Improved typings for probabilistic noninterference in a multi-threaded language

Geoffrey Smith
School of Computing and Information Sciences, Florida International University, Miami, FL 33199, USA
E-mail: smithg@cis.fiu.edu

With the variables of a program classified as L (low, public) or H (high, private), the secure information flow problem is concerned with preventing the program from leaking information from H variables to L variables. In the context of a multi-threaded imperative language with probabilistic scheduling, the goal can be formalized as a *probabilistic noninterference* property. Previous work identified a type system sufficient to guarantee probabilistic noninterference, but at the cost of severe restrictions to prevent timing leaks – for example, H variables were forbidden in the guards of **while** loops. Here we present a more permissive type system that 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. The type system gives each command a type of the form τ_1 *cmd* τ_2; this type says that the command assigns only to variables of level τ_1 (or higher) and has running time that depends only on variables of level τ_2 (or lower). Also it uses types of the form τ *cmd* n for commands that terminate in exactly n steps. With these typings, timing leaks can be prevented by demanding that no assignment to an L variable may sequentially follow a command whose running time depends on H variables. We prove that well-typed multi-threaded programs satisfy a property that we call *weak probabilistic noninterference*; it is based on 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. Finally, we show 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 [18] for a comprehensive survey. In this paper, as in [22] and [23], we consider a simple multi-threaded imperative programming language in which each variable is classified either as L (low, public) or H (high, private). Our goal is to develop a static analysis that ensures that a program cannot leak the values of H variables. Of course, the possible ways of leaking information depend on what is observable. If we can observe the running program from the *outside*, seeing running time or the usage of various system resources, then controlling leaks is very difficult, because leaks can be based on low-level implementation details, such as caching behavior. Hence our focus, as in previous work, is on controlling *internal* leaks, in which information about H

variables is somehow transmitted to L variables. This makes the task more tractable, because we can control what is observable by the running program – for example, we can deny it access to a real-time clock.

More precisely, we wish to achieve *noninterference* [24] properties, which assert that changing the initial values of H variables cannot affect the final values of L variables. Given the nondeterminism associated with multi-threading, and our assumption that thread scheduling is probabilistic, we require more precisely that changing the initial values of H variables cannot affect the joint probability distribution of the possible final values of L variables; this property is called *probabilistic noninterference* [23].

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

while $x > 0$ **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 Fig. 1, where the annotation on the double arrow 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 [23]) 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 where we have downloaded some mobile code, which we do not trust fully, to execute on our machine. Here we have the advantage that the execution of the mobile code

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

Fig. 1. An example probabilistic execution.

cannot be observed externally (at the originating site, for example) unless we allow it to be. At any rate, 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

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

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 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. 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 a weaker property, *possibilistic* *noninterference*, because, regardless of the initial value of x, it is *possible* for the final value of y to be either 1 or 2.

A type system that guarantees possibilistic noninterference was given in [22]. Building on that work, a type system for probabilistic noninterference was given in [23]. These systems classify an expression e as H if it contains any H variables, and L otherwise. The latter system uses these classifications to impose the following restrictions:

1. Only L expressions can be assigned to L variables.
2. A guarded command with H guard cannot assign to L variables.
3. The guard of a **while** loop must be L.
4. An **if** with H guard must be *protected*, so that it executes atomically, and can contain no **while** loops within its branches.

Restrictions 1 and 2 prevent what Denning [8] long ago called *explicit* and *implicit* flows, respectively. In a language without concurrency, restrictions 1 and 2 are sufficient to guarantee noninterference – see for example [24]. Restrictions 3 and 4 were introduced to prevent timing-based flows in multi-threaded programs; unfortunately, they restrict the set of allowable programs quite severely.

A recent paper by Honda, Vasconcelos, and Yoshida [12] explores secure information flow in the π-calculus, showing in particular that the system of [22] can be embedded into their system. Most interestingly, they propose *enriching* the set of command types of [22] from

– $H\ cmd$, for commands that assign only to H variables and are guaranteed to terminate; and
– $L\ cmd$, for commands that assign to L variables or might not terminate,
– $\tau\ cmd \Downarrow$, for commands that assign only to variables of type τ (or higher) and are guaranteed to terminate; and
– $\tau\ cmd \Uparrow$, for commands that assign only to variables of type τ (or higher) and might not terminate.

They then argue that in some cases H variables can be used in the guards of **while** loops without sacrificing possibilistic noninterference.

Inspired by this suggestion, we can observe that the command typings used in [22] and [23] conflate two distinct issues: what does a command *assign to*, and what is the command's *running time*. This leads us to propose command types with *two* parameters to address these two issues separately. More precisely, our new system will classify the expressions and commands of a program as follows:

1. An expression e is classified as H if it contains any H variables; otherwise it is classified as L.
2. A command c is classified as $\tau_1\ cmd\ \tau_2$ 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).
3. A command c is classified as $\tau\ cmd\ n$ 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 imposes the following restrictions:

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

The third restriction replaces restrictions 3 and 4 of [23]; it more accurately prevents flows based on timing.

For example, the original thread α considered above,

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

is rejected by the type system of [23], simply because the guard of the **while** loop contains the H variable x. Under our new system, this thread is allowed, as it should be. But the dangerous modified thread α,

```
**while** $x > 0$ **do** $x := x - 1$;
$y := 2$
```

is rejected – because the running time of the **while** loop depends on the H variable x, it cannot be followed sequentially by an assignment to the L variable y.

Let's consider some other examples informally, assuming that $x : H$ and $y : L$.

```
1. $x := 0 : H\ cmd\ 1$
2. $y := 0 : L\ cmd\ 1$
3. **if** $x = 0$ **then** $x := 5$ **else skip** : $H\ cmd\ 2$
4. **while** $y = 0$ **do skip** : $H\ cmd\ L$
5. **while** $y = 0$ **do** $y := y - 1$ : $L\ cmd\ L$
6. **if** $x = 0$ **then**
       **while** $y = 0$ **do skip**
   **else**
       **skip** : $H\ cmd\ H$
7. $y := 5$; **while** $x + 1$ **do skip** : $L\ cmd\ H$
```

Example 3 shows that an **if** can have a H guard and nevertheless have a known running time; such a command can be sequentially followed by a L assignment without any problem.

Our typings also satisfy interesting subtyping rules. As usual, we have $L \subseteq H$. Furthermore, command types are contravariant in their first position, and covariant in their second position. Also, $\tau\ cmd\ n \subseteq \tau\ cmd\ L$, because if a command always halts in n steps, then its running time doesn't depend on the values of H variables. This subtyping rule implies that example 3 above also has type $H\ cmd\ L$.

The main result of this paper is that any multi-threaded program that is well typed under our typing rules satisfies a bisimulation-based property that we call *weak probabilistic noninterference*. We give an overview of our approach here.

