Fast Probabilistic Simulation, Nontermination, and Secure Information Flow

Geoffrey Smith   Rafael Alpízar

Florida International University

{smithg,ralpi001}@cis.fiu.edu

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. We consider a "stripping" operation on programs and establish a fundamental relationship between the behavior of a well-typed program and of its stripped version; to prove this relationship, we introduce a new notion of fast probabilistic simulation on Markov chains. 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.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of Programs

General Terms Security, Languages, Verification

Keywords type systems, probabilistic noninterference

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 from 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 [2], 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.

Much later, Volpano, Smith, and Irvine [16] showed the soundness of the Denning restrictions by formulating them as a type system

```
$t \overset{?}{\leftarrow} \{0,1\}$;
if $t = 0$ then (
    while $h = 1$ do skip;
    $l := 0$)
else (
    while $h = 0$ do skip;
    $l := 1$)
```

Figure 1. A random assignment program

and proving that they ensure a *noninterference* property. Much work has followed; see [10] 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 skip** (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. [14]) or forbidding assignments to L variables that sequentially follow commands whose termination depends on H variables (e.g. [12]). 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 [9] 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* [15], 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. Note that $t \overset{?}{\leftarrow} \{0,1\}$ is a *random assignment* that assigns either 0 or 1 to t, each with probability $1/2$. Assuming that h is H and t and l are L, this program satisfies the Denning restrictions and it is well typed under the typing rules that we will present in Section 2. But,

```
$t \overset{?}{\leftarrow} \{0,1\}$;
if $t = 0$ then (
    skip;
    $l := 0$)
else (
    skip;
    $l := 1$)
```

Figure 2. Stripped version of the program

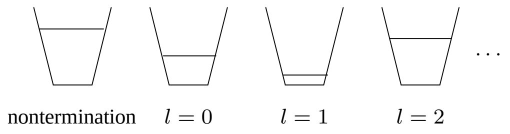
if $h = 0$, then this program terminates with $l = 0$ with probability $1/2$ and fails to terminate with probability $1/2$. And if $h = 1$, then it terminates with $l = 1$ with probability $1/2$ and fails to terminate with probability $1/2$. Thus this program does not satisfy probabilistic noninterference.

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.

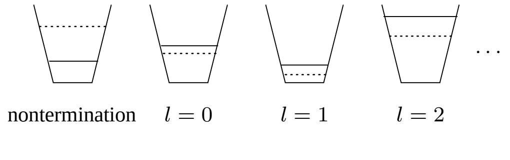
To make these intuitions precise, we define, for any well-typed program c, a "stripped" version of c, denoted by $\lfloor c \rfloor$. In $\lfloor c \rfloor$, certain subcommands of c (namely, those that make no assignments to L variables) are replaced with **skip**. For example, the stripped version of the program in Figure 1 is shown in Figure 2.

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 4.4 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, and one for nontermination. The probability of each outcome is indicated by the amount of water in each bucket. Suppose that c's buckets look like this:



Then Theorem 4.4 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:



$$
\begin{array}{rcl}
p & ::= & e \mid c \\
e & ::= & x \mid n \mid e_1 + e_2 \mid \ldots \\
c & ::= & x := e \mid x \overset{?}{\leftarrow} \mathcal{D} \mid \textbf{skip} \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & \textbf{while } e \textbf{ do } c \mid c_1 ; c_2
\end{array}
$$

Figure 3. Language Syntax

Note that Theorem 4.4 was claimed, without proof, as "Theorem" 3.6 of our earlier paper [13]. Furthermore, it was claimed there that the proof could be done using a *strong probablistic simulation* as defined by Jonsson and Larsen [5]. But that claim is incorrect; strong simulation turns out to be too restrictive for this purpose. For this reason, we here introduce a new probabilistic simulation, which we call *fast simulation*, and show that it can be used in proving Theorem 4.4.

Theorem 4.4 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 practical languages (like Jif) that allow termination channels. Second, this paper makes a technical contribution by introducing a new notion of probabilistic 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 probablistic 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 4.4, 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, in Section 6 we discuss related work and conclude. Due to space limitations, proofs and details are omitted; the full version is available at `www.cis.fiu.edu/~smithg/papers`.

