Nontermination and Secure Information Flow *

Geoffrey Smith and Rafael Alpízar
School of Computing and Information Sciences
Florida International University
Miami, FL 33199 USA

October 24, 2011

Abstract

In secure information flow analysis, the classic Denning restrictions allow a program's termination to be affected by the values of its H variables, resulting in potential information leaks. In an effort to quantify such leaks, in this work we study a simple imperative language with random assignments. As a thought experiment, we propose a "stripping" operation on programs, which eliminates all "high computation", and we prove a fundamental property: stripping cannot decrease the probability of any low outcome. To prove this property, we first introduce a new notion of *fast probabilistic simulation* on Markov chains and we show that it implies a key reachability property. Viewing the stripping function as a binary relation, we then prove that stripping is a fast simulation. As an application we prove that, under the Denning restrictions, well-typed probabilistic programs are guaranteed to satisfy an approximate probabilistic noninterference property, provided that their probability of nontermination is small.

1 Introduction

Secure information flow analysis aims to prevent untrusted programs from leaking the sensitive information that they manipulate. If we classify variables as H (high) or L (low), then our goal is to prevent information in H variables from flowing into L variables. (More generally, we may want a richer *lattice* of security levels.) The seminal work in this area was the Dennings' 1977 paper [DD77], which proposed what we call the *Denning restrictions*:

- An expression is classified as H if it contains any H variables; otherwise, it is classified as L.
- To prevent *explicit flows*, a H expression cannot be assigned to a L variable.
- To prevent *implicit flows*, an **if** or **while** command whose guard is H may not make *any* assignments to L variables.

*To appear in a special issue of Mathematical Structures in Computer Science.

```
t ←? {0, 1};
if t = 0 then (
    while h = 1 do done;
    l := 0
)
else (
    while h = 0 do done;
    l := 1
)
```

Figure 1: A random assignment program

Much later, Volpano, Smith, and Irvine [VSI96] showed the soundness of the Denning restrictions by formulating them as a type system and proving that they ensure a *noninterference* property. Much work has followed; see [SM03] for a survey up to 2003.

Noninterference says, roughly, that the final values of L variables are independent of the initial values of H variables. It is formalized using the concept of *low equivalence* of memories:

Definition 1.1 *Memories μ and ν are* low equivalent, *written $\mu \sim_L \nu$, if μ and ν agree on the values of all L variables.*

The idea is that if initial memories μ and ν are low equivalent, then running c under μ should be (in some sense) equivalent to running c under ν, as far as L variables are concerned. But the possibility of nontermination causes complications, because the Denning restrictions allow programs whose termination behavior depends on the values of H variables. For example, the program **while** $h = 1$ **do done** (where h is a H variable) might terminate under μ and loop under ν.

One reaction to the termination issue is to say that further restrictions are needed. A number of studies have proposed forbidding H variables in the guards of **while** loops (e.g. [VS97]) or forbidding assignments to L variables that sequentially follow commands whose termination depends on H variables (e.g. [Smi06]). But such additional restrictions may in practice be overly stringent, making it difficult to write useful programs. For this reason, practical secure information-flow languages like Jif [MCN+06] have chosen *not* to impose extra restrictions to control termination leaks.

In this paper, therefore, we study the behavior of potentially nonterminating programs typed just under the Denning restrictions. To be able to make *quantitative* statements about the effects of nontermination, we consider a *probabilistic* language. In such a language, we would like to achieve *probabilistic noninterference* [VS99], which asserts that the probability distribution on the final values of L variables is independent of the initial values of H variables.

For example, consider the program in Figure 1. (In the code, $t \xleftarrow{?} \{0, 1\}$ is a *random assignment* that assigns either 0 or 1 to t, each with probability $1/2$, and **done** is like **skip**.) Assuming that h is H and t and l are L, this program satisfies the Denning

$t \stackrel{?}{\leftarrow} \{0, 1\}$;
if $t = 0$ **then**
    $l := 0$
else
    $l := 1$

Figure 2: Stripped version of the program

restrictions and it is well typed under the typing rules that we will present in Section 2. But, if $h = 0$, then this program terminates with $l = 0$ with probability $1/2$ and fails to terminate with probability $1/2$. And if $h = 1$, then it terminates with $l = 1$ with probability $1/2$ and fails to terminate with probability $1/2$. Thus this program does not satisfy probabilistic noninterference.

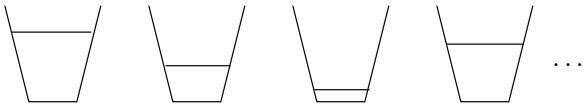
What goes wrong? Intuitively, the program is "trying" to set l to either 0 or 1, each with probability $1/2$, regardless of the value of h. But the **while** loops, whose termination depends on the value of h, sometimes prevent assignments to l from being reached. As a result, the probabilities of certain final values of l are lowered, because the paths that would have led to them become infinite loops. This suggests, perhaps, that if a well-typed program's probability of nontermination is small, then it will "almost" satisfy probabilistic noninterference.

To make these intuitions precise, we consider a thought experiment. For any well-typed program c, we define a "stripped" version of c, which we denote by $\lfloor c \rfloor$. In $\lfloor c \rfloor$, all purely "high computation" is removed; more precisely, we eliminate any subcommands that make no assignments to L variables. For example, the stripped version of the program in Figure 1 is shown in Figure 2.

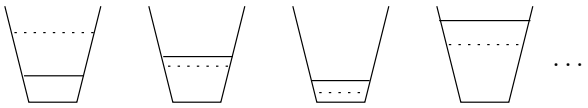
We emphasize that stripping is for us a *thought experiment*—it is not something that we would actually use in an implementation, but rather it is a means to *understand* a program's behavior.

The major technical effort of the paper is to prove a precise relationship between the behavior of a well-typed program c and of its stripped version $\lfloor c \rfloor$. We will show that the only effect of the stripping operation is to boost the probabilities of certain L outcomes by lowering the probability of nontermination. For example, consider the program in Figure 1 when $h = 0$. Stripping boosts the probability of terminating with $l = 1$ from 0 up to $1/2$ by lowering the probability of nontermination from $1/2$ down to 0; it leaves the probability of terminating with $l = 0$ unchanged at $1/2$.

More precisely, we will prove in Theorem 4.3 that the probability that c terminates with certain values for its L variables is always less than or equal to the corresponding probability for $\lfloor c \rfloor$. To visualize this theorem, imagine that the result of running c is shown as a sequence of buckets, one for each possible final value of c's L variables. Also, we have a bucket to represent nontermination. The probability of each outcome is indicated by the amount of water in each bucket; there is a total of one gallon of water among all the buckets. Suppose that c's buckets look like this:



Then Theorem 4.3 tells us that $\lfloor c \rfloor$'s buckets are gotten simply by pouring some of the water from c's nontermination bucket into some of the other buckets:



Our Theorem 4.3 was claimed, without proof, as Theorem 3.6 of our earlier paper [SA06]. Furthermore, it was claimed there that the proof could be done using *strong probabilistic simulation* as defined by Jonsson and Larsen [JL91]. But that claim is not correct; strong simulation turns out to be too restrictive for this purpose. For this reason, we introduce a new probabilistic simulation, which we call *fast simulation*, and show that it implies a key reachability property. Viewing the stripping function as a binary relation, we then prove in Theorem 4.2 the crucial fact that *stripping is a fast simulation*, which implies Theorem 4.3.

Theorem 4.3 gives us a way of bounding the effect of nontermination. For example, if c's nontermination bucket is empty, then $\lfloor c \rfloor$'s buckets are identical to c's, because there is no water to pour. More generally, we prove in Theorem 5.2 that if a well-typed program c fails to terminate with probability at most p, then c's deviation from probabilistic noninterference is at most $2p$.