First, we let Γ denote an *identifier typing*, which classifies program variables as L or H. Formally, Γ is a mapping from identifiers to types of the form $\tau\ var$. Next, we define the important property of *low-equivalence* of memories:

Definition 1.1. Memories μ and ν are low-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.

On individual commands, the key property ensured by the type system is that if a well-typed command c is run twice, under two low-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, which we also call $\sim_\Gamma$, on well-typed commands such that if low-equivalent commands c and d are run for a single step under low-equivalent memories μ and ν, then the results are equivalent in the sense that the resulting commands and memories are still low-equivalent. (This is the Sequential Noninterference Theorem, which will be proved in Section 5.) 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 $H\ cmd\ \tau$, which means that it will make no further assignments to L variables.

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 low-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 this context, a standard notion of equivalence is *probabilistic bisimulation* [15,19], which we review in Section 3.1. 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. For this reason, we introduce a notion of *weak probabilistic bisimulation*, which is more relaxed with respect to timing and which is suitable for our Markov chains; using this notion, we are able to prove the soundness of our typing rules.

The remainder of this paper is organized as follows. In Section 2, we define the multi-threaded language and its semantics. In Section 3, we define *weak probabilistic bisimulation* on Markov chains, and use this notion in defining our desired security property, which we call *weak probabilistic noninterference*. In Section 4, we define our type system formally. In Section 5, we establish the basic properties of the type system, culminating in the Sequential Noninterference Theorem. Then, in Section 6, we prove that every well-typed multi-threaded program satisfies weak probabilistic noninterference. In Section 7, we briefly show that we can accommodate an extended language with a **fork** command that allows us to spawn new threads dynamically. Finally, Section 8 discusses some related work and Section 9 concludes.

2. The multi-threaded language

Threads are written in the simple imperative language:

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

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 the (completely standard) structural operational semantics [10] shown in Fig. 2. In rule (ATOMICITY), note that (as usual) $\longrightarrow^*$ denotes the reflexive transitive closure of $\longrightarrow$. Also, we write $\longrightarrow^i$, with $i \geqslant 0$, to denote the i-fold self-composition of $\longrightarrow$.

Note that our sequential transition relation $\longrightarrow$ is *deterministic* and *total* (if some obvious restrictions are met):

Lemma 2.1. *If every identifier in c is in dom*(μ) *and no subcommand involving* **while** *is* **protect***ed in c, then there is a unique configuration C such that* $(c, \mu) \longrightarrow C$.

Also, the behavior of sequential composition is characterized by the following two lemmas, which have simple proofs by induction:

Lemma 2.2. *If* $(c_1, \mu) \longrightarrow^i \mu'$ *and* $(c_2, \mu') \longrightarrow^j \mu''$, *then* $(c_1; c_2, \mu) \longrightarrow^{i+j} \mu''$.

Lemma 2.3. *If* $(c_1; c_2, \mu) \longrightarrow^j \mu'$, *then there exist i and* μ'' *such that* $0 < i < j$, $(c_1, \mu) \longrightarrow^i \mu''$, *and* $(c_2, \mu'') \longrightarrow^{j-i} \mu'$.

We can further note that **while**-free commands are total:

Lemma 2.4. *If c contains no* **while** *loops and every identifier in c is in dom*(μ), *then there exists a unique* μ' *such that* $(c, \mu) \longrightarrow^* \mu'$.

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

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

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

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

Fig. 2. Structural operational semantics.

The multi-threaded programs that we consider here consist simply of a set of commands (the threads) running concurrently under a shared memory μ. We model this set as a *thread pool* O, which is a finite function from thread identifiers (α, β, $\gamma, \ldots$) to commands. A pair (O, μ), consisting of a thread pool and a shared memory, is called a *global configuration*.

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 $\overset{p}{\Longrightarrow}$ on global configurations; the rules are given in Fig. 3. The judgment $(O, \mu) \overset{p}{\Longrightarrow} (O', \mu')$ asserts that the probability of going from global configuration (O, μ) to (O', μ') is p. Note that $|O|$ denotes the number of threads in O, $O - \alpha$ denotes the thread pool obtained by removing thread α from O, and $O[\alpha := c']$ denotes the thread pool obtained by updating thread α to c'. The third rule (GLOBAL), which deals with an empty thread pool, is needed to make a multi-threaded program be a discrete Markov chain [9]. The states of the Markov chain are global configurations and the transition matrix is governed by $\overset{p}{\Longrightarrow}$.

(GLOBAL)
$$\frac{\begin{array}{l} O(\alpha) = c \\ (c, \mu) \longrightarrow \mu' \\ p = 1/|O| \end{array}}{(O, \mu) \stackrel{p}{\Longrightarrow} (O - \alpha, \mu')}$$

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

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

Fig. 3. Transitions on global configurations.

It should be acknowledged that building an efficient implementation of our language would be a challenge, because of the fine-grained interleaving specified by rule (GLOBAL). Clearly, ordinary real-time slicing would be inadequate, because the actual running times of commands will depend on low-level details (such as cache behavior) that are not modeled in our semantics. Also, rule (ATOMICITY) needs to be handled.

3. Our security property

In this section, we precisely define the *weak probabilistic noninterference* property that our type system aims to guarantee.

As defined in [23], a program satisfies *probabilistic noninterference* if the joint probability distribution of the final values of L variables is independent of the initial values of H variables. However, the effect of this definition is unclear in the case of programs that may not terminate – for such programs, the L variables may never get final values. The treatment of nontermination is especially important for our type system, because we allow H variables in the guards of **while** loops; as a result, the probability of nontermination can be affected by the initial values of H variables.

We therefore follow the approach of Sabelfeld and Sands [19] in defining our security property using bisimulation. We will say that thread pool O is secure provided that, for low-equivalent memories μ and ν, (O, μ) and (O, ν) are bisimilar, for an appropriate notion of probabilistic bisimilarity. Unlike [19], however, our notion of probabilistic bisimilarity needs to be *weak*, to allow the possibility that (O, μ) and (O, ν) do not run at the same speed.

Before proceeding further with the definition of our security property, we first pause to develop the needed notion of weak probabilistic bisimulation on Markov chains; we do this abstractly in the next subsection.

3.1. Probabilistic bisimulation on Markov chains

Given a finite or countably infinite Markov chain [9] 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.

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 [14, p. 124], who called it "lumpability"; Larsen and Skou [15] later called it "probabilistic bisimulation". More recently, Hermanns [11] includes a lengthy discussion of probabilistic bisimulation, and Sabelfeld and Sands [19] were the first to apply it to the information flow problem.

Note, however, that the probabilistic bisimulation 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.

As a result, this condition is not suitable for our type system, which allows the running times of threads to depend on the values of H variables. Instead, we need 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 Fig. 4, 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 see 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. We can now define *weak probabilistic bisimulation*:

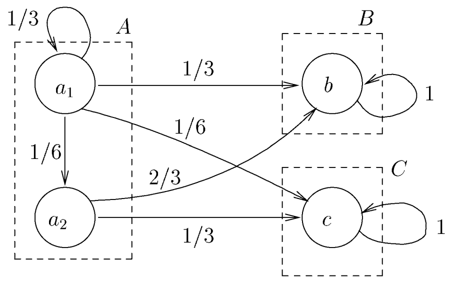

Fig. 4. An example weak probabilistic bisimulation.

