, which is more relaxed with respect to timing.

The remainder of this paper is organized as follows. In Section 2, we review the multi-threaded language and its type system. In Section 3, we discuss probabilistic bisimulation on Markov chains abstractly; we introduce a weak version and discuss techniques for calculating the relevant probabilities. Then in Section 4, we apply weak probabilistic bisimulation to establish probabilistic noninterference for well-typed multi-threaded programs. In Section 5, we show that we can accommodate an extended language with a **fork** command that allows us to spawn new threads dynamically. Finally, Section 6 concludes.

[1] We write H *cmd* _ to indicate that we don't care about the "running time" component of the type.

2 The Multi-Threaded Language and Type System

Threads are written in the simple imperative language:

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

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

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

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

A multi-threaded program consists of a pool of threads running under a shared memory μ. Formally, a *thread pool* O is a mapping from thread identifiers (α, β, ...) to commands. A multi-threaded program is executed in an interleaving manner, by repeatedly choosing a thread to run for a step. We assume that the choice is made probabilistically, with each thread having an equal probability of being chosen at each step—that is, we assume a *uniform thread scheduler*. We formalize this by defining a global transition relation $\stackrel{p}{\Longrightarrow}$ on global configurations:

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

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

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

The judgment $(O,\mu)\stackrel{p}{\Longrightarrow}(O',\mu')$ asserts that the probability of going from global configuration (O,μ) to (O',μ') is p. Note that $O-\alpha$ denotes the thread pool obtained by removing thread α from O, and $O[\alpha:=c']$ denotes the thread pool obtained by updating the command associated with α to c'. The third rule (GLOBAL), which deals with an empty thread pool, allows us to view a multi-threaded program as a discrete Markov chain [8]. The states of the Markov chain are global configurations and the transition matrix is governed by $\stackrel{p}{\Longrightarrow}$.

The type system of [16] is based upon the following types:

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

The rules of the type system are given in Appendix B; they allow us to prove *typing judgments* of the form $\Gamma \vdash p : \rho$ as well as *subtyping judgments* of the form $\rho_1 \subseteq \rho_2$. Here Γ denotes an *identifier typing* which maps identifiers to types of the form τ *var*. In the rules, $\vee$ denotes *join* and $\wedge$ denotes *meet*; more details can be found in [16]. Remarkably, Boudol and Castellani [7] independently developed a type system almost identical to that of [16], except that their system does not include types of the form τ *cmd* n.

3 Probabilistic Bisimulation on Markov Chains

In this section, we consider notions of probabilistic bisimulation on Markov chains abstractly; we will apply them to the secure information flow problem in Section 4.

Given a finite or countably infinite Markov chain [8] with state set S and transition probabilities p_{st} for $s,t \in S$, we may be able to define an equivalence relation $\approx$ on S such that for any equivalence class A, we don't care which state within the equivalence class we are in. Then when we "run" the Markov chain, we care only about the sequence of equivalence classes entered.

For example, in the information flow setting, we don't care whether the configuration is (O,μ) or (O,ν) if μ and ν agree on the values of L variables.

A natural question is whether we can form a *quotient Markov chain* $S/\!\approx$ whose states are the equivalence classes of $\approx$. This turns out to be possible iff $\approx$ is a *probabilistic bisimulation*, which means that for all equivalence classes A and B, the probability of going (in one step) from a state $a \in A$ to some state in B is independent of a; that is, for any $a' \in A$,

$$\sum_{b\in B} p_{ab} = \sum_{b\in B} p_{a'b} \tag{1}$$

We denote this common probability by p_{AB}. (The condition was first identified by Kemeny and Snell [11], who called it "lumpability"; Larsen and Skou [12] later called it "probabilistic bisimulation". It was first applied to the information flow problem by Sabelfeld and Sands [15].)

Note, however, that this condition is very strong with respect to timing; if we run the Markov chain starting from two equivalent states, then the two runs will need to pass through the same equivalence classes at the same times.