We see this paper as making two main contributions. First, it gives a quantitative account of information flows caused by nontermination in programs that satisfy just the Denning restrictions; this is important for understanding more precisely what is guaranteed in languages that allow termination channels. Second, this paper makes a technical contribution by introducing a new notion of simulation, called a *fast simulation*, and applying it to the area of secure information flow; to our knowledge probabilistic simulation (unlike probabilistic *bisimulation*) has not previously been used in secure information flow.

The rest of this paper is organized as follows. In Section 2, we review the simple imperative language with random assignment that we will study. In Section 3, we explore the theory of probabilistic simulation in the abstract setting of Markov chains, developing a variant that we call *fast simulation*. In Section 4, we define the stripping operation $\lfloor \cdot \rfloor$ formally and use the theory of fast simulation to prove Theorem 4.3, which gives a fundamental relationship between the behavior of a well-typed command and its stripped version. In Section 5, we develop several applications of these results. Finally, Section 6 presents related work and concludes.

$$
\begin{array}{llll}
(\textit{phrases}) & p & ::= & e \mid c \\
(\textit{expressions}) & e & ::= & x \mid n \mid e_1 + e_2 \mid \ldots \\
(\textit{commands}) & c & ::= & \textbf{done} \mid x := e \mid x \stackrel{?}{\leftarrow} \mathcal{D} \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } e \textbf{ do } c \mid c_1; c_2
\end{array}
$$

Figure 3: Language Syntax

2 A Random Assignment Language

In this section, we review the syntax, semantics, and type system for our simple imperative language with random assignment. The language syntax is defined in Figure 3. In the syntax, metavariables x, y, z range over identifiers and n over integer literals. Integers are the only values; we use 0 for false and nonzero for true.

A novelty of our language is that we have replaced the usual **skip** command with a **done** command instead; **done** can be used in much the same way as **skip** and (as will be seen below) it is also used to represent a terminated command in a configuration.

The command $x \stackrel{?}{\leftarrow} \mathcal{D}$ is a random assignment; here $\mathcal{D}$ ranges over some set of probability distributions on the integers. In examples, we use notation like $x \stackrel{?}{\leftarrow} \{0, 1, 2\}$ to denote a random assignment command that assigns either 0, 1, or 2 to x, each with equal probability. We remark that our language allows random *assignments* but not random *expressions*; this lets us use a simpler semantics.

A program c is executed under a *memory* μ, which maps identifiers to integers. We assume that expressions are total and evaluated atomically, and we write $\mu(e)$ to denote the value of expression e in memory μ. In our semantics, a *configuration* is a pair (c, μ) where c is a command and μ is a memory. Note that *terminal* configurations are written as $(\textbf{done}, \mu)$ in our semantics. We remark that the more common semantic approach is to have both non-terminal configurations (c, μ) and also terminal configurations μ, but this tends to lead to a proliferation of cases in proofs; it is therefore more pleasant to have only configurations of the form (c, μ).

Because of the random assignment command, the standard transition relation on configurations needs to be extended with probabilities—we write $(c, \mu) \xrightarrow{p} (c', \mu')$ to indicate that the probability of going from configuration (c, μ) to configuration (c', μ') is p. The semantic rules are given in Figure 4. They define a *Markov chain* [Fel68] on the set of configurations. Notice that terminal configurations $(\textbf{done}, \mu)$ are *absorbing states* in the Markov chain.

Next we describe our type system, which simply enforces the Denning restrictions. Here are the types it uses:

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

As usual, τ *var* is the type of variables that store information of level τ, while τ *cmd* is the type of commands that assign only to variables of level τ or higher; this implies that command types obey a *contravariant* subtyping rule. Typing judgments have the form $\Gamma \vdash p : \rho$, where Γ is an *identifier typing* that maps each variable to a type of the form

($done_s$) $(\textbf{done}, \mu) \xrightarrow{1} (\textbf{done}, \mu)$

($update_s$) $$\frac{x \in dom(\mu)}{(x := e, \mu) \xrightarrow{1} (\textbf{done}, \mu[x := \mu(e)])}$$

($random_s$) $$\frac{x \in dom(\mu) \;\; \mathcal{D}(v) > 0}{(x \overset{?}{\leftarrow} \mathcal{D}, \mu) \xrightarrow{\mathcal{D}(v)} (\textbf{done}, \mu[x := v])}$$

(if_s) $$\frac{\mu(e) \neq 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{1} (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{1} (c_2, \mu)}$$

($while_s$) $$\frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{1} (\textbf{done}, \mu)}$$

$$\frac{\mu_(e) \neq 0}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{1} (c; \textbf{while } e \textbf{ do } c, \mu)}$$

($compose_s$) $$\frac{(c_1, \mu) \xrightarrow{p} (\textbf{done}, \mu')}{(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')}$$

$$\frac{(c_1, \mu) \xrightarrow{p} (c_1', \mu') \;\; c_1' \neq \textbf{done}}{(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')}$$

Figure 4: Structural Operational Semantics

τ *var*. (We generally assume a single fixed Γ throughout, so in our discussions we will seldom mention Γ.) The typing and subtyping rules are given in Figure 5. The rules are the same as those in [VSI96], except for the new rules *done*$_t$ which types **done** as a high command and *random*$_t$ for random assignment. Rule *random*$_t$ says that a random assignment can be done to any variable x, but if x has type τ *var*, then the assignment gets type τ *cmd*; this type is used prevent improper implicit flows.

Under this type system, we have the usual Simple Security, Confinement, and Subject Reduction properties:

Lemma 2.1 (Simple Security) *If $\Gamma \vdash e : \tau$, then e contains only variables of level τ or lower.*

Proof. By induction on the structure of e. □

Lemma 2.2 (Confinement) *If $\Gamma \vdash c : \tau$ cmd, then c assigns only to variables of level τ or higher.*

Proof. By induction on the structure of c. □

Lemma 2.3 (Subject Reduction) *If $\Gamma \vdash c : \tau$ cmd and $(c, \mu) \xrightarrow{p} (c', \mu')$ for some $p > 0$, then $\Gamma \vdash c' : \tau$ cmd.*

Proof. By induction on the structure of c. □

But, as discussed in the Introduction, well-typed programs need not satisfy probabilistic noninterference because changes to H variables can result in infinite loops that block subsequent assignments to L variables, affecting the probabilities of different L outcomes. Before undertaking a deeper study of the behavior of well-typed programs, we first must develop some useful results in the theory of probabilistic simulation.

3 Probabilistic Simulation

In this section, we discuss the theory of probabilistic bisimulation and simulation in an abstract setting.

A (discrete-time) *Markov chain* [Fel68] is a pair $(S, \mathbf{P})$ where

- S is a countable set of states, and
- $\mathbf{P} : S \times S \to [0, 1]$ is a probability matrix satisfying $\sum_{t \in S} \mathbf{P}(s, t) = 1$ for all $s \in S$.

If $\mathbf{P}(s, t) > 0$, then we say that t is a *successor* of s. Also, for $T \subseteq S$, we write $\mathbf{P}(s, T)$ to denote $\sum_{t \in T} \mathbf{P}(s, t)$, the probability of going in one step from s to a state in T.

A classic equivalence relation on Markov chains is *probabilistic bisimulation*, due to Kemeny and Snell [KS60] and Larsen and Skou [LS91].