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.[1] 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 $\approx$.

We now discuss techniques for calculating the probabilities $\mathcal{P}(a, A, B)$. 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 Markov chain in Fig. 4 we have

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

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

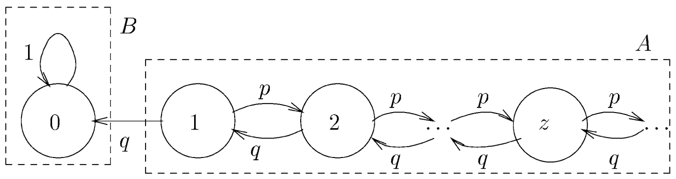

Fig. 5. A random walk Markov chain.

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

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

Calculating the probabilities $\mathcal{P}(a, A, B)$ is, unfortunately, more subtle in general than is suggested by this example. 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 [9], as shown in Fig. 5. (Here p and q can be any numbers satisfying $p, q \geqslant 0$ and $p + q = 1$.) In this case, equation system (2) specializes to

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

Now it is easy to see that

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

solves the equation system. But

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

also solves the equation system, provided that $p > 0$. In fact, Feller [9] shows that the actual probabilities are

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

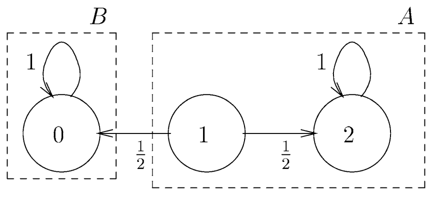

Fig. 6. A finite Markov chain with multiple solutions to equation system (2).

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 Fig. 6. In this case, equation system (2) specializes to

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

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 [17]:

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

We are now ready to define our security property precisely.

3.2. Weak probabilistic noninterference

As indicated above, we will say that O is secure provided that, for any low-equivalent memories μ and ν, $(O, \mu) \approx (O, \nu)$ for some weak probabilistic bisimulation $\approx$. However, we cannot allow $\approx$ to be just *any* weak probabilistic bisimulation. In particular, the universal relation (which puts all global configurations into the same equivalence class) is trivially a weak probabilistic bisimulation, but it is not suitable because it equates (O_1, μ) and (O_2, ν) even if μ and ν are not low-equivalent. This motivates the following definition:

Definition 3.2. Let $\approx$ be an equivalence relation on the space of global configurations (O, μ). We say that $\approx$ is a *weak probabilistic low bisimulation* if it is a weak probabilistic bisimulation such that $(O_1, \mu) \approx (O_2, \nu)$ implies that $\mu \sim_\Gamma \nu$. We further say that two global configurations are *weak probabilistic low bisimilar* if they are related by some weak probabilistic low bisimulation.

Finally, we can define our security property:

Definition 3.3. Thread pool O satisfies *weak probabilistic noninterference* if, for all low-equivalent memories μ and ν, (O, μ) and (O, ν) are weak probabilistic low bisimilar.

One way of justifying this definition is to consider what it implies about halting programs. To this end, let us say that thread pool O is *probabilistically total* if, for any memory μ, (O, μ) eventually reaches a terminal configuration (i.e. one with an empty thread pool) with probability 1. For example, thread pool

$$\{\alpha = \textbf{while } x \textbf{ do skip}, \beta = x := 0\}$$

is probabilistically total. Then we have the following theorem:

Theorem 3.2. *If O is probabilistically total and satisfies weak probabilistic noninterference, then the joint probability distribution on the final values of L variables is independent of the initial value of H variables.*

Proof. Under the hypotheses, if $\mu \sim_\Gamma \nu$, then (O, μ) and (O, ν) are weak probabilistic low bisimilar; 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.) $\square$

Now we are ready to present our type system formally.

4. The type system

Our type system is based upon the following types:

$(\textit{data types})\quad \tau ::= L \mid H$
$(\textit{phrase types})\ \rho ::= \tau \mid \tau\ \textit{var} \mid \tau_1\ \textit{cmd}\ \tau_2 \mid \tau\ \textit{cmd}\ n$

We write $\tau\ cmd\ \_$ when we don't care about the "running time" component of the type. For simplicity, we consider just two security levels, L and H, but it is straightforward to generalize to an arbitrary lattice of security levels.

The rules of the type system are given in Figs 7 and 8. In the rules (IF), (WHILE), and (COMPOSE), $\vee$ denotes *join* (*least upper bound*) and $\wedge$ denotes *meet* (*greatest lower bound*). The rules allow us to prove *typing judgments* of the form $\Gamma \vdash p : \rho$ as well as *subtyping judgments* of the form $\rho_1 \subseteq \rho_2$. (As mentioned in the Introduction, Γ denotes an *identifier typing*, mapping identifiers to phrase types of the form $\tau\ var$.) As usual, we say that phrase p is *well typed* under Γ if $\Gamma \vdash p : \rho$ for some ρ. Similarly, thread pool O is well typed under Γ if each thread in O is well typed under Γ.

Remarkably, Boudol and Castellani [7] independently developed a type system almost identical to this one, except that their system does not include types of the form $\tau\ cmd\ n$.

As an example, we show the derivation of the typing of example 7 from the Introduction:

$$\Gamma \vdash y := 5; \textbf{while}\ x + 1\ \textbf{do skip} : L\ cmd\ H,$$

assuming that $\Gamma(x) = H\ var$ and $\Gamma(y) = L\ var$. We have

$$\Gamma \vdash 5 : L \tag{3}$$

by rule (INT). Then we get

$$\Gamma \vdash y := 5 : L\ cmd\ 1 \tag{4}$$

by rule (ASSIGN) on (3), and

$$\Gamma \vdash y := 5 : L\ cmd\ L \tag{5}$$

by rule (SUBSUMP) on (4) using the third rule (CMD). Next we have

$$\Gamma \vdash x : H \tag{6}$$

from rule (R-VAL) and

$$\Gamma \vdash 1 : L \tag{7}$$

by rule (INT), which gives

$$\Gamma \vdash 1 : H \tag{8}$$

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

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

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

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

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

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

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

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

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

Fig. 7. Typing rules.

