Probabilistic Noninterference for Multi-threaded Programs[*]

Andrei Sabelfeld David Sands

Department of Computer Science
Chalmers University of Technology and the University of Göteborg
412 96 Göteborg, Sweden
E-mail: {andrei,dave}@cs.chalmers.se

Abstract

We present a probability-sensitive confidentiality specification – a form of probabilistic noninterference *– for a small multi-threaded programming language with dynamic thread creation. Probabilistic covert channels arise from a scheduler which is probabilistic. Since scheduling policy is typically outside the language specification for multi-threaded languages, we describe how to generalise the security condition in order to define robust security with respect to a wide class of schedulers, not excluding the possibility of deterministic (e.g., round-robin) schedulers and program-controlled thread priorities. The formulation is based on an adaptation of Larsen and Skou's notion of* probabilistic bisimulation. *We show how the security condition satisfies compositionality properties which facilitate straightforward proofs of correctness for, e.g., security type systems. We illustrate this by defining a security type system which improves on previous multi-threaded systems, and by proving it correct with respect to our stronger scheduler-independent security condition.*

1. Introduction

1.1. Motivation

When is an untrusted program safe to use? One aspect of safety is confidentiality. Given you have some confidential (high) data and some public (low) data in your computer, you want to make sure the attacker – the supplier of the untrusted code – will not learn anything about your personal data, despite the fact that the *application* (e.g., a spreadsheet) may require legitimate access to the confidential data in order to perform its task, and legitimate communication with the supplier of the code (e.g., a registration process for all users).

[*]This paper appeared in the Proceedings of the 13th IEEE Computer Security Foundations Workshop, Cambridge, England, July 3–5, 2000.

In this study we assume that the attacker is external to the (trusted) system upon which the program is run. Our aim is to specify when a program is safe to run – from the point of view of its confidentiality properties – with an aim to providing automatic methods for certifying programs prior to execution.

This paper describes some security specifications for multi-threaded programs executing on a single processor. Programs which execute several concurrent processes not only introduce nondeterminism, but also the potential for an attacker to observe the internal timing behaviours of programs, via the effect that the running time of commands might have on the scheduler's choice of when various threads are executed. Since a scheduler's behaviour might be probabilistic, this opens up an opportunity for the attacker to set up a probabilistic covert channel whereby information is obtained by statistical inferences drawn from the relative frequency of outcomes of a repeated computation.

We describe how *confidentiality* (the security property of not having insecure information flows) can be specified operationally for a simple imperative language with dynamic thread creation, and how the compositional nature of the security condition facilitates a straightforward proof of correctness of type-based program analyses.

An important contribution is to abstract over the scheduler's behaviour, so as to obtain a notion of security which holds for a broad class of schedulers, possibly including both probabilistic and deterministic behaviours. We illustrate this by defining a security type system which improves on a previous system of Volpano and Smith [19] for a simpler language), and by proving it correct with respect to our stronger explicitly scheduler-independent security condition.

1.2. Classification of Insecure Information Flows

It will be useful to informally classify a number of insecure information flows that might arise in programs. Let us restrict ourselves to a simplified setting: two variables h and l of high and low confidentiality respectively. Consider the examples of insecure information flows that security specifications and specific program analyses might attempt to rule out:

Direct: $l := h$
Direct information flow from the initial value of h to the final value of l.

Indirect: $\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$
Secret information *indirectly* leaks from the initial value of h to the final value of l.

Termination behaviour: $\mathsf{if}\ h = 1\ \mathsf{then\ loop}$
If the program terminates, the attacker knows that h was not 1.

Probabilistic:
$h := h\ \mathsf{mod}\ 2; (l := h \mathbin{[]_{\frac{1}{2}}} (l := 0 \mathbin{[]_{\frac{1}{2}}} l := 1))$
where $[]_{\frac{1}{2}}$ is a coin-flip choice operator. Here there is no information flow if we only consider the *possible* behaviours of the system, but the final value of l will reveal information about h with a probability of $\frac{3}{4}$.