But the general approach of the type system of [16] is to assume that the real running time of the program is not observable. (For example, that is what makes the use of the **protect** construct justifiable.) Given this assumption, it would be preferable to adopt a notion of probabilistic bisimulation that is less demanding about timing. In particular, if two runs reach the same outcome, but one runs more slowly than the other, this should be acceptable.

For example, consider the Markov chain given in Figure 2, where the dashed boxes denote the equivalence classes of $\approx$. In this case $\approx$ is not a probabilistic bisimulation, because states a_1 and a_2 have different probabilities of going in one step to the equivalence class B; a_1 goes with probability $1/3$, while a_2 goes with probability $2/3$.

However, if we abstract away from time, then it seems reasonable to say that states a_1 and a_2 are equivalent, since we can show that both have probability $2/3$ of going to equivalence class B, possibly after "stuttering" within class A for a while.

We can make this notion of *weak probabilistic bisimulation* precise using an approach similar to that of Baier and Hermanns [5]. Given two distinct equivalence classes A and B and state $a \in A$, we let $\mathcal{P}(a,A,B)$ denote the probability of starting at a, moving for 0 or more steps within A, and then entering B. Following [5], we observe that these probabilities solve the equation system:

$$\mathcal{P}(a,A,B) = \sum_{b\in B} p_{ab} + \sum_{a'\in A} p_{aa'}\mathcal{P}(a',A,B) \tag{2}$$

For example, in the above Markov chain we have

$$\mathcal{P}(a_2,A,B) = p_{a_2 b} = \frac{2}{3}$$

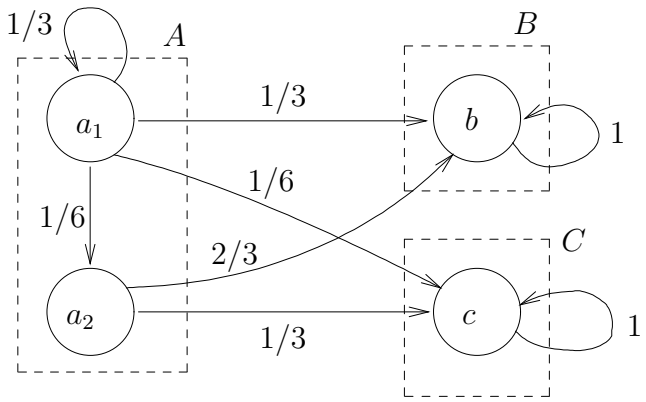

Figure 2. An example weak probabilistic bisimulation

and

$$\begin{aligned}\mathcal{P}(a_1, A, B) &= p_{a_1 b} + p_{a_1 a_1}\mathcal{P}(a_1, A, B) + \\ &\quad p_{a_1 a_2}\mathcal{P}(a_2, A, B) \\ &= \frac{1}{3} + \frac{1}{3}\mathcal{P}(a_1, A, B) + \frac{1}{6}\cdot\frac{2}{3} \\ &= \frac{4}{9} + \frac{1}{3}\mathcal{P}(a_1, A, B)\end{aligned}$$

which implies that

$$\mathcal{P}(a_1, A, B) = \frac{2}{3}.$$

We can now define *weak probabilistic bisimulation*:

Definition 3.1 *Equivalence relation $\approx$ is a* weak probabilistic bisimulation *if, for all distinct equivalence classes A and B, $\mathcal{P}(a, A, B)$ is independent of the choice of a. In this case, we define $\mathcal{P}(A, B)$ to be this unique value.*

Weak probabilistic bisimulation is an appropriate notion if we are interested in the sequence of equivalence classes of $\approx$ that are visited, but we don't care how long the chain remains in each class.