(BASE) $L \subseteq H$

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

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

$$\tau\, cmd\, n \subseteq \tau\, cmd\, L$$

(REFLEX) $\rho \subseteq \rho$

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

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

Fig. 8. Subtyping rules.

by rule (SUBSUMP) on (7) using rule (BASE), and

$$\Gamma \vdash x + 1 : H \tag{9}$$

by rule (SUM) on (6) and (8). Next

$$\Gamma \vdash \textbf{skip} : H\, cmd\, 1 \tag{10}$$

by rule (SKIP), and hence

$$\Gamma \vdash \textbf{skip} : H\, cmd\, L \tag{11}$$

by rule (SUBSUMP) on (10) using the third rule (CMD). Hence we get

$$\Gamma \vdash \textbf{while}\ x + 1\ \textbf{do skip} : H\, cmd\, H \tag{12}$$

by rule (WHILE) on (9) and (11), since $H \subseteq H$ (by rule (REFLEX)) and $L \subseteq H$ (by rule (BASE)), and since $H \vee L = H$. And finally, we get

$$\Gamma \vdash y := 5; \textbf{while}\ x + 1\ \textbf{do skip} : L\, cmd\, H \tag{13}$$

by the second rule (COMPOSE) on (5) and (12), since $L \subseteq H$, $L \wedge H = L$, and $L \vee H = H$.

We now give some discussion of the typing rules. The first (IF) rule says that an **if** statement takes $n+1$ steps if both its branches take n steps. This rule can sometimes be used to "pad" a command to eliminate timing leaks, as in the transformation approach proposed by Johan Agat [1]. For example, if $x : H$ and $y : L$, then the thread

```
**if** $x = 0$ **then**
    $x := x * x;\ x := x * x$
**else**
    $x := x + 1;$
$y := 0$
```

is dangerous, because the time at which y is assigned 0 depends on the value of x. And this program is not well typed under our rules – the **then** branch of the **if** has type $H\ \mathit{cmd}\ 2$ and the **else** branch has type $H\ \mathit{cmd}\ 1$, which means that the first (IF) rule does not apply. Instead we must coerce the two branches to type $H\ \mathit{cmd}\ L$ and use the second (IF) rule, which gives the **if** type $H\ \mathit{cmd}\ H$. But this makes it illegal (under the second rule (COMPOSE)) to sequentially compose the **if** with the assignment $y := 0$, which has type $L\ \mathit{cmd}\ 1$, and $H \not\subseteq L$. To make the program well typed, we can pad the **else** branch to $x := x + 1;$ **skip**, which has type $H\ \mathit{cmd}\ 2$. Now we can type the **if** using the first (IF) rule, giving it type $H\ \mathit{cmd}\ 3$, and then we can give the thread type $L\ \mathit{cmd}\ 4$, using the first rule (COMPOSE). It should be noted, however, that Agat's transformation approach is more general than what we can achieve here.

The second rule (IF) is rather complex. One might hope that we could exploit subtyping to simplify the rule, but this is not possible here. We would not want to coerce the type of e up to τ_2, because then it would appear that the execution time of the **if** depends on τ_2 variables. Nor would we want to coerce the types of c_1 and c_2 to $\tau_1\ \mathit{cmd}\ \tau_3$, because then it would appear that the **if** can assign to τ_1 variables.

The constraint $\tau_3 \subseteq \tau_2$ in rule (WHILE) can be understood by considering that the command **while** e **do** c implicitly involves sequential composition of c and the entire loop, as shown in the second rule (LOOP). As a result, if c's running time depends on H variables, then c must not assign to L variables. For example, if x is H and y is L, then without the constraint $\tau_3 \subseteq \tau_2$ in rule (WHILE), the following program would be well typed:

```
**while** $1$ **do**
    $(y := y + 1;$ **while** $x$ **do skip**$)$
```

Note that $y := y + 1$ has type $L\ \mathit{cmd}\ L$ and **while** x **do skip** has type $H\ \mathit{cmd}\ H$, so the loop body has type $L\ \mathit{cmd}\ H$. Hence, without the constraint $\tau_3 \subseteq \tau_2$, the **while** loop could be given type $L\ \mathit{cmd}\ H$. But the loop is dangerous – if $x \neq 0$, then y is incremented only once, and if $x = 0$, then y is incremented repeatedly.

Finally, we note that **protect** c takes a command that is guaranteed to terminate and makes it appear to run in just one step. This gives another way of dealing with the example program discussed above; rather than padding the **else** branch, we can just **protect** the **if** (or just its **then** branch), thereby masking any timing differences resulting from different values of x.

5. Properties of the type system

In this section, we formally establish the properties of the type system. We begin with two lemmas describing the expressiveness of our type system. First, we show that our type system does not restrict a thread at all unless the thread involves both L and H variables:

Lemma 5.1. *Any command involving only L variables has type L cmd L. Any command involving only H variables has type H cmd H.*

Note this is certainly not the case for the type system of [23], since (for example) that system disallows H variables in the guards of **while** loops.

Next we show that our type system is strictly less restrictive than that of [23]. Recall that that system requires that each command c be both well typed and *protected*, which means that any **if** commands with H guards within c are wrapped with **protect**, so that they execute atomically.

Lemma 5.2. *If c has type τ cmd and is protected under the type system of* [23]*, then c has type τ cmd L under our type system.*

Proof. By induction on the structure of c. □

Now we turn to the question of the soundness of the type system, beginning with some standard lemmas.

Lemma 5.3 (Simple Security). *If $\Gamma \vdash e : L$, then e contains only L variables.*

Proof. By induction on the structure of e. □

Lemma 5.4 (Confinement). *If $\Gamma \vdash c : H\,cmd\,\_$, then c does not assign to any L variables.*

Proof. By induction on the structure of c. □

Now we have a slightly unusual Subject Reduction lemma, because on commands of type τ *cmd* n, the running time "counts down" as we execute:

Lemma 5.5 (Subject Reduction). *Suppose $(c, \mu) \longrightarrow (c', \mu')$. If $\Gamma \vdash c : \tau_1\,cmd\,\tau_2$, then $\Gamma \vdash c' : \tau_1\,cmd\,\tau_2$. And if $\Gamma \vdash c : \tau\,cmd\,n$, where $n > 1$, then $\Gamma \vdash c' : \tau\,cmd\,(n-1)$.*

Proof. By induction on the structure of c.

First, under the hypothesis that $(c, \mu) \longrightarrow (c', \mu')$, c cannot be of the form $x := e$, **skip**, or **protect** c'.

If c is of the form **if** e **then** c_1 **else** c_2, then c' is either c_1 or c_2. Now, if c has type $\tau\ cmd\ n$, then it must be typed by the first rule (IF), which implies that both c_1 and c_2 have type $\tau\ cmd\,(n-1)$. And if c has type $\tau_1\ cmd\ \tau_2$, then it is typed either with the first rule (IF) (using the fact that $\tau_1\ cmd\ m \subseteq \tau_1\ cmd\ \tau_2$), or with the second rule (IF). In the first case, c_1 and c_2 have type $\tau_1\ cmd\ m$, which implies that they also have type $\tau_1\ cmd\ \tau_2$. In the second case, c_1 and c_2 have type $\tau_1\ cmd\ \tau_3$, for some τ_3 with $\tau_3 \subseteq \tau_2$. So they have type $\tau_1\ cmd\ \tau_2$ as well, by rule (SUBSUMP).