Definition 3.1 *Let R be an equivalence relation on S. R is a* strong bisimulation *if whenever sRt we have $\mathbf{P}(s, T) = \mathbf{P}(t, T)$ for every equivalence class T of R.*

($rval_t$) $\dfrac{\Gamma(x) = \tau \ \mathit{var}}{\Gamma \vdash x : \tau}$

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

($plus_t$) $\dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$

($done_t$) $\Gamma \vdash \textbf{done} : H \ \mathit{cmd}$

($update_t$) $\dfrac{\Gamma(x) = \tau \ \mathit{var} \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau \ \mathit{cmd}}$

($random_t$) $\dfrac{\Gamma(x) = \tau \ \mathit{var}}{\Gamma \vdash x \overset{?}{\leftarrow} \mathcal{D} : \tau \ \mathit{cmd}}$

(if_t) $\dfrac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau \ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}}{\Gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \tau \ \mathit{cmd}}$

($while_t$) $\dfrac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau \ \mathit{cmd}}{\Gamma \vdash \textbf{while } e \textbf{ do } c : \tau \ \mathit{cmd}}$

($compose_t$) $\dfrac{\Gamma \vdash c_1 : \tau \ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}}{\Gamma \vdash c_1; c_2 : \tau \ \mathit{cmd}}$

(*base*) $L \subseteq H$

(*cmd*) $\dfrac{\tau \subseteq \tau'}{\tau' \ \mathit{cmd} \subseteq \tau \ \mathit{cmd}}$

(*reflex*) $\rho \subseteq \rho$

(*trans*) $\dfrac{\rho_1 \subseteq \rho_2 \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$

(*subsump*) $\dfrac{\Gamma \vdash p : \rho_1 \quad \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$

Figure 5: Typing and subtyping rules

Both strong and weak versions of bisimulation have been applied fruitfully in secure information flow analysis, for example by Gray [Gra90], Sabelfeld and Sands [SS00], and Smith [Smi06]. The basic idea is that a secure program should behave (in some sense) "indistinguishably" when run under two low-equivalent initial memories; this indistinguishability can be formalized as a bisimulation.

In this paper, we apply instead non-symmetric *probabilistic simulation* relations, explored earlier by Jonsson and Larsen [JL91] and Baier, Katoen, Hermanns, and Wolf [BKHW05]. Roughly speaking, binary relation R is a strong simulation if whenever sRt and s has a successor s', t has a "matching" successor t' that simulates s' (i.e. $s'Rt'$). But in the probabilistic setting, we must also make sure that the probabilities are preserved. Suppose for example that $\mathbf{P}(s, s') = p$. Then t must be able to match that much probability. But t need not have a single successor t' such that $\mathbf{P}(t, t') = p$. Instead, it is enough for t to have several successors, each simulating s', such that the total probability is p. However, in doing this simulation we must not "double count" t's probabilities—for example, if s goes to s' with probability 1/3 and t goes to t' with probability 1/2, then if we use t' to match the move to s' we must remember that 1/3 of t''s probability is "used up", leaving just 1/6 to be used in matching other moves of s. These considerations lead to what is called a *weight function* Δ to specify how the probabilities are matched up, giving the following definition (adapted from Definition 16 of [BKHW05]):

Definition 3.2 *Let R be a binary relation on S. R is a* strong simulation *if, whenever sRt, there exists a function $\Delta : S \times S \to [0, 1]$ such that*

1. $\Delta(s', t') > 0$ *implies that* $s'Rt'$,
2. $\mathbf{P}(s, s') = \sum_{u \in S} \Delta(s', u)$ *for all* $s' \in S$,
3. $\mathbf{P}(t, t') = \sum_{u \in S} \Delta(u, t')$ *for all* $t' \in S$.

For our exploration of the stripping operation $\lfloor \cdot \rfloor$ in Section 4, however, it turns out that strong simulation isn't quite what we want, because it does not allow the simulating state t to run "faster" than the simulated state s. The issue is that s could make "insignificant" moves to states that are already simulated by t; in this case t shouldn't need to make a matching move. Such "insignificant" moves are allowed by the more flexible notion of *weak simulation* in Definition 34 of [BKHW05]. But their definition also allows t to make "insignificant" moves, which are not appropriate for us, since we want t to run at least as fast as s. So here we develop a restricted kind of weak simulation, which we call a *fast simulation*.

Suppose that sRt. We partition the successors of s into two sets, U and V. The states in V represent "insignificant" moves, and we require that t itself simulates each of them. The states in U represent "significant" moves, and we require that t be able to match such moves with some weight function. Intuitively, then, t matches s's moves either by doing nothing or by moving. Formally, we have the following definition:

Definition 3.3 *Let R be a binary relation on S. R is a* fast simulation *if, whenever sRt, the successors of s can be partitioned into two sets U and V such that*

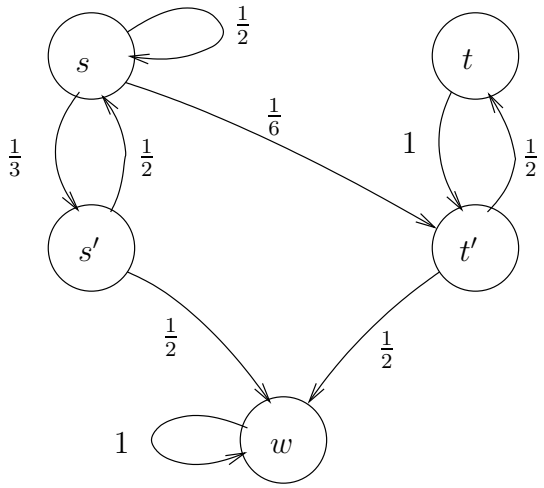


Figure 6: An example Markov chain

1. vRt *for every* $v \in V$*, and*
2. *letting* $K = \sum_{u \in U} \mathbf{P}(s,u)$*, if* $K > 0$ *then there exists a function* $\Delta : S \times S \to [0,1]$ *such that*
   (a) $\Delta(u,w) > 0$ *implies that* uRw*,*
   (b) $\mathbf{P}(s,u)/K = \sum_{w \in S} \Delta(u,w)$ *for all* $u \in U$*, and*
   (c) $\mathbf{P}(t,w) = \sum_{u \in U} \Delta(u,w)$ *for all* $w \in S$*.*

Notice that in condition 2(b), $\mathbf{P}(s,u)/K$ is the *conditional probability* of going from s to u, given that s goes to U. (The reason for using a conditional probability here may be intuitively unclear; in fact the best justification for this definition is its utility in the proof of the key Theorem 3.2 below.)

Example 3.1 *Consider the Markov chain in Figure 6, where* $S = \{s,t,s',t',w\}$*. Define* R *by* sRt*,* $s'Rt'$*, together with* uRu *for every* $u \in S$*. (In other words,* R *is the* reflexive closure *of* $\{(s,t),(s',t')\}$*.) Then we can show that* R *is a fast simulation:*

- *For pairs of the form* xRx*, we can always satisfy the requirements of Definition 3.3 by choosing* U *to be the set of successors of* x *and* V *to be* $\emptyset$*. Then* $K = 1$*, and for each* $u \in U$ *we can choose* $\Delta(u,u) = \mathbf{P}(x,u)$*. It is straightforward to verify that these choices satisfy conditions 1, 2(a), 2(b), and 2(c).*
- *For* sRt *we can choose* $U = \{s',t'\}$ *and* $V = \{s\}$*, which makes* $K = \frac{1}{2}$*, and we can choose* $\Delta(s',t') = \frac{2}{3}$ *and* $\Delta(t',t') = \frac{1}{3}$*.*
- *Finally, for* $s'Rt'$ *we can choose* $U = \{s,w\}$ *and* $V = \emptyset$*, which makes* $K = 1$*, and we can choose* $\Delta(s,t) = \frac{1}{2}$ *and* $\Delta(w,w) = \frac{1}{2}$*.*

We remark that every strong simulation is also a fast simulation, since a strong simulation is just a fast simulation in which all the V sets are empty. Furthermore, every fast simulation is also a weak simulation as defined in Definition 34 of [BKHW05].