As noted above, our notion of weak probabilistic bisimulation is similar to that proposed in [5]. Also, Aldini [3] has recently applied this notion to the secure information flow problem. However, it should be noted that these efforts are based in a process algebra setting, in which transitions are labeled with actions and the "weakness" of the bisimulation is based on disregarding "internal" actions, namely those labeled with τ.[2] In contrast, the weak probabilistic bisimulation that we develop here does not rely on an *a priori* notion that certain "internal" transitions can be ignored. Instead, our notion of which transitions can be ignored is based solely on the equivalence relation $\approx$; that is, a Markov chain transition can be ignored precisely if it stays within the same equivalence class of $\approx$.

Calculating the probabilities $\mathcal{P}(a, A, B)$ is, unfortunately, more subtle in general than is suggested by the example above. The trouble is that equation system (2) need not have a unique solution. One classic example that illustrates this is a *random walk* Markov chain [8], as shown in Figure 3. (Here p and q can be any numbers satisfying $p, q \geq 0$ and $p + q = 1$.) In this case, equation system (2) specializes to

$$\begin{aligned}\mathcal{P}(1, A, B) &= q + p\mathcal{P}(2, A, B) \\ \mathcal{P}(z, A, B) &= q\mathcal{P}(z-1, A, B) + p\mathcal{P}(z+1, A, B), \\ &\quad \text{for } z > 1\end{aligned}$$

Now it is easy to see that

$$\mathcal{P}(z, A, B) = 1$$

solves the equation system. But

$$\mathcal{P}(z, A, B) = (\frac{q}{p})^z$$

also solves the equation system, provided that $p > 0$. In fact, Feller [8] shows that the actual probabilities are

$$\mathcal{P}(z, A, B) = \begin{cases} 1, & \text{if } p \leq q \\ (\frac{q}{p})^z, & \text{if } p \geq q \end{cases}$$

We further remark that equation system (2) need not be uniquely solvable even in the case of a Markov chain with finitely many states. Consider the example of Figure 4. In this case, equation system (2) specializes to

$$\begin{aligned}\mathcal{P}(1, A, B) &= \frac{1}{2} + \frac{1}{2}\mathcal{P}(2, A, B) \\ \mathcal{P}(2, A, B) &= 1\mathcal{P}(2, A, B),\end{aligned}$$

[2] Of course, this use of the symbol "τ" has nothing to do with our use of it in the type system!

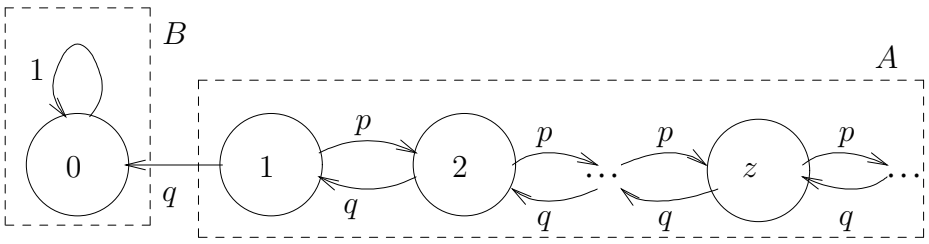

Figure 3. A random walk Markov chain

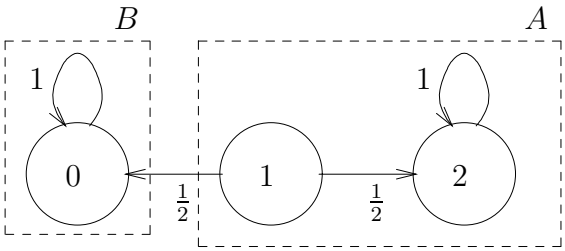

Figure 4. A finite Markov chain with multiple solutions to equation system (2)

so that infinitely many solutions are possible. Of course, it is obvious here that really $\mathcal{P}(1, A, B) = 1/2$ and $\mathcal{P}(2, A, B) = 0$. Notice that these values are the *minimal* non-negative solutions to the equation system. This turns out to hold in general, as is shown by the following theorem, which is adapted from Theorem 1.3.2 of [13]:

Theorem 3.1 *The values of $\mathcal{P}(a, A, B)$ for $a \in A$ are the minimal non-negative solution to the equation system*

$$\mathcal{P}(a, A, B) = \sum_{b \in B} p_{ab} + \sum_{a' \in A} p_{aa'} \mathcal{P}(a', A, B)$$

(Here minimality means that if x_a satisfies

$$x_a = \sum_{b \in B} p_{ab} + \sum_{a' \in A} p_{aa'} x_{a'}$$

and $x_a \geq 0$ for all a, then $x_a \geq \mathcal{P}(a, A, B)$ for all a.)