Externally observable timing: $\mathsf{if}\ h = 1\ \mathsf{then\ sleep}\ 100$
The attacker with a stop-watch can learn information about h. This paper does not consider externally observable timing.

Internally observable timing:
$((\mathsf{if}\ h = 1\ \mathsf{then\ sleep}\ 100); l := 1) \mid l := 0$
where $\mid$ denotes the concurrent execution of the two commands. With many schedulers, the timing leaks will translate into an effect on the (distribution of) final value(s) of l. The information flow in this case is a combination of the timing and potentially probabilistic flows in the case that the scheduler exhibits stochastic behaviour. The ability to dynamically create threads also leads to similar possibilities.

1.3. Related Work

There is a substantial body of work in the security community in studying definitions and reasoning principles relating to information flow and confidentiality. For an overview see, e.g., [15]. The basic approach we take is based on an extentional characterisation usually known as *noninterference*. The particular flavour of noninterference studied here is most closely related to Gray's notion of *P-Restrictiveness* [10], since it aims to eliminate dependence between classified data and the probability distribution of unclassified outputs.

The use of program analysis as a means to eliminate insecure flows was pioneered by Denning [6, 7]. A modern incarnation of this work which recasts Denning's analysis as a type system and proves its correctness is [20].

Semantically, our security condition is founded on a probabilistic notion of bisimulation. Focardi and Gorrieri have promoted the use of (non-probabilistic) bisimulation in formalising and analysing security conditions in process calculus setting (see, e.g., [8]).

Focusing on concurrent imperative programs, there have been some recent considerations of concurrent programming languages. Banâtre, Bryce and Le Métayer [4] consider a programming language with static process structure and CSP-like communication primitives (see also [2]), although there is no extensional specification of correctness in these studies, and probabilistic information flows are not considered. Heintze and Riecke [11] present an analysis for a security-oriented lambda calculus, extended with threads and imperative features. Although they prove a noninterference property for the functional core of the language, they (quote) *"have not proved a noninterference theorem for the concurrent setting because ... the notion is unclear in the concurrent setting"*. Smith and Volpano [17] extend their previous work ([20]) with static threads. They prove a possibilistic correctness result for their analysis, and note that the analysis does not eliminate probabilistic covert channels (other than by making it overly restrictive). A later work [19] shows how to lift this restriction and proves some correctness properties based on a probabilistic semantics. We will return to consider this last work in more detail. In [16] we showed how partial equivalence relations (pers) and denotational semantics could capture security properties in sequential languages, including probabilistic information flows (noninterference-style) with the help of probabilistic powerdomains. In principle this machinery should be sufficient to handle concurrent languages, but the mathematical overheads are high. In this paper we also use partial equivalence relations, but this time operationally-based using probabilistic-bisimulation-like relations.

1.4. Overview

The rest of the paper is organised as follows. **Section 2** introduces syntax and probabilistic semantics of a multi-threaded language. **Section 3** defines the security condition based on probabilistic bisimulation. In **Section 4**, we introduce a scheduler-independent security condition. We go on to strengthen it by defining a strong security and illustrate its properties in **Section 5**. **Section 6** considers the applicability of the conditions to proving the soundness of compositional security analyses. **Section 7** concludes.

2. A Simple Concurrent Language

In order to be concrete – and in order to explore the compositionality properties of some of our security conditions – we consider a very simple shared-variable language described by the grammar in Figure 1. Let $C, D, E, \dots$ range over commands Com, and let $\vec{C}$ denote a vector of commands of the form $\langle C_0 \dots C_{n-1}\rangle$. Vectors $\vec{C}, \vec{D}, \vec{E}, \dots$ range over $\vec{Com} = \cup_{n\in\mathbb{N}} Com^n$, the set of thread pools (or programs).