2. A Random Assignment Language

In this section, we review the syntax, semantics, and type system of the simple imperative language with random assignment as defined in [13]. The language syntax is defined in Figure 3. In the 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. The command $x \overset{?}{\leftarrow} \mathcal{D}$ is a random assignment; here $\mathcal{D}$ ranges over some set of probability distributions on the integers. In examples, we use notation like $x \overset{?}{\leftarrow} \{0,1,2\}$ 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, with $\mu(e)$ denoting the value of expression e in memory μ. A *configuration* is either a pair (c, μ) or simply a memory μ.

In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ.

Because of the random assignment command, the transition relation on configurations needs to be extended with probabilities—we write $C \xrightarrow{p} C'$ to indicate that the probability of going from configuration C to configuration C' is p. The semantic rules can be found in [13]; they make the set of configurations into a (discrete-time) Markov chain [3].

Now we describe the type system for secure information flow which we will use; it simply enforces the Denning restrictions. Here are the types we will use:

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

Intuitively, 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 τ *var*. (We generally assume a single fixed Γ throughout, so in our discussions we will usually not mention Γ.) The typing and subtyping rules are omitted; they can be found in [13]. They are the same as those in [16], except for the new rule for random assignment, which says that a random assignment can be done to any variable.

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.*

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

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

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* [3] is a pair $(S, \mathbf{P})$ where

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

Also, for $T \subseteq S$, we write $\mathbf{P}(s, T)$ to denote $\sum_{s' \in T} \mathbf{P}(s, s')$, 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 [6] and Larsen and Skou [7].

DEFINITION 3.1. *Let R be an equivalence relation on S. R is a* strong bisimulation *if whenever $s_1 R s_2$ we have $\mathbf{P}(s_1, T) = \mathbf{P}(s_2, T)$ for every equivalence class T of R.*

Both strong and weak versions of bisimulation have been applied fruitfully in secure information flow analysis, for example by Gray [4], Sabelfeld and Sands [11], and Smith [12]. 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, first defined by Jonsson and Larsen [5]. Intuitively, t simulates s if it can simulate whatever s can do. Suppose s can go to some state s' with probability p. To match this, t should be able to go to one or more states t', t'', t''', …, each of which simulates s', with total probability at least 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 simulate 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 simulating 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 (from [5]):

DEFINITION 3.2. *Let R be a binary relation on S. R is a* strong simulation *if, whenever $s_1 R s_2$, there exists a function $\Delta : S \times S \rightarrow [0, 1]$ such that*

1. *$\Delta(s, s') > 0$ implies that sRs',*
2. *$\mathbf{P}(s_1, s'_1) = \sum_{s \in S} \Delta(s'_1, s)$ for all $s'_1 \in S$,*
3. *$\mathbf{P}(s_2, s'_2) = \sum_{s \in S} \Delta(s, s'_2)$ for all $s'_2 \in S$.*

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 s_2 to run "faster" than the simulated state s_1. A more flexible notion is that of *weak simulation* as defined in Definition 34 of Baier, Katoen, Hermanns, and Wolf [1]. The idea is that s_1 could make "insignificant" moves to states that are already simulated by s_2; in this case s_2 shouldn't need to make a matching move. But that definition also allows s_2 to make "insignificant" moves, which are not appropriate for us, since we want s_2 to run at least as fast as s_1. So here we develop a restricted kind of weak simulation, which we call a *fast simulation*.

Suppose that $s_1 R s_2$. We partition the states reachable in one step from s_1 into two sets, U and V. The states in V represent "insignificant" moves, and we require that s_2 itself simulates each of them. The states in U represent "significant" moves, and we require that s_2 be able to match such moves with some weight function. Intuitively, then, s_2 matches s_1's behavior either by moving or by "stuttering" for a step. Formally, we have the following definition:

DEFINITION 3.3. *Let R be a binary relation on S. R is a* fast simulation *if, whenever $s_1 R s_2$, the states reachable in one step from s_1 can be partitioned into two sets U and V such that*

1. *vRs_2 for every $v \in V$, and*
2. *letting $K = \sum_{u \in U} \mathbf{P}(s_1, u)$, if $K > 0$ then there exists a function $\Delta : S \times S \rightarrow [0, 1]$ such that*
   (a) $\Delta(u, w) > 0$ implies that uRw,
   (b) $\mathbf{P}(s_1, u)/K = \sum_{w \in S} \Delta(u, w)$ for all $u \in U$, and $\mathbf{P}(s_2, w) = \sum_{u \in U} \Delta(u, w)$ for all $w \in S$.