In the following section, we will apply Theorem 3.1 in calculating $\mathcal{P}(a, A, B)$ for the equivalence relation $\sim_\Gamma$ that we will define; this will enable us to show that $\sim_\Gamma$ is a weak probabilistic bisimulation.

4 Noninterference via Weak Probabilistic Bisimulation

In this section, we apply weak probabilistic bisimulation to prove that our type system guarantees probabilistic noninterference. We do this by defining an equivalence relation $\sim_\Gamma$ on well-typed global configurations (O, μ) and arguing that it is a weak probabilistic bisimulation.

The idea is that we should have $(O, \mu) \sim_\Gamma (O, \nu)$ provided that μ and ν agree on the values of L variables, and (assuming that $\sim_\Gamma$ is a weak probabilistic bisimulation) we will know that the probability of ending up eventually in some equivalence class from (O, μ) will be the same as the probability of reaching it from (O, ν).

Of course this means that $\sim_\Gamma$ cannot be just *any* weak probabilistic bisimulation. In particular, the identity relation (which puts each global configuration into a distinct equivalence class) and the universal relation (which puts all global configurations into the same equivalence class) are both trivially weak probabilistic bisimulations, but they are not suitable. The identity relation is too fine, because it does not equate (O, μ) and (O, ν) if μ and ν are distinct memories, even if they agree on the values of L variables. And the universal relation is too coarse, because it equates (O, μ) and (O', ν) even if μ and ν disagree on the values of L variables.

So to get probabilistic noninterference, we must design $\sim_\Gamma$ to be a weak probabilistic bisimulation on well-typed global configurations that satisfies

- $(O, \mu) \sim_\Gamma (O, \nu)$, if μ and ν agree on the values of L variables, and
- $(O, \mu) \not\sim_\Gamma (O', \nu)$, if μ and ν disagree on the values of L variables.

Beyond these conditions, we have freedom in designing $\sim_\Gamma$.

We begin by reviewing some definitions and results from [16]. First we define $\sim_\Gamma$ on *memories*:

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

Next we define $\sim_\Gamma$ on *commands*. Before doing this, we note that any command c can be written in the *standard form*

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

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

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

Definition 4.2 *Commands c and d are equivalent with respect to Γ, written $c \sim_\Gamma d$, if c and d are both well typed under Γ and either*

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

(The last possibility is needed to handle executions of an **if** command with type H *cmd* n.)

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

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

(In effect, we are saying that a command with type of the form H *cmd* $\_$ is equivalent to a terminated command.) Finally, we recall the key Sequential Noninterference result from [16]:

Theorem 4.1 (Sequential Noninterference) *Suppose that $(c, \mu) \sim_\Gamma (d, \nu)$, $(c, \mu) \longrightarrow C'$, and $(d, \nu) \longrightarrow D'$. Then $C' \sim_\Gamma D'$.*

Now we are ready to define $\sim_\Gamma$ on global configurations. The basic idea is that $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ iff $\mu \sim_\Gamma \nu$ and $O_1(\alpha) \sim_\Gamma O_2(\alpha)$ for all α. However, because threads may terminate at different times due to changes in the initial values of H variables, we must allow O_1 and O_2 to each contain extra threads not contained in the other, provided that such threads have types of the form H *cmd* $\_$, making them unimportant as far as L variables are concerned.