We now develop the key properties of fast simulation.

Definition 3.4 *Let R be a binary relation on S. A set T of states is* upwards closed with respect to R *if, whenever $s \in T$ and sRs', we also have $s' \in T$.*

If $s \in S$, n is a natural number, and $T \subseteq S$, then let us write $\Pr(s, n, T)$ to denote the probability of reaching a state in T from s in at most n steps. Following [BKHW05], we can calculate $\Pr(s, n, T)$ with a recurrence:

$$\Pr(s, n, T) \quad = \quad \begin{cases} 1, & \text{if } s \in T \\ \sum_{s' \in S} \mathbf{P}(s, s') \Pr(s', n-1, T), & \text{if } n > 0 \text{ and } s \notin T \\ 0, & \text{if } n = 0 \text{ and } s \notin T \end{cases}$$

Note that $\Pr(s, n, T)$ increases monotonically with n:

Lemma 3.1 $\Pr(s, n, T) \leq \Pr(s, n+1, T)$*, for all n, s, and T.*

Proof. By induction on n. For the basis, if $s \notin T$, then $\Pr(s, 0, T) = 0 \leq \Pr(s, 1, T)$. And if $s \in T$, then $\Pr(s, 0, T) = 1 = \Pr(s, 1, T)$.

For the induction, assume that for some $k \geq 0$ we have $\Pr(s, k, T) \leq \Pr(s, k+1, T)$ for all s and T. We must show that $\Pr(s, k+1, T) \leq \Pr(s, k+2, T)$. First note that if $s \in T$ then we have $\Pr(s, k+1, T) = 1 = \Pr(s, k+2, T)$. It remains to consider the case when $s \notin T$. In this case we have

$$\begin{aligned} \Pr(s, k+2, T) &= \sum_{s' \in S} \mathbf{P}(s, s') \Pr(s', k+1, T) \\ &\geq \sum_{s' \in S} \mathbf{P}(s, s') \Pr(s', k, T) \\ &\quad \text{(by induction)} \\ &= \Pr(s, k+1, T) \end{aligned}$$

□

We now proceed to the key theorem about fast simulation; its proof is similar to the proof of Theorem 54 of [BKHW05], though that theorem refers to *strong simulation*.

Theorem 3.2 *If R is a fast simulation, T is upwards closed with respect to R, and $s_1 R s_2$, then $\Pr(s_1, n, T) \leq \Pr(s_2, n, T)$ for every n.*

Proof. By induction on n. For the basis, note that if $s_1 \notin T$, then $\Pr(s_1, 0, T) = 0 \leq \Pr(s_2, 0, T)$. And if $s_1 \in T$, then $s_2 \in T$, since $s_1 R s_2$ and T is upwards closed. So we have $\Pr(s_1, 0, T) = 1 = \Pr(s_2, 0, T)$.

For the induction, first note that if $s_1 \in T$ then as above we have $\Pr(s_1, k+1, T) = 1 = \Pr(s_2, k+1, T)$, and if $s_2 \in T$ then we have $\Pr(s_1, k+1, T) \leq 1 = \Pr(s_2, k+1, T)$. It remains to consider the case when $s_1 \notin T$ and $s_2 \notin T$. In this case we have

$$\begin{aligned}\Pr(s_1, k+1, T) &= \sum_{s \in S} \mathbf{P}(s_1, s)\Pr(s, k, T) \\ &= \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) + \sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T)\end{aligned}$$

where U and V are as specified in Definition 3.3. (Note that the rearrangement is valid because the series are absolutely convergent.)

Now, for every $v \in V$, since vRs_2 we get by induction that $\Pr(v, k, T) \leq \Pr(s_2, k, T)$. Also, letting $K = \sum_{u \in U} \mathbf{P}(s_1, u)$, we note that $\sum_{v \in V} \mathbf{P}(s_1, v) = (1 - K)$. Hence we have

$$\begin{aligned}\sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) &\leq \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(s_2, k, T) \\ &= (\sum_{v \in V} \mathbf{P}(s_1, v))\Pr(s_2, k, T) \\ &= (1 - K)\Pr(s_2, k, T) \\ &\leq (1 - K)\Pr(s_2, k+1, T)\end{aligned}$$

Note that if $K = 0$, then $U = \emptyset$, which implies that

$$\begin{aligned}\Pr(s_1, k+1, T) &= \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) + \sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T) \\ &\leq (1 - 0)\Pr(s_2, k+1, T) + 0 \\ &= \Pr(s_2, k+1, T)\end{aligned}$$

We are left with the case when $K > 0$. In that case, by Definition 3.3 there exists a function Δ satisfying conditions 2(a), 2(b), and 2(c). Hence we have

$$\begin{aligned}\sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T) &= \sum_{u \in U} (K \sum_{w \in S} \Delta(u, w))\Pr(u, k, T) \\ &= K \sum_{u \in U} (\sum_{w \in S} \Delta(u, w)\Pr(u, k, T)) \\ &\leq K \sum_{u \in U} (\sum_{w \in S} \Delta(u, w)\Pr(w, k, T)) \\ &\quad \text{(by induction, as } \Delta(u, w) > 0 \text{ implies } uRw\text{)} \\ &= K \sum_{w \in S} (\sum_{u \in U} \Delta(u, w)\Pr(w, k, T)) \\ &= K \sum_{w \in S} (\sum_{u \in U} \Delta(u, w))\Pr(w, k, T) \\ &= K \sum_{w \in S} \mathbf{P}(s_2, w)\Pr(w, k, T) \\ &= K\Pr(s_2, k+1, T)\end{aligned}$$

Finally, we have

$$\begin{aligned}\Pr(s_1, k+1, T) &= \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) + \sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T) \\ &\leq (1-K)\Pr(s_2, k+1, T) + K\Pr(s_2, k+1, T) \\ &= \Pr(s_2, k+1, T)\end{aligned}$$

□

We can illustrate Theorem 3.2 by considering the Markov chain in Example 3.1 above and its fast simulation R. Because $s'Rt'$ and $\{w\}$ is upwards closed with respect to R, Theorem 3.2 tells us that $\Pr(s', n, \{w\}) \leq \Pr(t', n, \{w\})$, for every n. Direct calculation confirms these inequalities for $n \leq 5$:

n	$\Pr(s', n, \{w\})$	$\Pr(t', n, \{w\})$
0	0	0
1	$\frac{1}{2}$	$\frac{1}{2}$
2	$\frac{1}{2}$	$\frac{1}{2}$
3	$\frac{5}{8}$	$\frac{3}{4}$
4	$\frac{11}{16}$	$\frac{3}{4}$
5	$\frac{73}{96}$	$\frac{7}{8}$

We remark that the *universal relation* $R_U = S \times S$ is trivially a fast simulation. But under R_U the only upwards closed sets are $\emptyset$ and S itself, which means that Theorem 3.2 is uninteresting in that case.