Notice that in condition *2(b)*, $\mathbf{P}(s_1, u)/K$ is the *conditional probability* of going from s_1 to u, given that s_1 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.)

For example, consider the Markov chain given in Figure 4, where $S = \{s_1, s_2, t_1, t_2, w\}$. Define R by $s_1 R s_2$, $t_1 R t_2$, together with sRs for every $s \in S$. Then we can show that R is a fast simulation:

- For pairs of the form sRs, we can always satisfy the requirements of Definition 3.3 by choosing U to be the set of succes-

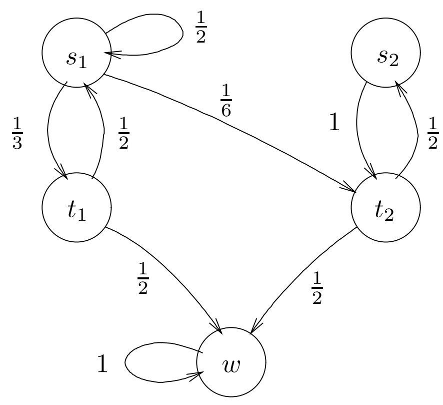


Figure 4. An example Markov chain

sors of s and V to be $\emptyset$. Then $K = 1$, and for each $u \in U$ we can choose $\Delta(u,u) = \mathbf{P}(s,u)$. It is straightforward to verify that these choices satisfy conditions *1*, *2(a)*, and *2(b)*.

- For $s_1 R s_2$ we can choose $U = \{t_1, t_2\}$ and $V = \{s_1\}$, which makes $K = \frac{1}{2}$, and we can choose $\Delta(t_1, t_2) = \frac{2}{3}$ and $\Delta(t_2, t_2) = \frac{1}{3}$.
- Finally, for $t_1 R t_2$ we can choose $U = \{s_1, w\}$ and $V = \emptyset$, which makes $K = 1$, and we can choose $\Delta(s_1, s_2) = \frac{1}{2}$ and $\Delta(w, w) = \frac{1}{2}$.

We remark that every strong simulation is also 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 Baier et al. [1].

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 [1], we can calculate $\Pr(s, n, T)$ with a recurrence:

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

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 s, n, and T.*

We now proceed to the key theorem about fast simulation; its proof is similar to the proof of Theorem 54 of [1], 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.*

We can illustrate Theorem 3.2 by considering the Markov chain in Figure 4 and its fast simulation R defined above. Because $t_1 R t_2$ and $\{w\}$ is upwards closed with respect to R, Theorem 3.2 tells us that $\Pr(t_1, n, \{w\}) \leq \Pr(t_2, n, \{w\})$, for every n.

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 Baier et al. [1].

Now that we have developed fast simulation on abstract Markov chains, we are ready to apply it to our study of secure information flow. We do this next.

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 variables; it is easy to see that this is the same as eliminating subcommands of type H *cmd*. More precisely, we have the following definition, which first appeared in [13]:

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

- $\lfloor x := e \rfloor = x := e$
- $\lfloor x \overset{?}{\leftarrow} \mathcal{D} \rfloor = x \overset{?}{\leftarrow} \mathcal{D}$
- $\lfloor$**if** e **then** c_1 **else** $c_2 \rfloor =$ **if** e **then** $\lfloor c_1 \rfloor$ **else** $\lfloor c_2 \rfloor$
- $\lfloor$**while** e **do** $c_1 \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$
- $\lfloor c_1; c_2 \rfloor = \lfloor c_1 \rfloor; \lfloor c_2 \rfloor$