We consider a small-step semantics for the language. A *configuration* c $(d, e, \dots)$ is a pair of a command and a state (memory). In this paper, a *state* $s \in \mathbf{St}$ is a finite mapping from variables to values. The set of variables is partitioned into high and low security classes. h and l will denote typical high and low variables respectively. In concrete examples we will often take just variables h and l, and in this case we will write the state as a pair. Define low-equivalence $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same. Let $Config$ denote $\vec{Com} \times \mathbf{St}$. The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 2. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\rightharpoonup$ transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s' |\!\rangle$, which means termination with the final state s', or $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| C'\vec{D}, s' |\!\rangle$. Here, one step of computation starting with command C in a state s gives a new main thread C', (possibly empty) vector of spawned processes $\vec{D}$ and a new state s'. Command $\mathsf{fork}(C\vec{D})$ dynamically creates a new vector of processes $\vec{D}$ which run in parallel with the main thread C. This has the effect of adding the vector of processes to the configuration.

The concurrent part of the semantics is presented in Figure 3. The $\rightarrow_p$ transitions are probabilistic. The scheduler modelled here is purely probabilistic; it chooses to execute one of the threads of $\vec{C}$ with equal probability. We call this a *uniform* scheduler. In Section 4 we will parameterise over the choice of scheduler. Write $\langle\!| \vec{C}, s |\!\rangle \rightarrow \langle\!| \vec{D}, s' |\!\rangle$ whenever $\exists p > 0.\, \langle\!| \vec{C}, s |\!\rangle \rightarrow_p \langle\!| \vec{D}, s' |\!\rangle$.

As is standard, in this paper we use multiset comprehensions $\{\!|p| \dots |\!\}$ when summing probabilities.

Labelling the transitions in the rules for parallel composition is performed in order to ensure that the sum of probabilities of transitions from a given nonterminal configuration is one. If there were no labels or rule Σ, then there would be $\langle\!| \langle \mathsf{skip} \mid \mathsf{skip}\rangle, s |\!\rangle \rightarrow_{1/2} \langle\!| \langle \mathsf{skip}\rangle, s |\!\rangle$ rather than $\langle\!| \langle \mathsf{skip} \mid \mathsf{skip}\rangle, s |\!\rangle \rightarrow_{1} \langle\!| \langle \mathsf{skip}\rangle, s |\!\rangle$, where $\mid$ is used for separating threads in a thread pool. That is due to the fact that transition $\langle\!| \langle \mathsf{skip}, \mathsf{skip}\rangle, s |\!\rangle \rightarrow_{1/2} \langle\!| \langle \mathsf{skip}\rangle, s |\!\rangle$ could be performed by two different rules. That cannot be observed without the labelling or by making similar distinctions between different ways of obtaining the same transition. Similar construc-

$$
\begin{aligned}
C ::= {} & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \\
& \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C\vec{D})
\end{aligned}
$$

Figure 1. Command syntax

$$\langle\!| \mathsf{skip}, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s |\!\rangle$$

$$\frac{\langle\!| Exp, s |\!\rangle \downarrow n}{\langle\!| Id := Exp, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, [Id = n]s |\!\rangle}$$

$$\frac{\langle\!| C_1, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \rightharpoonup \langle\!| C_2, s' |\!\rangle} \qquad \frac{\langle\!| C_1, s |\!\rangle \rightharpoonup \langle\!| C_1'\vec{D}, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \rightharpoonup \langle\!| (C_1'; C_2)\vec{D}, s' |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \rightharpoonup \langle\!| C_1, s |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \rightharpoonup \langle\!| C_2, s |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \rightharpoonup \langle\!| C; \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s |\!\rangle}$$

$$\langle\!| \mathsf{fork}(C\vec{D}), s |\!\rangle \rightharpoonup \langle\!| C\vec{D}, s |\!\rangle$$