If c is of the form **while** e **do** c_1, then c' is of the form $c_1; c$. In this case, c cannot have type $\tau\ cmd\ n$; it must have type $\tau_1\ cmd\ \tau_2$ by rule (WHILE). Hence c_1 has type $\tau_1\ cmd\ \tau_3$ for some τ_3 with $\tau_3 \subseteq \tau_2$ and $\tau_3 \subseteq \tau_1$. Therefore, by the second rule (COMPOSE), $c_1; c$ has type $\tau_1\ cmd\ \tau_2$. (Note that this last step would fail without the constraint $\tau_3 \subseteq \tau_1$.)

Finally, if c is of the form $c_1; c_2$, then c' is either c_2 (if $(c_1, \mu) \longrightarrow \mu'$) or $c_1'; c_2$ (if $(c_1, \mu) \longrightarrow (c_1', \mu')$). If c has type $\tau\ cmd\ n$, then it must be typed by the first rule (COMPOSE) which means that c_1 has type $\tau\ cmd\ k$ and c_2 has type $\tau\ cmd\ l$ for some k and l with $k + l = n$. If c' is c_2, then we must have $k = 1$, so c_2 has type $\tau\ cmd\,(n-1)$. If c' is $c_1'; c_2$, then by induction c_1' has type $\tau\ cmd\,(k-1)$, and therefore c' has type $\tau\ cmd\,(k-1+l) = \tau\ cmd\,(n-1)$. And if c has type $\tau_1\ cmd\ \tau_2$, then it must be typed by the second rule (COMPOSE) which means that c_1 has type $\tau_3\ cmd\ \tau_4$ and c_2 has type $\tau_5\ cmd\ \tau_6$, for some τ_3, τ_4, τ_5, and τ_6 satisfying the subtyping constraints $\tau_4 \subseteq \tau_5$, $\tau_4 \subseteq \tau_2$, $\tau_6 \subseteq \tau_2$, $\tau_1 \subseteq \tau_3$, and $\tau_1 \subseteq \tau_5$. Now, if c' is c_2, then by rule (SUBSUMP) it also has type $\tau_1\ cmd\ \tau_2$, since $\tau_5\ cmd\ \tau_6 \subseteq \tau_1\ cmd\ \tau_2$. And if c' is $c_1'; c_2$, then by induction c_1' has type $\tau_3\ cmd\ \tau_4$, and therefore c' has type $\tau_1\ cmd\ \tau_2$. □

Lemma 5.6. *If $\Gamma \vdash c : \tau\ cmd\ 1$ and $dom(\mu) = dom(\Gamma)$, then $(c, \mu) \longrightarrow \mu'$ for some μ'.*

Proof. It is easy to see that no commands have type $\tau\ cmd\ 0$. Hence the only commands with type $\tau\ cmd\ 1$ are $x := e$, **skip**, and **protect** c_1. The result is immediate in the first two cases. In the case of **protect** c_1, note that rule (PROTECT) requires c_1 to be free of **while** loops; hence by Lemma 2.4 c_1 is guaranteed to terminate. □

We now explore the behavior of a well-typed command c when run under two low-equivalent memories. We begin with a Mutual Termination lemma for **while**-free programs:

Lemma 5.7 (Mutual Termination). *Suppose that c is well typed under Γ, c contains no* **while** *loops, and $\mu \sim_\Gamma \nu$. Then there exist unique memories μ' and ν' such that $(c, \mu) \longrightarrow^* \mu'$, $(c, \nu) \longrightarrow^* \nu'$, and $\mu' \sim_\Gamma \nu'$.*

Proof. By induction on the structure of c, using Lemma 2.4. □

In the context of multi-threaded programs, however, it is not enough to consider only the final memory resulting from the execution of c (as in the Mutual Termination lemma); we must also consider timing. The key property that lets us establish weak probabilistic noninterference is this: if a well-typed command c is run under two low-equivalent memories, it makes exactly the same sequence of assignments to L variables, *at the same times*. Hence the two memories will remain low-equivalent after every execution step. (Note however that c may terminate more slowly, or even not terminate at all, under one of the two memories. But then the slower execution will not make any more assignments to L variables.)

Here's an example that illustrates some of the working of the type system. Suppose that c is a well-typed command of the form

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2); c_3.$$

What happens when c is run under two low-equivalent memories μ and ν? If $e : L$, then $\mu(e) = \nu(e)$ by Simple Security, and hence both executions will choose the same branch. If, instead, $e : H$, then the two executions may choose different branches. But if the **if** is typed using the first rule (IF), then both branches have type $H\ cmd\ n$ for some n. Therefore neither branch assigns to L variables, by Confinement, and both branches terminate after n steps, by Subject Reduction. Hence both executions will reach c_3 at the same time, and the memories will still be low-equivalent. And if the **if** is typed using the second rule (IF), then it gets type $H\ cmd\ H$, which is also the type given to each branch. Again, neither branch assigns to L variables, by Confinement. Now, in this case the two branches may not terminate at the same time – indeed, one may terminate and the other may not. But the entire command (**if** e **then** c_1 **else** c_2); c_3 will have to be typed by the second rule (COMPOSE), which means that $c_3 : H\ cmd\ H$. Hence c_3 makes no assignments to L variables, which means that, as far as L variables are concerned, it doesn't matter when (or even whether) c_3 is executed.

To formalize these ideas, we need to define a notion of low-equivalence on configurations; then we can argue that $\longrightarrow$ takes low-equivalent configurations to low-equivalent configurations. But first we make some observations about sequential composition. Any command c can be written in the *standard form*

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

for some $k \geqslant 1$, 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

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

Now, if c is executed, it follows from the (SEQUENCE) rules that the first execution step touches only c_1; that is, we have either

$$(c_1; c_2; \ldots; c_k, \mu) \longrightarrow (c_2; \ldots; c_k, \mu'),$$

if $(c_1, \mu) \longrightarrow \mu'$, or else

$$(c_1; c_2; \ldots; c_k, \mu) \longrightarrow (c_1'; c_2; \ldots; c_k, \mu'),$$

if $(c_1, \mu) \longrightarrow (c_1', \mu')$. Now we define our notion of low-equivalence on commands:

Definition 5.1. We say that commands c and d are low-equivalent with respect to Γ, written $c \sim_\Gamma d$, if c and d are both well typed under Γ and either

1. $c = d$,
2. c and d both have type $H\ cmd\ \_$, or
3. c is of the form $c_1; c_2; \ldots; c_k$, d is of the form $d_1; c_2; \ldots; c_k$, 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 low-equivalence to configurations by saying that configurations C and D are low-equivalent, written $C \sim_\Gamma D$, if any of the following four cases applies:

1. C is of the form (c, μ), D is of the form (d, ν), $c \sim_\Gamma d$, and $\mu \sim_\Gamma \nu$.
2. C is of the form (c, μ), D is of the form ν, c has type $H\ cmd\ \_$, and $\mu \sim_\Gamma \nu$.
3. C is of the form μ, D is of the form (d, ν), d has type $H\ cmd\ \_$, and $\mu \sim_\Gamma \nu$.
4. 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 of type $H\ cmd\ \_$ is low-equivalent to a terminated command.) Now we are ready to show the key Sequential Noninterference result:

Theorem 5.8 (Sequential Noninterference). *If $(c, \mu) \sim_\Gamma (d, \nu)$, $(c, \mu) \longrightarrow C'$, and $(d, \nu) \longrightarrow D'$, then $C' \sim_\Gamma D'$.*

Proof. We begin by dealing with the case when c and d both have type $H\ cmd\ \_$. In this case, by the Confinement Lemma, neither c nor d assigns to any L variables. Hence the memories of C' and D' remain low-equivalent. Now, if C' is of the form (c', μ') and D' is of the form (d', ν'), then by the Subject Reduction Lemma, c' and d' both have type $H\ cmd\ \_$, which gives $c' \sim_\Gamma d'$. The cases when C' is of the form μ' and/or D' is of the form ν' are similar.

Now consider the case where c and d do not both have type $H\ cmd\ \_$. We can see from the definition of $\sim_\Gamma$ that either $c = d$, or else c is of the form $c_1; c_2; \ldots; c_k$, d

is of the form $d_1; c_2; \ldots ; c_k$, for some k, and c_1 and d_1 both have type $H\ \mathit{cmd}\ n$, for some n.

In the latter case, we have by the Confinement Lemma that neither c_1 nor d_1 assigns to any L variables. Hence the memories of C' and D' are low-equivalent. And if $n > 1$, then by the Subject Reduction Lemma, C' and D' are of the form $(c'_1; c_2; \ldots ; c_k, \mu')$ and $(d'_1; c_2; \ldots ; c_k, \nu')$, respectively, where c'_1 and d'_1 both have type $H\ \mathit{cmd}\,(n-1)$. Thus $C' \sim_\Gamma D'$. And if $n = 1$, then C' and D' are of the form $(c_2; \ldots ; c_k, \mu')$ and $(c_2; \ldots ; c_k, \nu')$, respectively.[2] So again $C' \sim_\Gamma D'$.

We are left, finally, with the case where $c = d$. Let them have standard form $c_1; c_2; \ldots ; c_k$ and consider in turn each of the possible forms of c_1:

Case $x := e$**.** In this case, we have that C' is

$$(c_2; \ldots ; c_k, \mu[x := \mu(e)])$$

and D' is

$$(c_2; \ldots ; c_k, \nu[x := \nu(e)]).$$

Now if x is H, then $\mu[x := \mu(e)] \sim_\Gamma \nu[x := \nu(e)]$. And if x is L, then by rule (ASSIGN) $e : L$. Hence $\mu(e) = \nu(e)$ by Simple Security, so again $\mu[x := \mu(e)] \sim_\Gamma \nu[x := \nu(e)]$. Therefore $C' \sim_\Gamma D'$.

Case skip. In this case C' is $(c_2; \ldots ; c_k, \mu)$ and D' is $(c_2; \ldots ; c_k, \nu)$. So $C' \sim_\Gamma D'$.

Case if e **then** c_{11} **else** c_{12}**.** If $e : L$, then by Simple Security $\mu(e) = \nu(e)$. Hence C' and D' both choose the same branch. That is, either

$$C' = (c_{11}; c_2; \ldots ; c_k, \mu)$$

and

$$D' = (c_{11}; c_2; \ldots ; c_k, \nu),$$

or else

$$C' = (c_{12}; c_2; \ldots ; c_k, \mu)$$

and

$$D' = (c_{12}; c_2; \ldots ; c_k, \nu).$$

So $C' \sim_\Gamma D'$.

[2]Actually, if $k = 1$ then C' and D' are just μ' and ν' here. We'll ignore this point in the rest of this proof.

If e doesn't have type L, then if c_1 is typed by the first rule (IF), then we have that c_{11} and c_{12} both have type $H\ cmd\ n$ for some n. Therefore, whether or not C' and D' take the same branch, we have $C' \sim_\Gamma D'$.

And if c_1 is typed by the second rule (IF), than it gets type $H\ cmd\ H$. But, by rule (COMPOSE), this means that $c_1; c_2$ also has type $H\ cmd\ H$. This in turn implies that $c_1; c_2; c_3$ has type $H\ cmd\ H$, and so on, until we get that c has type $H\ cmd\ H$. So, since $c = d$ here, this case has already been handled.

Case while e **do** c_{11}**.** As in the case of **if**, if $e : L$, then by Simple Security $\mu(e) = \nu(e)$. Hence the two computations stay together. That is, either $C' = (c_2; \ldots; c_k, \mu)$ and $D' = (c_2; \ldots; c_k, \nu)$, or else

$$C' = (c_{11}; \textbf{while}\ e\ \textbf{do}\ c_{11}; c_2; \ldots; c_k, \mu)$$

and

$$D' = (c_{11}; \textbf{while}\ e\ \textbf{do}\ c_{11}; c_2; \ldots; c_k, \nu).$$

So $C' \sim_\Gamma D'$.

If e doesn't have type L, then by rule (WHILE) we have that $c_1 : H\ cmd\ H$. As in the case of **if**, this implies that $c : H\ cmd\ H$, so this case has again already been handled.

Case protect c_{11}**.** By rule (PROTECT), c_{11} contains no **while** loops. Hence this case follows from the Mutual Termination lemma. □

We remark here that if c is well typed and $\mu \sim_\Gamma \nu$, then $(c, \mu) \sim_\Gamma (c, \nu)$. Hence, by applying the Sequential Noninterference Theorem repeatedly, we see that, for all k, the configuration reached from (c, μ) after k steps will be low-equivalent to the configuration reached from (c, ν) after k steps. Hence, as we claimed above, the two executions make exactly the same assignments to L variables, at the same times.

Now we are ready to change our focus from the execution of an *individual* thread c, which is deterministic, to the execution of a *pool* of threads O, which is a Markov chain. In the next section, we prove the soundness of our type system by showing that every well-typed thread pool O satisfies *weak probabilistic noninterference*, as defined in Section 3.

6. Well-typed thread pools satisfy weak probabilistic noninterference

Suppose that O is a well-typed thread pool. To show that O satisfies weak probabilistic noninterference, as defined in Definition 3.3, we must show that for all low-equivalent memories μ and ν, (O, μ) and (O, ν) are weak probabilistic low bisimilar. To do this, we first construct a suitable weak probabilistic low bisimulation, which we again call $\sim_\Gamma$.

Recall that we have already defined $\sim_\Gamma$ on well-typed commands in Definition 5.1. We use that definition in extending $\sim_\Gamma$ to well-typed 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 type $H\ cmd\ \_$, making them unimportant as far as L variables are concerned.