Note that Theorem 3.2 also holds if R is a strong simulation, since every strong simulation is also a fast simulation. Interestingly, Theorem 3.2 *fails* if R is a weak simulation as defined in Definition 34 of [BKHW05]. Here is a counterexample:

Example 3.2 *Consider the Markov chain in Figure 7, from page 197 of [BKHW05]. If R is the reflexive closure of $\{(s, s'), (u_1, u_2)\}$, then R is a weak simulation and $\{w\}$ is upwards closed with respect to R, yet* $\Pr(s, 3, \{w\}) = 7/16$ *and* $\Pr(s', 3, \{w\}) = 6/16$. *Notice that in this case R is not a fast simulation.*

Now that we have developed fast simulation on abstract Markov chains, we are almost ready to apply it to our study of the stripping operation $\lfloor \cdot \rfloor$. It turns out, however, that when we study the behavior of $\lfloor \cdot \rfloor$ we will see that we do not need the great flexibility allowed by fast simulation as defined in Definition 3.3. We therefore introduce a restricted kind of fast simulation as follows:

Definition 3.5 *A binary relation R on S is a* simple fast simulation *if, whenever sRt, either*

1. *for every successor s' of s, we have $s'Rt$; or else*
2. *there is a bijection δ from the successors of s to the successors of t such that for every successor s' of s, we have $\mathbf{P}(s, s') = \mathbf{P}(t, \delta(s'))$ and $s'R\delta(s')$.*

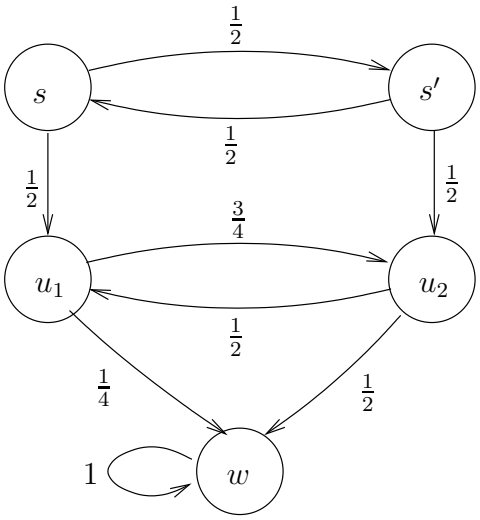


Figure 7: Another example Markov chain

Compared with fast simulation in Definition 3.3, we can see that case 1 here corresponds to the situation where all of the moves from s are "inessential", allowing us to take $U = \emptyset$. And case 2 corresponds to the situation where all of the moves from s are "essential", allowing us to take $V = \emptyset$, and moreover we can use a very simple weight function that pairs up the successors of s and the successors of t in a one-to-one manner.

Next we show that every simple fast simulation is indeed a fast simulation.

Theorem 3.3 *Every simple fast simulation R is a fast simulation.*

Proof. Suppose that R is a simple fast simulation and that sRt.

If case 1 holds, then we satisfy Definition 3.3 by letting V be the set of successors of s and letting U be $\emptyset$. Note in this case that $K = 0$, so condition 2 of Definition 3.3 is satisfied vacuously.

If case 2 holds, then we let U be the set of successors of s and let V be $\emptyset$. Now condition 1 of Definition 3.3 is satisfied vacuously. To satisfy condition 2, note that $K = 1$ and define $\Delta(s', \delta(s')) = \mathbf{P}(s, s') = \mathbf{P}(t, \delta(s'))$. (All other values of Δ are 0.) It is straightforward to verify that Δ satisfies conditions 2(a), 2(b), and 2(c). □

4 Stripping and its Properties

In this section, we formally define the stripping operation on well-typed commands in our random assignment language and use our results about fast simulation to prove a fundamental result about the relationship between the behavior of c and of $\lfloor c \rfloor$. Intuitively, $\lfloor c \rfloor$ eliminates all subcommands of c that contain no assignments to L vari-

ables; it is easy to see that this is the same as eliminating subcommands of type H *cmd*. More precisely, we have the following definition, an early version of which appeared in [SA06]:

Definition 4.1 *Let c be a well-typed command. We define $\lfloor c \rfloor =$* **done** *if c has type H cmd; otherwise, define $\lfloor c \rfloor$ by*

- $\lfloor x := e \rfloor = x := e$
- $\lfloor x \stackrel{?}{\leftarrow} \mathcal{D} \rfloor = x \stackrel{?}{\leftarrow} \mathcal{D}$
- $\lfloor \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rfloor = \textbf{if } e \textbf{ then } \lfloor c_1 \rfloor \textbf{ else } \lfloor c_2 \rfloor$
- $\lfloor \textbf{while } e \textbf{ do } c \rfloor = \textbf{while } e \textbf{ do } \lfloor c \rfloor$
- $\lfloor c_1; c_2 \rfloor = \begin{cases} \lfloor c_2 \rfloor & \textit{if } c_1 : H \textit{ cmd} \\ \lfloor c_1 \rfloor & \textit{if } c_2 : H \textit{ cmd} \\ \lfloor c_1 \rfloor; \lfloor c_2 \rfloor & \textit{otherwise} \end{cases}$

Also, we define $\lfloor \mu \rfloor$ to be the result of deleting all H variables from μ and we extend $\lfloor \cdot \rfloor$ to well-typed configurations by $\lfloor (c, \mu) \rfloor = (\lfloor c \rfloor, \lfloor \mu \rfloor)$.

We remark that stripping as defined in [SA07] replaced subcommands of type H *cmd* with **skip**; in contrast our new definition here aggressively *eliminates* such subcommands in sequential compositions and replaces them with **done** in the other cases. Note that $\lfloor \mu \rfloor \sim_L \mu$. Now we have a simple lemma:

Lemma 4.1 *For any command c, $\lfloor c \rfloor$ contains only L variables.*

Proof. By induction on the structure of c. If c has type H *cmd*, then $\lfloor c \rfloor =$ **done**, which (vacuously) contains only L variables. If c does not have type H *cmd*, then we consider the form of c:

- If c is $x := e$, then $\lfloor c \rfloor = x := e$. Since c does not have type H *cmd*, then by rule *update*$_t$ we must have that x is a L variable and $e : L$, which implies by Simple Security that e contains only L variables.
- The case of $x \stackrel{?}{\leftarrow} \mathcal{D}$ is similar.
- If c is **if** e **then** c_1 **else** c_2, then $\lfloor c \rfloor =$ **if** e **then** $\lfloor c_1 \rfloor$ **else** $\lfloor c_2 \rfloor$. By rule *if*$_t$ we have $e : L$, which implies by Simple Security that e contains only L variables. And, by induction, $\lfloor c_1 \rfloor$ and $\lfloor c_2 \rfloor$ contain only L variables.
- If c is **while** e **do** c_1, then $\lfloor c \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$. By rule *while*$_t$, $e : L$, which implies by Simple Security that e contains only L variables. And, by induction, $\lfloor c_1 \rfloor$ contains only L variables.
- If c is $c_1; c_2$ then if $c_1 : H$ *cmd* then $\lfloor c \rfloor = \lfloor c_2 \rfloor$ which by induction contains only L variables. The other subcases are similar.

□

We now specialize fast simulation from arbitrary Markov chains to the particular Markov chain of well-typed configurations (c, μ) of our random assignment language. In addition, we impose the requirement that the simulating configuration's memory must be low equivalent to the simulated configuration's memory:

Definition 4.2 *A binary relation R on well-typed configurations is a* fast low simulation *if R is a fast simulation such that whenever $(c_1, \mu_1)R(c_2, \mu_2)$, we have $\mu_1 \sim_L \mu_2$.*

In set theory, recall that any function is also a relation. Hence our stripping function can also be viewed as a relation. When we view it that way, we use the notation $\lfloor \cdot \rfloor$ to denote the stripping relation, and we write $(c_1, \mu_1)\lfloor \cdot \rfloor(c_2, \mu_2)$ to denote $\lfloor (c_1, \mu_1) \rfloor = (c_2, \mu_2)$.