Figure 2. Small-step deterministic semantics of commands

$$\frac{\langle\!| C_i, s |\!\rangle \rightharpoonup \langle\!| \vec{C}, s' |\!\rangle}{\langle\!| \langle C_0 \dots C_{n-1}\rangle, s |\!\rangle \rightarrow^{i}_{\frac{1}{n}} \langle\!| \langle C_0 \dots C_{i-1}\vec{C}C_{i+1} \dots C_{n-1}\rangle, s' |\!\rangle}$$

$$(\Sigma)\,\frac{p = \sum \{\!| q \mid \langle\!| \vec{C}, s |\!\rangle \rightarrow^{i}_{q} \langle\!| \vec{D}, s' |\!\rangle |\!\}}{\langle\!| \vec{C}, s |\!\rangle \rightarrow_p \langle\!| \vec{D}, s' |\!\rangle}$$

Figure 3. Uniform probabilistic semantics of thread pools

tions can be found in other probabilistic semantics.

The probabilistic semantics satisfies the following "soundness" properties for all nonterminal programs $\vec{C}$:

$$(i)\ \forall s.\ \sum\{\!|p|\langle\!|\vec{C},s|\!\rangle \rightarrow_p \langle\!|\vec{D},s'|\!\rangle|\!\} = 1,$$
$$(ii)\ \langle\!|\vec{C},s|\!\rangle \rightarrow_p \langle\!|\vec{D},s'|\!\rangle, \langle\!|\vec{C},s|\!\rangle \rightarrow_q \langle\!|\vec{D},s'|\!\rangle \Longrightarrow p = q.$$

Observe that rule Σ guarantees property (ii) while property (i) must be independently enforced (and is dependent on expressions being total). Given these soundness conditions, the semantics can be viewed as an *unlabelled probabilistic transition system* [13], or a simple probabilistic automaton.

3. Probabilistic Noninterference

3.1. Background

Now that we have defined the syntax and semantics of the multi-threaded language, we are ready to specify what it means for a program to be secure. We are taking the view of *noninterference* to express the confidentiality property of a program. The central idea of noninterference [9] is that a program is secure whenever varying the initial values of high variables cannot change the low-observable (observable by the attacker) behaviour of the program.

A natural covert channel generalising the nontermination channel is the timing channel. A realistic version of noninterference should include timing as a possibly low-observable action. The first step in the direction of incorporating timing into noninterference was made by Volpano and Smith [19]. Volpano and Smith's noninterference is based on a lock-step execution of *probabilistic states*. A distribution μ of a set X (write $\mu \in \mathcal{D}(X)$) is a function $\mu \in X \rightarrow [0,1]$ such that $\sum_{x\in X} \mu x = 1$. A probabilistic state μ is a distribution of configurations $\mu \in \mathcal{D}(\vec{C} \times \mathbf{St})$. The computation is modelled by a Markov chain of probabilistic states. The initial probabilistic state is $\langle\!|\vec{C},s|\!\rangle \mapsto 1$ where s is the initial memory and $\vec{C}$ is a static pool of threads (they do not handle new thread generation). Every following probabilistic state μ' is computed by multiplying the row vector of probabilities from μ by the stochastic transition matrix T. The rows and columns of T correspond to possible configurations reachable from the initial one. There might be a countably infinite number of such states, so the matrix might be countably infinite. The elements $T(i,j)$ of T are the probabilities of a transition from configuration i to configuration j.

From the point of view of the low observer two probabilistic states are indistinguishable (write $\sim_l$) iff projecting out high variables (which involves collapsing several probabilistic states into one in case the commands and low parts of the memories are the same; see [19] for the formal definition) makes the two states the same.

From the main correctness theorem we can extract the following probabilistic noninterference condition for a program $\vec{C}$ with stochastic transition matrix T:

$$\mu_1 \sim_l \mu_2 \text{ implies } \mu_1 T \sim_l \mu_2 T. \qquad (*)$$

While intuitive and adequate to prove the security of Volpano and Smith's type-system based analysis as an analysis-independent *definition* of security it suffers from the three drawbacks:

Noncompositional The probabilistic states are distributions of configurations (elements of $\mathcal{D}(\vec{C} \times \mathbf{St})$). This makes compositional reasoning at the command level impossible – commands are parts of configurations together with memories, i.e., the meaning of a term contains syntax.