In addition, under Definition 3.2, $\sim_\Gamma$ must also be defined on global configurations that are *not* well typed. We deal with this by simply letting $\sim_\Gamma$ be the *identity relation* on such configurations. (In contrast, Sabelfeld and Sands [19] address this issue by instead using *partial* probabilistic low bisimulations.)

Definition 6.1. $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ iff either

- $O_1 = O_2$ and $\mu = \nu$, or
- O_1 and O_2 are well typed and
  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 $H\ cmd\ \_$ for all $\alpha \in dom(O_1) - dom(O_2)$, and
  4. $O_2(\alpha)$ has type $H\ cmd\ \_$ for all $\alpha \in dom(O_2) - dom(O_1)$.

Now we must argue that $\sim_\Gamma$ is a weak probabilistic low bisimulation, as defined in Definition 3.2. It is immediate from our definition that $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ implies that $\mu \sim_\Gamma \nu$, so the only challenge is to show that $\sim_\Gamma$ is a weak probabilistic bisimulation, as defined in Definition 3.1. This requires us to show that if $(O_1, \mu) \sim_\Gamma (O_2, \nu)$, then (O_1, μ) and (O_2, ν) have equal probabilities of eventually leaving the current equivalence class and going to any other equivalence class. In the case of non-well-typed global configurations, this holds trivially, since their equivalence classes are singletons. And for well-typed global configurations, 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 $H\ cmd\ \_$ 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 eventually going from (O_1, μ) to any other equivalence class.

Before giving the proof, we introduce some terminology. Let A be an equivalence class of $\sim_\Gamma$ and let (O, μ) be a well-typed global configuration in A. A thread $\alpha \in O$ is said to be *essential* if running α for a step takes us to a global configuration in an equivalence class different from A. Otherwise α is *unessential*. For example, recall the program shown in Fig. 1, where x is H and y is L. Consider the global configuration

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

This global configuration contains two essential threads (β and γ) and one unessential thread (α). Note that β takes us to 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 γ takes us to the equivalence class of

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

We also need the following lemma:

Lemma 6.1. *Every unessential thread has type H cmd _.*

Proof. Suppose that (O, μ) contains an unessential thread $\alpha = c$. If $(c, \mu) \longrightarrow \mu'$, then executing α takes (O, μ) to $(O - \alpha, \mu')$. Under Definition 6.1, $(O, \mu) \sim_\Gamma (O - \alpha, \mu')$ only if $c : H\ cmd\ \_$. If, instead, $(c, \mu) \longrightarrow (c', \mu')$, then executing α takes (O, μ) to $(O[\alpha := c'], \mu')$. Now $(O, \mu) \sim_\Gamma (O[\alpha := c'], \mu')$ only if $c \sim_\Gamma c'$. Under Definition 5.1, this holds only if either

1. $c = c'$,
2. $c : H\ cmd\ \_$, or
3. c is of the form $c_1; c_2; \ldots; c_k$, c' is of the form $c_1'; c_2; \ldots; c_k$, and both c_1 and c_1' have type $H\ cmd\ n$ for some n.

But it is easy to see that option 1 is impossible under the rules in Fig. 2. Also, option 3 is impossible, since we know from Subject Reduction that c_1' will actually have type $H\ cmd\ (n - 1)$. □

Theorem 6.2. *Relation $\sim_\Gamma$ is a weak probabilistic low bisimulation on the space of global configurations.*

Proof. As discussed above, it is immediate that $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ implies that $\mu \sim_\Gamma \nu$.

So it remains only to show that if A and B are distinct equivalence classes of $\sim_\Gamma$, then $\mathcal{P}(a, A, B)$ is independent of the choice of a. Consider any state a in A; of course a is actually a global configuration (O, μ). Note that if O is not well typed, then A is a singleton, which gives the result trivially. So we are left with the case where A contains only well-typed global configurations.

Suppose that A contains (O, μ) and that O contains an essential thread α that takes us to some equivalence class C other than A. Then thread α cannot have type H *cmd* _, since otherwise running α would keep us in class A. Therefore (by Definition 6.1) *every* global configuration in A must contain an equivalent thread α which (by the Sequential Noninterference Theorem) must also take us to equivalence class C. The conclusion is that every global configuration in A must contain the same number e of essential threads that take us out of class A, and also the same number e_C of threads that take us to equivalence class C.

In addition, each global configuration $a \in A$ contains some number u_a of unessential threads whose execution leaves us within class A. By the lemma above, such threads have type H *cmd* _. Different global configurations in A can have different numbers of unessential threads, but note that if global configuration $a' \in A$ is reachable from a, then $u_{a'} \leqslant u_a$, since 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 $\mathcal{P}(a, A, B) = e_B/e$.

To justify this, first note that if we start from a, then the probability of leaving A in one step is $e/(e + u_a)$. So, remembering that $u_{a'} \leqslant u_a$ for any a' reachable from a, we see that the probability of *not* leaving A after k steps is at most $(u_a/(e+u_a))^k$, 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 $(e + u_a)/e$.) Hence, if we let $\mathcal{C}$ denote the set of all equivalence classes of $\sim_\Gamma$, we have

$$\sum_{B \in \mathcal{C} - \{A\}} \mathcal{P}(a, A, B) = 1. \qquad (14)$$

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

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

solves the equation system:

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

So by the minimality condition, we have

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

Hence, by equation (14),

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

Therefore, equality holds.

So $\sim_\Gamma$ is a weak probabilistic low bisimulation. □

Corollary 6.3. *Every well-typed thread pool O satisfies weak probabilistic noninterference.*

Proof. If O is well typed and $\mu \sim_\Gamma \nu$, then under Definition 6.1 we have $(O,\mu) \sim_\Gamma (O,\nu)$. □

As an illustration of the timing differences allowed by weak probabilistic low bisimulation, consider again the example program of Fig. 1, which is well typed if x is H and y is L. 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 $1/2$.

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

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

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

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

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

Fig. 9. Global configurations and their probabilities after three computation steps.

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 Fig. 9. 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 $1/2$.

7. Dynamic thread creation

Our language can be safely extended with a **fork** command for generating new threads dynamically. Here we informally sketch an argument justifying this claim.

Let us introduce a new command, $\textbf{fork}(c_1, \ldots, c_n)$, which terminates in one step, generating fresh names $\alpha_1, \ldots, \alpha_n$ and adding new threads $\alpha_1 = c_1, \ldots, \alpha_n = c_n$ to the thread pool.

We claim that we can safely type **fork** with the following typing rule:

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

What is significant here is that if each c_i has type $H\ cmd\ \_$, then $\textbf{fork}(c_1, \ldots, c_n)$ can be given type $H\ cmd\ 1$, which means that it can appear in the body of an **if** or **while** with a H guard. In this case, changing the initial values of H variables can affect what threads are forked. But, because such threads have type $H\ cmd\ \_$, they are *unessential*.