We now are ready for our main theorem, which is that the relation $\lfloor \cdot \rfloor$ is a fast low simulation. But first we we give an example that illustrates how the behaviors of (c, μ) and $\lfloor (c, \mu) \rfloor$ can differ. Consider the program

$$\begin{array}{l} \textbf{while } h \textbf{ do } h \overset{?}{\leftarrow} \{0, 1\}; \\ l := 1 \end{array}$$

and its stripped version

$$l := 1$$

Notice that under memory $\{h = 1, l = 0\}$ the original program can run for an arbitrary number of steps; it has infinitely many terminating traces, whose probabilities sum to 1. In contrast, the stripped program always terminates in exactly one step.

Theorem 4.2 *$\lfloor \cdot \rfloor$ is a fast low simulation.*

Proof. First note that if $(c, \mu)\lfloor \cdot \rfloor(d, \nu)$, then we have $\nu = \lfloor \mu \rfloor$, which implies $\mu \sim_L \nu$. More substantially, we must show that $\lfloor \cdot \rfloor$ is a fast simulation. By Theorem 3.3, it suffices to show the stronger result that $\lfloor \cdot \rfloor$ is a simple fast simulation. Note that the stronger result is *easier* to prove, because it gives us a stronger induction hypothesis to use in the case of $c_1; c_2$. (This is an example of what George Pólya called the "Inventor's Paradox".)

Now suppose that $(c, \mu)\lfloor \cdot \rfloor(d, \nu)$, which means that that $d = \lfloor c \rfloor$ and $\nu = \lfloor \mu \rfloor$. We must show that either condition 1 or condition 2 of Definition 3.5 holds. We make this argument by induction on the structure of c.

First, if c has type H *cmd*, then $d =$ **done**. Now consider the (possibly infinite) set of successors of (c, μ):

$$\begin{array}{lcl} (c, \mu) & \xrightarrow{p_1} & (c_1, \mu_1) \\ (c, \mu) & \xrightarrow{p_2} & (c_2, \mu_2) \\ (c, \mu) & \xrightarrow{p_3} & (c_3, \mu_3) \\ \ldots & & \end{array}$$

For every i, by Subject Reduction we have $c_i : H$ *cmd*, and by Confinement we have $\mu_i \sim_L \mu$. Hence we have $(c_i, \mu_i)\lfloor \cdot \rfloor(d, \nu)$. Thus we can see that condition 1 of Definition 3.5 is satisfied.

Next, if c does not have type H *cmd*, then consider the possible forms of c:

1. $c = x := e$.

   Here $d = c$. By *update*$_t$, $x : L$ *var* and $e : L$. So by Simple Security and the fact that $\nu = \lfloor \mu \rfloor$, we have $\mu(e) = \nu(e)$, which implies that $\nu[x := \nu(e)] = \lfloor \mu[x := \mu(e)] \rfloor$. Hence the move $(c, \mu) \xrightarrow{1} (\textbf{done}, \mu[x := \mu(e)])$ is matched by the move $(d, \nu) \xrightarrow{1} (\textbf{done}, \nu[x := \nu(e)])$. Formally, condition 2 of Definition 3.5 is satisfied by choosing $\delta((\textbf{done}, \mu[x := \mu(e)])) = (\textbf{done}, \nu[x := \nu(e)])$.

2. $c = x \overset{?}{\leftarrow} \mathcal{D}$.

   This case is similar to case 1; again $d = c$ and each move $(c, \mu) \xrightarrow{\mathcal{D}(v)} (\textbf{done}, \mu[x := v])$ is matched by $(d, \nu) \xrightarrow{\mathcal{D}(v)} (\textbf{done}, \nu[x := v])$. Formally, for each v such that $\mathcal{D}(v) > 0$, we choose $\delta((\textbf{done}, \mu[x := v])) = (\textbf{done}, \nu[x := v])$.

3. $c =$ **if** e **then** c_1 **else** c_2.

   Here $d =$ **if** e **then** $\lfloor c_1 \rfloor$ **else** $\lfloor c_2 \rfloor$. By *if*$_t$, $e : L$ and by Simple Security $\mu(e) = \nu(e)$. So if $\mu(e) \neq 0$, then $(c, \mu) \xrightarrow{1} (c_1, \mu)$ is matched by $(d, \nu) \xrightarrow{1} (\lfloor c_1 \rfloor, \nu)$. Formally, we choose $\delta((c_1, \mu)) = (\lfloor c_1 \rfloor, \nu)$. The case when $\mu(e) = 0$ is similar.

4. $c =$ **while** e **do** c_1.

   Here $d =$ **while** e **do** $\lfloor c_1 \rfloor$. By *while*$_t$, $e : L$ and c_1 does not have type H *cmd*. By Simple Security, we have $\mu(e) = \nu(e)$. So if $\mu(e) \neq 0$, then the move $(c, \mu) \xrightarrow{1} (c_1; \textbf{while } e \textbf{ do } c_1, \mu)$ is matched by $(d, \nu) \xrightarrow{1} (\lfloor c_1 \rfloor; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor, \nu)$. (Notice that because c_1 does not have type H *cmd*, we have $\lfloor c_1; \textbf{while } e \textbf{ do } c_1 \rfloor = \lfloor c_1 \rfloor; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor$.) Formally, we choose

   $$\delta((c_1; \textbf{while } e \textbf{ do } c_1, \mu)) = (\lfloor c_1 \rfloor; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor, \nu).$$

   The case when $\mu(e) = 0$ is similar.

5. $c = c_1; c_2$.

   First note that under rule *compose*$_s$, any move from $(c_1; c_2, \mu)$ results from a move

   $$(c_1, \mu) \xrightarrow{p} (c_1', \mu')$$

   where c_1' may or may not be **done**. If so, then the move is

   $$(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')$$

   and if not, then the move is

   $$(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu').$$

   Now we split into subcases, depending on the types of c_1 and c_2.

   (a) If c_1 has type H *cmd*, then $d = \lfloor c_2 \rfloor$. By Subject Reduction we have $c_1' : H$ *cmd*, and by Confinement we have $\mu' \sim_L \mu$. Hence (d, ν) can match either of the possible moves from $(c_1; c_2, \mu)$ by doing nothing, since

we have both $(c_2, \mu')\lfloor\cdot\rfloor(d, \nu)$ and $(c_1'; c_2, \mu')\lfloor\cdot\rfloor(d, \nu)$. So condition 1 of Definition 3.5 is satisfied.[1]

(b) If neither c_1 nor c_2 has type H *cmd*, then $d = \lfloor c_1\rfloor; \lfloor c_2\rfloor$. Define $d_1 = \lfloor c_1\rfloor$ and $d_2 = \lfloor c_2\rfloor$. Now, by induction (d_1, ν) can match the moves from (c_1, μ) so that either condition 1 or 2 of Definition 3.5 is satisfied. We consider these two possibilities in turn:

- If condition 1 is satisfied, then we always have $(c_1', \mu')\lfloor\cdot\rfloor(d_1, \nu)$. Now observe that d_1 cannot be **done**, since $d_1 = \lfloor c_1\rfloor$ and c_1 does not have type H *cmd*. This implies that c_1' cannot have type H *cmd* and in particular that c_1' is not **done**. Hence each move from c must be $(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')$. Moreover we have $(c_1'; c_2, \mu')\lfloor\cdot\rfloor(d_1; d_2, \nu)$. Hence condition 1 is satisfied for $(c_1; c_2, \mu)$ and $(d_1; d_2, \nu)$.
- If condition 2 is satisfied, then there is a bijection δ from the successors of (c_1, μ) to the successors of (d_1, ν). If we let (d_1', ν') denote $\delta((c_1', \mu'))$, then under condition 2 we have $(d_1, \nu) \xrightarrow{p} (d_1', \nu')$ and $(c_1', \mu')\lfloor\cdot\rfloor(d_1', \nu')$. Now consider c_1'.
  If $c_1' =$ **done**, then the move from $(c_1; c_2, \mu)$ is $(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')$. In this case we have $d_1' =$ **done** as well, so we have the matching move $(d_1; d_2, \nu) \xrightarrow{p} (d_2, \nu')$.
  And if $c_1' \neq$ **done**, then the move from $(c_1; c_2, \mu)$ is
  $$(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu').$$
  This move is matched by $(d_1; d_2, \nu) \xrightarrow{p} (d_1'; d_2, \nu')$, but only if c_1' does not have type H *cmd*. For if $c_1' : H$ *cmd*, then $\lfloor c_1'; c_2\rfloor = \lfloor c_2\rfloor = d_2 \neq d_1; d_2$. But in this case we have $d_1' =$ **done**, which means that we actually have the matching move $(d_1; d_2, \nu) \xrightarrow{p} (d_2, \nu')$.[2]

(c) Finally, if c_1 does not have type H *cmd* and $c_2 : H$ *cmd*, then the argument is essentially the same as in case (b).