Definition 4.3 $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ *iff*

1. $\mu \sim_\Gamma \nu$,
2. $O_1(\alpha) \sim_\Gamma O_2(\alpha)$ *for all* $\alpha \in$ *dom*$(O_1) \cap$ *dom*(O_2),
3. $O_1(\alpha)$ *has type of the form* H *cmd* $\_$ *for all* $\alpha \in$ *dom*$(O_1) -$ *dom*(O_2), *and*
4. $O_2(\alpha)$ *has type of the form* H *cmd* $\_$ *for all* $\alpha \in$ *dom*$(O_2) -$ *dom*(O_1).

We remark that this definition significantly relaxes that of [16]; there we required that *dom*$(O_1) =$ *dom*(O_2).

With this relaxed definition, $\sim_\Gamma$ is no longer a probabilistic bisimulation, but it is a weak probabilistic bisimulation. The basic idea is that if $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ and O_1 and O_2 both contain a thread α, then by definition we have $O_1(\alpha) \sim_\Gamma O_2(\alpha)$, which implies (by the Sequential Noninterference Theorem) that if thread α is chosen by the scheduler, then (O_1, μ) goes to the same equivalence class as does (O_2, ν). But if O_1 contains a thread β not present in O_2, then $O_1(\beta)$ must have type of the form H *cmd* $\_$ and hence choosing β to run for a step will keep (O_1, μ) in the *same* equivalence class. Thus such extra threads only add extra "stuttering"; they don't affect the probabilities of going from (O_1, μ) to any other equivalence class.

Theorem 4.2 *Relation $\sim_\Gamma$ is a weak probabilistic bisimulation on the Markov chain of global configurations.*

Proof. Let A and B be distinct equivalence classes of $\sim_\Gamma$. Then we must show that $\mathcal{P}(a, A, B)$ is independent of the choice of a.

Consider any state a in A; of course a is actually a global configuration (O, μ). Suppose that O contains a thread α such that running α for a step takes us to a global configuration in some equivalence class C other than A; we will say that such a thread is *essential*. Then thread α cannot have a type of the form H *cmd* $\_$, since otherwise running α would keep us in class A. Therefore (by Definition 4.3) *every* global configuration in A must contain an equivalent thread α which (by the Sequential Noninterference Theorem) must also take us to a global configuration in class C. The conclusion is that every global configuration in A must contain the same number e of essential threads that lead directly out of class A, and also the same number e_C

of threads leading directly to equivalence class C. In addition, each global configuration $a \in A$ contains some number u_a of *unessential* threads, whose types are of the form H *cmd* _ and whose execution leaves us within class A.

Recall for example the program shown in Figure 1. Let a be the global configuration

$$\left(\left\{ \begin{array}{l} \alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1 \\ \beta = y := 1 \\ \gamma = y := 2 \end{array} \right\}, [x = 2, y = 0] \right)$$

and let A be its equivalence class. Also, let B be the equivalence class of

$$\left(\left\{ \begin{array}{l} \alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1, \\ \gamma = y := 2 \end{array} \right\}, [x = 2, y = 1] \right)$$

and C be the equivalence class of

$$\left(\left\{ \begin{array}{l} \alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1, \\ \beta = y := 1 \end{array} \right\}, [x = 2, y = 2] \right)$$

Then for each global configuration in A, $e_B = 1$, $e_C = 1$, and $e = 2$; each must contain essential threads equivalent to β and γ. Furthermore, $u_a = 1$ because a contains just one unessential thread, α; other global configurations in A can contain more or fewer unessential threads.

Note however that if global configuration $a' \in A$ is reachable from a, then $u_{a'} \leq u_a$, since (with the current language) new threads cannot be created during program execution.

We are now ready to calculate $\mathcal{P}(a, A, B)$ and to show that its value is independent of a. To begin with, note that if $e = 0$, then $\mathcal{P}(a, A, B) = 0$, since there is no possibility of leaving class A. Next suppose that $e > 0$. Then we claim that $\mathcal{P}(a, A, B)$ is independent of u_a, and in fact $\mathcal{P}(a, A, B) = \frac{e_B}{e}$.

