Weak Probabilistic Bisimulation for Secure Information Flow

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

November 20, 2001

Abstract

This paper proposes a notion of *weak probabilistic bisimulation* for Markov chains and shows how it can be used to justify the correctness of a recently-published type system for secure information flow. The type system guarantees that well-typed multi-threaded programs running under a uniform probabilistic scheduler satisfy the probabilistic noninterference property.

1 Introduction

In [Smi01], a type system is presented that aims to ensure secure information flow in a simple multi-threaded imperative programming language running under a uniform probabilistic scheduler. The idea is to classify program variables as either L (public) or H (private) and to establish 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

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

and let thread β be

$$y := 1$$

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 or thread β with probability $1/2$. 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 following sequence of global configurations:

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

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—with probability 1, the final value of y is 1 regardless of the initial value of x. In this work (as in [Smi01] and [VS99]) 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 be. And, of course, preventing external leaks requires much more severe restrictions on programs—see for example [Aga00].

Returning to our example program, suppose that we change thread α to

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

In this case, the program is unsafe, even with respect to internal observations, because now there is a race between the assignments $y := 2$ in thread α and $y := 1$ in thread β, and the likely outcome of the race depends on the initial value of x. More precisely, the larger the initial value of x, the greater the probability that the final value of y is 2. Hence this program does not satisfy probabilistic noninterference.[1]

[1] Note that it does, however, satisfy *possibilistic noninterference* because it is *possible* for the final value of y to be either 1 or 2, regardless of the initial value of x.

The type system of [Smi01] 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 is 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.

The actual typing and subtyping rules are shown in Figures 1 and 2; more details can be found in [Smi01]. (Remarkably, Boudol and Castellani [BC01] have independently developed a type system almost identical to that of [Smi01], except that their system does not include types of the form τ *cmd* n.)

How can we argue that our type system guarantees probabilistic noninterference? First, we say that two memories μ and ν are equivalent, written $\mu \sim \nu$, if they agree on the values of L variables. 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 $\sim$ 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 Theorem 4.7 of [Smi01].) 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 type H *cmd* τ, which means that it will make no further assignments to L variables.

But when we consider a pool O of threads running concurrently, this variability in termination complicates the situation. 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 the equivalence of the Markov chains starting from (O, μ) and from (O, ν). As proposed by [SS00], this can be approached using the framework of *probabilistic bisimulation*, which we discuss abstractly in the next section.

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

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

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

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

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

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

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

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

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

Figure 1: Typing rules

(BASE) $L \subseteq H$

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

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

$$\tau \; cmd \; n \subseteq \tau \; cmd \; L$$

(REFLEX) $\rho \subseteq \rho$

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

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

Figure 2: Subtyping rules

2 Probabilistic Bisimulation

Given a finite or countably infinite Markov chain [KS60] 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* [KS60, LS91], which means that for all equivalence classes A, B, the probability of going (in one step) from a state $a \in A$ to some state in B is independent of a:

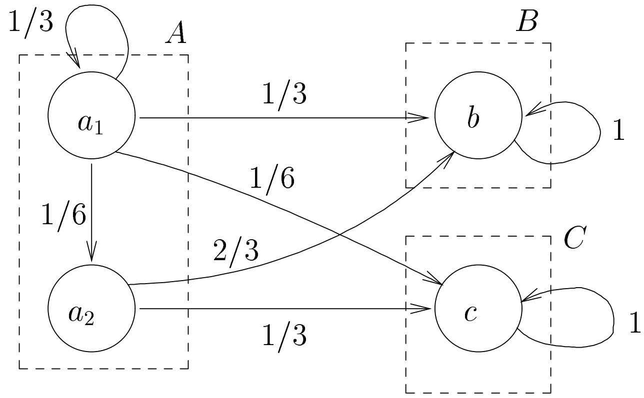
$$\sum_{b \in B} p_{ab} = \sum_{b \in B} p_{a'b},$$

for any $a' \in A$. We denote this common probability by p_{AB}.

However, this notion is quite strong with respect to timing, since 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 [Smi01] is to assume that the real running time of the program are 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 following Markov chain, 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 [BH97]. 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 [BH97], we observe that these probabilities solve the equation system:

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

For example, in the above Markov chain we have

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

and

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

which implies that

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

If, for all distinct equivalence classes A and B, $\mathcal{P}(a, A, B)$ is independent of the choice of a, then we define $\mathcal{P}(A, B)$ to be this unique value, and we say that $\approx$ is a *weak probabilistic bisimulation*. 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 [BH97]. Also, Aldini [Ald01] 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 $\approx$.

3 Noninterference via Weak Probabilistic Bisimulation

In [Smi01], the probabilistic noninterference property is established by defining an equivalence $\sim$ on global configurations (O, μ) and arguing that it is a probabilistic bisimulation. The definition used there says that $(O_1, \mu) \sim (O_2, \nu)$ iff $\mu \sim \nu$, $\mathit{dom}(O_1) = \mathit{dom}(O_2)$, and $O_1(\alpha) \sim O_2(\alpha)$ for all $\alpha \in \mathit{dom}(O_1)$. That is, the two thread pools must contain the same number of threads. Unfortunately this does not accord with the fact that equivalent commands, when run under equivalent memories, can terminate at different times. As a result, [Smi01] uses a strange semantics in which no threads are ever removed from the thread pool—completed threads remain alive performing **skip**s, wasting processor time.

But using the framework of weak probabilistic bisimulation described above, we can do better. We can restore the obvious semantics that removes completed threads from the thread pool by allowing equivalent global configurations to contain a different number of threads. More precisely, we relax the definition of $(O_1, \mu) \sim (O_2, \nu)$ so that O_1 and O_2 can each contain threads not contained in the other, provided that such threads have type $H\ \mathit{cmd}\ \tau$, which implies that they make no assignments to L variables.

With this relaxed definition, $\sim$ is no longer a probabilistic bisimulation, but it is a weak probabilistic bisimulation. The basic idea is that if $(O_1, \mu) \sim (O_2, \nu)$ and O_1 and O_2 both contain a thread α, then by definition we have $O_1(\alpha) \sim O_2(\alpha)$, which implies (by Theorem 4.7 of [Smi01]) 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 $H\ \mathit{cmd}\ \tau$ 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.

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

4 Conclusion

The framework of weak probabilistic bisimulation allows us to establish probabilistic noninterference even when the running times of threads can depend on the initial values of H variables. This framework should also be useful in treating an extended language that includes constructs for dynamic thread creation.

References

[Aga00] Johan Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, December 2000.

[Ald01] Alessandro Aldini. Probabilistic information flow in a process algebra. In *Proc. CONCUR 2001 – Concurrency Theory*, pages 152–168. Lecture Notes in Computer Science2154, August 2001.

[BC01] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. Technical Report 4254, INRIA, September 2001.

[BH97] Christel Baier and Holger Hermanns. Weak bisimulation for fully probabilistic processes. In *Proc. Computer Aided Verification '97*, pages 119–130. Lecture Notes in Computer Science1254, 1997.

[KS60] John Kemeny and J. Laurie Snell. *Finite Markov Chains*. D. Van Nostrand, 1960.

[LS91] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[Smi01] Geoffrey 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.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, UK, July 2000.

[VS99] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.