□

Now we are able to prove that if (c, μ) is well typed and can terminate within at most n steps with its L variables having certain values, then $\lfloor(c, \mu)\rfloor$ can do the same, with probability at least as great. (This property was claimed, without proof, as Theorem 3.6 of [SA06].)

Let us say that a *low memory property* Φ is any property that depends only on the values of L variables. For example, if x and y are L variables, then "$x = 5$ and y is even" is a low memory property.

[1] We remark that this is the critical case that prevents us from showing that $\lfloor\cdot\rfloor$ is a strong simulation. For example, if c is $h := 2; l := 3$ and μ is $\{h = 0, l = 1\}$, then d is $l := 3$ and ν is $\{l = 1\}$. In this case the move $(c, \mu) \xrightarrow{1} (l := 3, \{h = 2, l = 1\})$ cannot be matched by the move $(d, \nu) \xrightarrow{1} (\textbf{done}, \{l = 3\})$, since the resulting memories are not low equivalent. Instead (d, ν) must match the move by doing nothing.

[2] An example illustrating this scenario is when c is (**if** 0 **then** $l := 1$ **else** $h := 2$); $l := 3$. This goes in one step to $h := 2; l := 3$, which strips to $l := 3$. In this case, $\lfloor c\rfloor =$ (**if** 0 **then** $l := 1$ **else done**); $l := 3$, which goes *in one step* to $l := 3$.

Theorem 4.3 *Let c be well typed and let Φ be a low memory property. For any n, the probability that (c, μ) terminates within n steps in a final memory satisfying Φ is less than or equal to the corresponding probability for $\lfloor (c, \mu) \rfloor$.*

Proof. By Theorem 4.2 $\lfloor \cdot \rfloor$ is a fast simulation. Now, let $T = \{(\textbf{done}, \nu) \mid \nu \text{ satisfies } \Phi\}$. It is easy to see that T is upwards closed with respect to $\lfloor \cdot \rfloor$. For if $(\textbf{done}, \nu_1) \in T$ and $(\textbf{done}, \nu_1) \lfloor \cdot \rfloor (\textbf{done}, \nu_2)$, then ν_1 satisfies Φ and $\nu_1 \sim_L \nu_2$, which implies that ν_2 also satisfies Φ. Therefore we can apply Theorem 3.2 to deduce that $\Pr((c, \mu), n, T) \leq \Pr(\lfloor (c, \mu) \rfloor, n, T)$ for every n. □

Note that we can extend Theorem 4.3 to the case of *eventually* terminating in T, since the probability of eventually terminating in T is just $\lim_{n \to \infty} \Pr((c, \mu), n, T)$.

5 Applications

Theorem 4.3 gives us the ability to quantify how the behavior of a well-typed program c can deviate from its stripped version $\lfloor c \rfloor$. But it also gives us a way to quantify how the behavior of c under memory μ can deviate from its behavior under ν, assuming that μ and ν are low equivalent. The reason is that, by Lemma 4.1, $\lfloor c \rfloor$ contains only L variables, which means that its behavior under μ must be *identical* to its behavior under ν. Hence we can build a "bridge" between (c, μ) and (c, ν):

$$(c, \mu) \overset{\text{Thm 4.3}}{\longleftrightarrow} (\lfloor c \rfloor, \lfloor \mu \rfloor) \equiv (\lfloor c \rfloor, \lfloor \nu \rfloor) \overset{\text{Thm 4.3}}{\longleftrightarrow} (c, \nu)$$

In this section, we develop several applications of these ideas.

First, suppose that c is well typed and *probabilistically total*, which means that it halts with probability 1 from all initial memories. Then (c, μ)'s nontermination bucket is empty, which implies by Theorem 4.3 that (c, μ)'s buckets are identical to $(\lfloor c \rfloor, \lfloor \mu \rfloor)$'s buckets. Similarly, (c, ν)'s buckets are identical to $(\lfloor c \rfloor, \lfloor \nu \rfloor)$'s buckets. Hence (c, μ)'s buckets are identical to (c, ν)'s buckets. So we have proved the following corollary:

Corollary 5.1 *If c is well typed and probabilistically total, then c satisfies probabilistic noninterference.*

This same result was proved in a different way as Corollary 3.5 of [SA06]; the proof there used a weak probabilistic bisimulation.

More interestingly, we can now prove an *approximate probabilistic noninterference* result for well-typed programs whose probability of nontermination is bounded.

Corollary 5.2 *Suppose that c is well typed and fails to terminate from any initial memory with probability at most p. If μ and ν are low equivalent, then the deviation between the distributions of L outcomes under μ and under ν is at most $2p$.*

Proof. Since (c, μ)'s nontermination bucket contains at most p units of water, the sum of the absolute value of the differences between the L outcome buckets of (c, μ) and of $(\lfloor c \rfloor, \lfloor \mu \rfloor)$ is at most p. Similarly for (c, ν). Hence the sum of the absolute value of the the differences between the L outcome buckets of (c, μ) and of (c, ν) is at most $2p$. □

Notice that the program in Figure 1 achieves the upper bound of this corollary. From any initial memory, this program fails to terminate with probability at most $1/2$, so here $p = 1/2$. When $h = 0$, it terminates with $l = 0$ with probability $1/2$ and terminates with $l = 1$ with probability 0. When $h = 1$, it terminates with $l = 0$ with probability 0 and terminates with $l = 1$ with probability $1/2$. Hence the deviation between the two distributions of L outcomes is $|1/2 - 0| + |0 - 1/2| = 1$, which is $2p$. In general, applying Corollary 5.2 usefully requires a good bound p on the probability of nontermination; of course such bounds may be hard to obtain.

As an application of the approximate noninterference result, notice that an adversary $\mathcal{A}$, given the final values of c's L variables, might try to distinguish between initial memories μ and ν through statistical hypothesis testing. Assuming that the probability p of nontermination is small, then the approximate noninterference property gives us a way to bound $\mathcal{A}$'s ability to do this. Similar ideas are considered (in the context of a process algebra) in the work of Di Pierro, Hankin, and Wiklicky [DPHW02].