To justify this, first note that if we start from a, then the probability of leaving A in one step is $\frac{e}{e+u_a}$. So, remembering that $u_{a'} \leq u_a$ for any a' reachable from a, we see that the probability of *not* leaving A after k steps is at most $(\frac{u_a}{e+u_a})^k$, which goes to 0 as $k \to \infty$. Hence with probability 1, A is eventually left. (Indeed, using standard facts about geometric random variables, the expected number of steps is at most $\frac{e+u_a}{e}$.) Hence, if we let $\mathcal{C}$ denote the set of all equivalence classes of $\sim_\Gamma$, we have

$$\sum_{B \in \mathcal{C} - \{A\}} \mathcal{P}(a, A, B) = 1. \tag{3}$$

Now, by Theorem 3.1, the values of $\mathcal{P}(a, A, B)$ are the minimal non-negative solution to the equation system

$$\mathcal{P}(a, A, B) = \sum_{b \in B} p_{ab} + \sum_{a' \in A} p_{aa'} \mathcal{P}(a', A, B).$$

Next we observe that

$$\mathcal{P}(a, A, B) = \frac{e_B}{e}$$

solves the equation system:

$$\begin{array}{rcl} \mathcal{P}(a, A, B) & = & \displaystyle\sum_{b \in B} p_{ab} + \sum_{a' \in A} p_{aa'} \mathcal{P}(a', A, B) \\ & = & \displaystyle\frac{e_B}{e + u_a} + \frac{u_a}{e + u_a}(\frac{e_B}{e}) \\ & = & \displaystyle e_B(\frac{e + u_a}{(e + u_a)e}) \\ & = & \displaystyle\frac{e_B}{e} \end{array}$$

So by the minimality condition, we have

$$0 \leq \mathcal{P}(a, A, B) \leq \frac{e_B}{e}.$$

Hence, by equation (3),

$$1 = \sum_{B \in \mathcal{C} - \{A\}} \mathcal{P}(a, A, B) \leq \sum_{B \in \mathcal{C} - \{A\}} \frac{e_B}{e} = 1.$$

Therefore, equality holds. □

We can finally argue, as a corollary to Theorem 4.2, that well-typed programs satisfy probabilistic noninterference. For if O is well typed and $\mu \sim_\Gamma \nu$, then $(O, \mu) \sim_\Gamma (O, \nu)$; hence the probability of reaching any equivalence class from (O, μ) is the same as the probability of reaching it from (O, ν), and therefore the probability that the L variables end up with some values from (O, μ) is the same as the probability that they end up with those values from (O, ν); of course the *time* required to reach those values may differ.

For example, referring again to the example program of Figure 1, if we start with global configuration

$$\left(\left\{ \begin{array}{l} \alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1 \\ \beta = y := 1 \\ \gamma = y := 2 \end{array} \right\}, [x = 0, y = 0] \right)$$

then after three computation steps the configuration is either $(\{\ \}, [x = 0, y = 1])$ or $(\{\ \}, [x = 0, y = 2])$, each with probability 1/2.

But if we start with the equivalent global configuration

$$\left(\left\{ \begin{array}{l} \alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1 \\ \beta = y := 1 \\ \gamma = y := 2 \end{array} \right\}, [x = 5, y = 0] \right)$$

the program runs more slowly—after three computation steps there are five possible configurations, shown with their probabilities in Figure 5. Nevertheless, the final result is the same—after 13 steps, the configuration is either $(\{\ \}, [x = 0, y = 1])$ or $(\{\ \}, [x = 0, y = 2])$, each with probability 1/2.