We begin with a simple lemma:

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

We now define a binary relation R_L on configurations, prove that it is a fast simulation, and then show that $(c, \mu) R_L (\lfloor c \rfloor, \mu)$ for every well-typed c. We first define R_L on commands and then extend it to configurations:

DEFINITION 4.2. *If c and d are well typed, then we say that $c R_L d$ if this can be proved from the following six rules:*

1. *$c_1 R_L$***skip***, if $c_1 : H$ cmd.*
2. *$(x := e) R_L (x := e)$.*
3. *$(x \overset{?}{\leftarrow} \mathcal{D}) R_L (x \overset{?}{\leftarrow} \mathcal{D})$.*
4. *(***if*** e ***then*** c_1 ***else*** $c_2) R_L ($***if*** e ***then*** d_1 ***else*** $d_2)$, if $e : L$, $c_1 R_L d_1$, and $c_2 R_L d_2$.*
5. *(***while*** e ***do*** $c_1) R_L ($***while*** e ***do*** $d_1)$, if $e : L$ and $c_1 R_L d_1$.*
6. *$(c_1; c_2) R_L (d_1; d_2)$, if $c_1 R_L d_1$ and $c_2 R_L d_2$.*

We extend R_L to configurations with the following two rules:

1. *$\mu R_L \nu$, if $\mu \sim_L \nu$.*
2. *$(c, \mu) R_L (d, \nu)$, if $c R_L d$ and $\mu \sim_L \nu$.*

Here is the key theorem about R_L:

THEOREM 4.2. *R_L is a fast simulation.*

We need one more simple lemma:

LEMMA 4.3. *For any well-typed c, $c R_L \lfloor c \rfloor$*

Now we are able to prove that if c is well typed and can terminate within at most n steps with its L variables having certain values, then $\lfloor c \rfloor$ can do the same, with probability at least as great.

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.

THEOREM 4.4. *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 $(\lfloor c \rfloor, \mu)$.*

We can extend this result 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 4.4 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):

$$(c, \mu) \overset{\text{Thm 4.4}}{\longleftrightarrow} (\lfloor c \rfloor, \mu) \equiv (\lfloor c \rfloor, \nu) \overset{\text{Thm 4.4}}{\longleftrightarrow} (c, \nu)$$

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 4.4 that (c, μ)'s buckets are identical to $(\lfloor c \rfloor, \mu)$'s buckets. Similarly, (c, ν)'s buckets are identical to $(\lfloor c \rfloor, \nu)$'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 [13]; 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 $2p$.*

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.

Finally, we emphasize that Theorem 4.4 is critical to all of the main results of our earlier paper [13]. The key reduction (Theorem 4.1) of that paper shows that it is sound to give type L to the encryption of a H expression (under a suitably-chosen and protected symmetric key), provided that the encryption scheme is IND-CPA secure. The correctness of the reduction depends crucially on Theorem 4.4's precise characterization of the behavior of random assignment programs.

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, which relies on a new notion of fast simulation, shows that the theory of probabilistic simulation can be applied fruitfully to the secure information flow problem, allowing us to make quantitative statements about the effects of nontermination.

The theme of this paper is somewhat similar to Malacaria's recent paper [8]. 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.

Acknowledgments

This work was partially supported by the National Science Foundation under grant HRD-0317692.

References

[1] C. Baier, J.-P. Katoen, H. Hermanns, and V. Wolf. Comparative branching-time semantics for Markov chains. *Information and Computation*, 200(2):149–214, 2005.

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

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

[4] J. W. Gray, III. Probabilistic interference. In *Proc. 1990 IEEE Symp. on Security and Privacy*, pages 170–179, Oakland, CA, May 1990.

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

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

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

[8] P. Malacaria. Assessing security threats of looping constructs. In *Proceedings 34th Symposium on Principles of Programming Languages*, pages 225–235, Nice, France, Jan. 2007.

[9] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. *Jif: Java + information flow*. Cornell University, 2006. Available at `http://www.cs.cornell.edu/jif/`.

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

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

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

[13] G. Smith and R. 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, Nov. 2006.

[14] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

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

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