Finally, we emphasize that Theorem 4.3 is critical to all of the main results of our earlier paper [SA06]. For instance, Theorem 4.1 of that paper is a reduction proof that shows that it is sound to give type L to the encryption of H expression, under a suitably-chosen and protected symmetric key, provided that the encryption scheme is IND-CPA secure.[3] (Notice that such a rule violates the Denning restrictions, which would classify $\mathcal{E}(h)$ as H.) Define a *leaking adversary* $\mathcal{B}$ to be a program in the language of [SA06] that tries to leak a randomly-initialized one-bit H variable h into a L variable l. If $\mathcal{B}$ succeeds with probability q, then its *leaking advantage* is $2q - 1$. (It is defined in this way because $\mathcal{B}$ can trivially succeed with probability $1/2$.) Theorem 4.1 of [SA06] shows that if a leaking adversary $\mathcal{B}$ is well typed under the Denning restrictions together with the above rule for typing encryptions, then there exists an IND-CPA adversary $\mathcal{A}$ such that

- $\mathcal{A}$ is about as efficient as $\mathcal{B}$, and
- $\mathcal{A}$'s IND-CPA advantage is at least half of $\mathcal{B}$'s leaking advantage.

The reduction works roughly as follows: $\mathcal{A}$ runs $\mathcal{B}$ with a randomly-chosen value of h; whenever $\mathcal{B}$ calls its encryption primitive $\mathcal{E}(e)$, $\mathcal{A}$ passes the pair $(0^n, e)$ to its LR oracle and returns the result to $\mathcal{B}$. When $\mathcal{B}$ terminates, $\mathcal{A}$ guesses that $b = 1$ if $\mathcal{B}$ was successful in leaking h and guesses that $b = 0$ otherwise. Now, if $b = 1$ then $\mathcal{B}$ is run *faithfully*, and if $b = 0$ then $\mathcal{B}$ is run simply as a random assignment program (since its encryptions $\mathcal{E}(e)$ just return $\mathcal{E}_K(0^n)$, a random number that has nothing to do with e). Subtlely, this introduces the possibility of nontermination when $b = 0$. Hence, to prove that $\mathcal{A}$'s IND-CPA advantage is as claimed, we need a precise characterization of the behavior of random assignment programs, as is given by Theorem 4.3. See [SA06] for the details.

[3] An IND-CPA adversary $\mathcal{A}$ [BR05] is given an *LR oracle* of the form $\mathcal{E}_K(\mathrm{LR}(\cdot, \cdot, b))$, where K is a randomly generated key and b is an internal *selection bit*, which is either 0 or 1. When $\mathcal{A}$ sends a pair of equal-length messages (M_0, M_1) to the LR oracle, it returns $\mathcal{E}_K(M_b)$. Thus when $\mathcal{A}$ sends a sequence of pairs of messages to the LR oracle, it either gets back encryptions of the *left* messages (if $b = 0$) or else encryptions of the *right* messages (if $b = 1$). $\mathcal{A}$'s challenge is to guess the value of b; the encryption scheme is IND-CPA secure if an efficient $\mathcal{A}$ cannot succeed with probability significantly better than 1/2.

6 Related Work and Conclusion

This paper has shown that, under the Denning restrictions, well-typed probabilistic programs are guaranteed to satisfy an *approximate* probabilistic noninterference property, provided that their probability of nontermination is small. The proof is based on a new notion of *fast simulation*, which builds on the work of Baier, Katoen, Hermanns, and Wolf [BKHW05] on strong and weak simulation on discrete and continuous Markov chains. The theorem that stripping is a fast simulation shows that the theory of probabilistic simulation can be applied fruitfully to the secure information flow problem, giving another proof technique in addition to the more common bisimulation-based approach of work like [LV05], [SA06], and [FR08] on languages with cryptography, and [AFG98], [SV98], [Smi03], [ACF06], [FC08] on multi-threaded languages. The recent work [AS09] on secure information flow in a distributed language also makes use of the technique of stripping and fast simulation, although in a non-probabilistic context.

Stripping is somewhat reminiscent of the work of Agat [Aga00], which proposes to eliminate external timing leaks in programs through a transformation-based approach. But our stripping operation is not an implementation technique, but rather a *thought experiment* that we use to better understand the behavior of programs under the Denning restrictions.

The theme of our paper is somewhat similar to [Mal07]. That paper uses Shannon's information theory to assign a quantitative measure to the amount of leakage caused by **while** loops. However, the focus there is not on leaks caused by nontermination, but instead on **while** loops that *violate* the Denning restrictions, for example by assigning to a L variable in the body of a **while** loop with a H guard. Nevertheless, it would be interesting to explore deeper connections between that work and this.

More closely related is [AHSS08], which explores termination-insensitive noninterference (as guaranteed by the Denning restrictions) in the context of a deterministic programming language with an **output** command. They observe that such programs can leak an unbounded amount of information (albeit slowly) by going into an infinite loop at some point within a sequence of outputs.

Acknowledgments

An early version of this work appeared as [SA07]. This work was partially supported by the National Science Foundation under grants HRD-0317692 and CNS-0831114.

References

[ACF06] M. Abadi, R.J. Corin, and C. Fournet. Computational secrecy by typing for the pi calculus. In N. Kobayashi, editor, *Fourth Asian Symposium on Programming Languages and Systems (APLAS 2006)*, volume 4279 of *Lecture Notes in Computer Science*, pages 253–269, London, November 2006. Springer Verlag.

[AFG98] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 105, Washington, DC, USA, 1998. IEEE Computer Society.

[Aga00] Johan Agat. Transforming out timing leaks. In *POPL*, pages 40–53, Boston, MA, January 2000.

[AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, 2008.

[AS09] Rafael Alpízar and Geoffrey Smith. Secure information flow for distributed systems. In *Proc. Formal Aspects of Security and Trust (FAST 2009)*, Eindhoven, Netherlands, November 2009.

[BKHW05] Christel Baier, Joost-Pieter Katoen, Holger Hermanns, and Verena Wolf. Comparative branching-time semantics for Markov chains. *Information and Computation*, 200(2):149–214, 2005.

[BR05] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. At `http://www-cse.ucsd.edu/users/mihir/cse207/classnotes.html`, 2005.

[DD77] Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[DPHW02] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. In *Proceedings15thIEEE Computer Security Foundations Workshop*, pages 1–17, Cape Breton, Nova Scotia, Canada, June 2002.

[FC08] Riccardo Focardi and Matteo Centenaro. Information flow security of multi-threaded distributed programs. In *PLAS '08: Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2008. ACM.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley & Sons, Inc., Third edition, 1968.

[FR08] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings35thSymposium on Principles of Programming Languages*, San Francisco, California, January 2008.

[Gra90] James W. Gray, III. Probabilistic interference. In *Proceedings1990IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, CA, May 1990.

[JL91] Bengt Jonsson and Kim Larsen. Specification and refinement of probabilistic processes. In *Proc. 6th IEEE Symposium on Logic in Computer Science*, pages 266–277, 1991.

[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.

[LV05] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Symposium on Fundamentals of Computational Theory*, volume 3623 of *Lecture Notes in Computer Science*, pages 365–377, Lübeck, Germany, 2005.

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In *Proceedings34thSymposium on Principles of Programming Languages*, pages 225–235, Nice, France, January 2007.

[MCN+06] Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. *Jif: Java + information flow*. Cornell University, 2006. Available at `http://www.cs.cornell.edu/jif/`.

[SA06] Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *Proc. 4th ACM Workshop on Formal Methods in Security Engineering*, pages 33–43, Fairfax, Virginia, November 2006.

[SA07] Geoffrey Smith and Rafael Alpízar. Fast probabilistic simulation, nontermination, and secure information flow. In *Proc. 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 67–71, San Diego, California, June 2007.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[Smi03] Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings16thIEEE Computer Security Foundations Workshop*, pages 3–13, Pacific Grove, California, June 2003.

[Smi06] Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings13thIEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, UK, July 2000.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings25thSymposium on Principles of Programming Languages*, pages 355–364, San Diego, CA, January 1998.

[VS97] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings10thIEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

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

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.