$$(\{\alpha = x := x - 1; \textbf{while } x > 0 \textbf{ do } x := x - 1,\ \beta = y := 1,\ \gamma = y := 2\}, [x = 4, y = 0]) \quad : 1/27$$

$$(\{\alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1,\ \gamma = y := 2\}, [x = 4, y = 1]) \quad : 19/108$$

$$(\{\alpha = \textbf{while } x > 0 \textbf{ do } x := x - 1,\ \beta = y := 1\}, [x = 4, y = 2]) \quad : 19/108$$

$$(\{\alpha = x := x - 1; \textbf{while } x > 0 \textbf{ do } x := x - 1\}, [x = 5, y = 2]) \quad : 11/36$$

$$(\{\alpha = x := x - 1; \textbf{while } x > 0 \textbf{ do } x := x - 1\}, [x = 5, y = 1]) \quad : 11/36$$

Figure 5. Global configurations and their probabilities after three computation steps

5 Dynamic Thread Creation

The key condition that allows us to prove weak probabilistic bisimulation is equation (3), which says that, provided that it is possible to leave equivalence class A, the probability of leaving A eventually is 1. In consequence, we can allow programs to generate new threads, so long as they cannot be generated so quickly as to disturb equation (3).

Let us introduce a new command, **fork**$(c_1, \ldots, c_n)$, which terminates in one step but adds new threads c_1, ..., c_n to the thread pool.[3] Here is a typing rule for **fork**:

$$\frac{\Gamma \vdash c_1 : \tau\ \mathit{cmd}\ \_, \ldots, \Gamma \vdash c_n : \tau\ \mathit{cmd}\ \_}{\Gamma \vdash \textbf{fork}(c_1, \ldots, c_n) : \tau\ \mathit{cmd}\ 1}$$

With **fork** in the language, we no longer have the property that if global configuration $a' \in A$ is reachable from a, then $u_{a'} \leq u_a$. The reason is that the unessential threads of a could use **fork** to create more unessential threads. The question arises whether these additional threads could make the probability of leaving A eventually be less than 1.

But we can note that unessential threads cannot be generated too quickly. In particular, let n be the largest number of commands forked by any of the threads in global configuration a_0. Then if execution starts at a_0 and passes successively through global configurations a_1, a_2, a_3, ..., all in class A, then we can see that $u_{a_1} \leq u_{a_0} + n$, $u_{a_2} \leq u_{a_1} + n$, and so forth. If we let ϵ_i denote the probability of leaving class A at step i, $i \geq 0$, we see that

$$\epsilon_i \geq \frac{e}{e + u_{a_0} + in}$$

Now the probability of never leaving A is given by the infinite product

$$\prod_{i=0}^{\infty}(1 - \epsilon_i).$$

[3] Notice that this makes it awkward to model the thread pool O as a mapping from thread identifiers to commands, since it is unclear what the names of the newly-generated threads should be. It might be better, then, to follow [15] and to view the thread pool as a *multi-set* of commands.

By Theorem 12-55 of Apostol [4], this is equal to 0 iff

$$\sum_{i=0}^{\infty} \epsilon_i = \infty$$

This holds in our case, since we have

$$\sum_{i=0}^{\infty} \frac{e}{e + u_{a_0} + in} = \infty.$$

The point is that the probabilities of leaving A do not decrease quickly enough to give a nonzero probability of staying in A forever; this is the case so long as we can only fork a fixed number of threads in any computation step.

6 Conclusion

The notion of weak probabilistic bisimulation on Markov chains proposed in this paper gives a way of arguing for the equivalence of probabilistic systems that do not "run" at the same rate. It is applied in this paper to prove the soundness of the type system of [16], which allows the running times of threads to depend on the values of H variables, so long as these timing variations do not affect the values of L variables.

It would be interesting to extend the simple imperative language considered here with richer language constructs, such as arrays. Arrays are challenging because of the possibility of out-of-bounds indices. The simplest approach is to require that array indices be L, as in Agat's work [1], but it would be valuable to be more permissive. Also it would be interesting to consider a Java-like language with objected-oriented features; Banerjee and Naumann [6] treat such a language, but they do not consider threads. Finally, it would be valuable to explore further connections with the work of Honda et al. [9, 10] on secure information flow in the π-calulus.