To argue that well-typed thread pools still satisfy weak probabilistic noninterference, we need to modify Definition 6.1 to allow $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ even when O_1

and O_2 use different thread names. To this end, we say that $(O_1, \mu) \sim_\Gamma (O_2, \nu)$ if there is a one-to-one renaming function r from thread names to thread names such that $(O_1 \circ r, \mu) \sim_\Gamma (O_2, \nu)$ under Definition 6.1.

Now, the key condition that allows us to prove that $\sim_\Gamma$ is a weak probabilistic low bisimulation is equation (14), which says that, provided that it is possible to leave equivalence class A, the probability of leaving A eventually is 1. With **fork** in the language, we no longer have the property used in the proof of Theorem 6.2 that if global configuration $a' \in A$ is reachable from a, then $u_{a'} \leqslant u_a$. 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 $u_{a_1} \leqslant u_{a_0} + n$, $u_{a_2} \leqslant u_{a_1} + n$, and so forth. If we let ϵ_i denote the probability of leaving class A at step i, $i \geqslant 0$, we see that

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

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

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

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.

8. Related work

As can be seen from the excellent survey by Sabelfeld and Myers [18], the secure information flow problem has been heavily studied on a range of languages from simple imperative languages [24] to object-oriented languages [6] to the π-calculus [13]. We briefly discuss those works most closely related to this.

Myers's Jif language [16] considers a large and complex subset of Java, including exceptions but not threads. Interestingly, his rule for sequential composition is somewhat similar to ours, in that his rule prevents an assignment to a L variable from sequentially following a command that might raise an exception depending on the value of a H variable. However, Jif's type system has not been proved to guarantee a noninterference property.

As mentioned in Section 4, Boudol and Castellani [7] independently developed a type system almost identical to ours. Their soundness proof, however, stays within a possibilistic setting, avoiding consideration of probabilities.

Honda and Yoshida [13] develop a linear/affine typed pi-calculus and show how to embed several languages for secure information flow into their pi-calculus. In particular they show an embedding of the multi-threaded language considered here, and they extend it with general references and higher-order procedures. However, because their pi-calculus has a purely nondeterministic semantics, they address only possibilistic noninterference properties.

Zdancewic and Myers [26] have proposed a very different approach to achieving secure information flow in a multi-threading language. Their approach is based on forbidding observable nondeterminism like that exhibited by the example program in Fig. 1. It would be interesting to compare the restrictiveness in practice of their system with ours; in general the practicality of secure information flow analysis has yet to be convincingly demonstrated.

9. Conclusion

The type system proposed here improves on that of [23] by admitting a much larger class of programs than previously permitted. In particular, there seems to be hope that the restrictions might not be too hard to accommodate in practice, since any threads that involve only H variables or only L variables are automatically well typed.

On the other hand, the soundness of our type system relies crucially on the fine-grained, probabilistic scheduler specified in Fig. 3. As mentioned at the end of Section 2, this scheduler is of course very far from typical practical implementations like round-robin time slicing, suggesting that our results are mainly of theoretical interest. While this is a valid concern, it is also important to recognize that something like round-robin time slicing gives rise to very subtle timing leaks that would be very hard to prevent with typing rules. Consider the following example, adapted from Agat [2].

```
int i, count, xs[4096], ys[4096];

for (count = 0; count < 1000; count++) {
  if (secret != 0)
    for (i = 0; i < 4096; i += 2)
      xs[i]++;
  else
    for (i = 0; i < 4096; i += 2)
      ys[i]++;
  for (i = 0; i < 4096; i += 2)
    xs[i]++;
}
leak = 0;
```

Suppose that `secret` is either 0 or 1. At an abstract level, the amount of work done by this program does not seem to depend on the value of `secret`. But, when run on a local Sparc server with a 16K data cache, it takes twice as long when `secret` is 0 as it takes when `secret` is 1. (When `secret` is 1, the array `xs` can remain in the data cache throughout the program's execution; when `secret` is 0, the data cache holds `xs` and `ys` alternately.) As a result, if this program were run concurrently with another thread

```
leak = 1;
```

then, under round-robin time slicing, we are likely to end up copying `secret` into `leak`. It is arguable, then, that if we really want to prevent information leaks in multi-threaded programs, the best approach may be to adopt the probabilistic scheduling used here, in spite of its inefficiency.

In future work on our type system, it would be interesting to explore algorithms for type checking or type inference, and to extend the language with additional features, such as synchronization primitives.

Acknowledgements

This paper integrates the contents of [20,21]. I thank Zhenyue Deng, Andrei Sabelfeld, Dennis Volpano, and the anonymous referees for helpful comments, and Ryan Yocum for developing the implementation [25] used to compute the example probabilities in this paper. This work was partially supported by the National Science Foundation under grants CCR-9900951 and HRD-0317692.

References

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

[2] J. Agat, Type based techniques for covert channel elimination and register allocation, PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2000.

[3] A. Aldini, Probabilistic information flow in a process algebra, in: *Proc. CONCUR 2001 – Concurrency Theory*, Lecture Notes in Computer Science 2154, 2001, pp. 152–168.

[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*, Lecture Notes in Computer Science 1254, 1997, pp. 119–130.

[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*, Cape Breton, Nova Scotia, Canada, 2002, pp. 253–267.

[7] G. Boudol and I. Castellani, Non-interference for concurrent programs and thread systems, *Theoretical Computer Science* **281**(1) (2002), 109–130.

[8] D. Denning and P. Denning, Certification of programs for secure information flow, *Communications of the ACM* **20**(7) (1977), 504–513.

[9] W. Feller, *An Introduction to Probability Theory and Its Applications*, Volume I, third edition, Wiley, 1968.

[10] C.A. Gunter, *Semantics of Programming Languages*, The MIT Press, 1992.

[11] H. Hermanns, Interactive Markov cains, PhD thesis, University of Erlangen-Nürnberg, 1998.

[12] 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*, 2000, pp. 180–199.

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

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

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

[16] A. Myers, JFlow: Practical mostly-static information flow control, in: *Proceedings 26th Symposium on Principles of Programming Languages*, San Antonio, TX, 1999, pp. 228–241.

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

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

[19] A. Sabelfeld and D. Sands, Probabilistic noninterference for multi-threaded programs, in: *Proceedings 13th IEEE Computer Security Foundations Workshop*, Cambridge, UK, 2000, pp. 200–214.

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

[21] G. Smith, Probabilistic noninterference through weak probabilistic bisimulation, in: *Proceedings 16th IEEE Computer Security Foundations Workshop*, Pacific Grove, California, 2003, pp. 3–13.

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

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

[24] D. Volpano, G. Smith and C. Irvine, A sound type system for secure flow analysis, *Journal of Computer Security* **4**(2,3) (1996), 167–187.

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

[26] S. Zdancewic and A.C. Myers, Observational determinism for concurrent program security, in: *Proceedings 16th IEEE Computer Security Foundations Workshop*, Pacific Grove, California, 2003, pp. 29–43.