7 Acknowledgments

I thank the anonymous referees and Zhenyue Deng for helpful comments, and I thank Ryan Yocum for developing the implementation used to compute the example probabilities in this paper [19]. This work was partially supported by the National Science Foundation under grant CCR-9900951.

References

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

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

[3] A. Aldini. Probabilistic information flow in a process algebra. In *Proc. CONCUR 2001 – Concurrency Theory*, pages 152–168. Lecture Notes in Computer Science2154, Aug. 2001.

[4] T. M. Apostol. *Mathematical Analysis*. Addison-Wesley, 1960.

[5] C. Baier and H. Hermanns. Weak bisimulation for fully probabilistic processes. In *Proc. Computer Aided Verification '97*, pages 119–130. Lecture Notes in Computer Science1254, 1997.

[6] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings 15th IEEE Computer Security Foundations Workshop*, pages 253–267, Cape Breton, Nova Scotia, Canada, June 2002.

[7] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. Technical Report 4254, INRIA, Sept. 2001.

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

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

[10] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings 29th Symposium on Principles of Programming Languages*, pages 81–92, Portland, Oregon, Jan. 2002.

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

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

[13] J. R. Norris. *Markov Chains*. Cambridge University Press, 1998.

[14] A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

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

[16] G. Smith. A new type system for secure information flow. In *Proceedings 14th IEEE Computer Security Foundations Workshop*, pages 115–125, Cape Breton, Nova Scotia, Canada, June 2001.

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

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

[19] R. Yocum. Type checking for secure information flow in a multi-threaded language. Master's thesis, Florida International University, 2002.

A Structural Operational Semantics

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

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

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

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

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

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

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

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

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

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

B Typing and Subtyping Rules

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

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

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

(ASSIGN) $$\frac{\Gamma(x) = \tau \ \mathit{var}, \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau \ \mathit{cmd}\ 1}$$

(SKIP) $$\Gamma \vdash \textbf{skip} : H \ \mathit{cmd}\ 1$$

(IF) $$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau \ \mathit{cmd}\ n \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}\ n}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau \ \mathit{cmd}\ n + 1}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \subseteq \tau_2 \quad \Gamma \vdash c_1 : \tau_2 \ \mathit{cmd}\ \tau_3 \quad \Gamma \vdash c_2 : \tau_2 \ \mathit{cmd}\ \tau_3}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau_2 \ \mathit{cmd}\ \tau_1 \vee \tau_3}$$

(WHILE) $$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \subseteq \tau_2 \quad \tau_3 \subseteq \tau_2 \quad \Gamma \vdash c : \tau_2 \ \mathit{cmd}\ \tau_3}{\Gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau_2 \ \mathit{cmd}\ \tau_1 \vee \tau_3}$$

(COMPOSE) $$\frac{\Gamma \vdash c_1 : \tau \ \mathit{cmd}\ m \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}\ n}{\Gamma \vdash c_1; c_2 \ : \ \tau \ \mathit{cmd}\ m + n}$$

$$\frac{\Gamma \vdash c_1 : \tau_1 \ \mathit{cmd}\ \tau_2 \quad \tau_2 \subseteq \tau_3 \quad \Gamma \vdash c_2 : \tau_3 \ \mathit{cmd}\ \tau_4}{\Gamma \vdash c_1; c_2 \ : \ \tau_1 \wedge \tau_3 \ \mathit{cmd}\ \tau_2 \vee \tau_4}$$

(PROTECT) $$\frac{\Gamma \vdash c : \tau_1 \ \mathit{cmd}\ \tau_2 \quad c \text{ contains no } \textbf{while} \text{ loops}}{\Gamma \vdash \textbf{protect}\ c : \tau_1 \ \mathit{cmd}\ 1}$$

(BASE) $$L \subseteq H$$

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

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

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

(REFLEX) $$\rho \subseteq \rho$$

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

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