Imprecise Since syntactic equality for commands is involved in the definition of low-equivalence, there is also a loss of precision. Consider the following example (or indeed any program with conditionals and no low variables):

$$\mathsf{if}\ h < 0\ \mathsf{then}\ h := h+1\ \mathsf{else}\ h := h-1.$$

Intuitively, for the low observer this program has the same behaviour regardless of the initial value of h – since the program does not modify l. However, it is considered insecure if one takes $(*)$ as a definition. Indeed, take $\mu_1 = \{\langle\!|\mathsf{if}\ h<0\ \mathsf{then}\ h := h+1\ \mathsf{else}\ h := h-1,(7,0)|\!\rangle \mapsto 1\}$ and $\mu_2 = \{\langle\!|\mathsf{if}\ h<0\ \mathsf{then}\ h := h+1\ \mathsf{else}\ h := h-1,(-7,0)|\!\rangle \mapsto 1\}$. Note $\mu_1 \sim_l \mu_2$. Now, $\mu_1 T = \{\langle\!|h := h-1,(7,0)|\!\rangle \mapsto 1\}$ and $\mu_2 T = \{\langle\!|h := h+1,(-7,0)|\!\rangle \mapsto 1\}$. After projecting out the high variable we are still left with syntactically different commands which implies $\mu_1 T \not\sim_l \mu_2 T$. To make the above example secure according to $(*)$, Volpano and Smith require that any such conditional with a high guard be executed atomically.

Scheduler Specific The security condition is specific to a particular scheduler, namely the scheduler with uniform probability of picking every thread. If the scheduler is changed then secure programs may become insecure. [1]

In the remainder of this section we begin by addressing the question of *precision* in the definition of security. The central problem of $(*)$ is that it does not abstract sufficiently from the syntactic program state. To abstract from the state we should factor in this (probabilistic) equivalence,

[1] Note that this is not a problem for the actual security type system presented by Volpano and Smith; the proofs of correctness of their type system are easily seen to go through with other choices of probabilities in the transition matrix.

in the manner of Gray's P-Restrictiveness condition [10]. We will develop probabilistic noninterference criteria based on probabilistic bisimulation. The idea is based on the definition of bisimulation on probabilistic transition systems [13, 3].

3.2. Relational Definitions and Notation

Let us first introduce some standard definitions and notation for relations. Let $Rel(A)$ denote the set of binary relations on A. If $R \in Rel(A)$ is an equivalence relation then A/R is the set of $R-$equivalence classes, $A/R \subseteq \wp(A)$. Id_A denotes the identity relation on some set A; we write just Id when A is clear from the context. As usual, R^+ denotes the transitive closure of R and R^* denotes the transitive and reflexive closure of R. For binary relations $P \in Rel(A)$, $Q \in Rel(B)$ we define the relation $P \times Q$ on $A \times B$ by

$$(x, y)\ P \times Q\ (x', y') \iff x\ P\ x'\ \&\ y\ Q\ y'.$$

A *partial equivalence relation* (per) on a set A is a binary relation on A which is *symmetric* and *transitive*. If P is such a per let $dom(P)$ denote the domain of P, given by

$$dom(P) = \{x \in A \mid x\ P\ x\}.$$

Note that the domain and range of a per P are both equal to $dom(P)$ (so for any $x, y \in A$, if $x\ P\ y$ then $x\ P\ x$ and $y\ P\ y$), and that the restriction of P to $dom(P)$ is an equivalence relation. Clearly, an equivalence relation is just a per which is reflexive (so $dom(P) = A$).

3.3. Probabilistic Bisimulation

In this subsection we recall the definition of probabilistic bisimulation, specialising it to the case of unlabelled systems.

An *unlabelled probabilistic transition system* (following [13]) is a set of states $\mathbb{S}$ and a set of transitions of the form $\mathbb{s} \to \mu$ where $\mathbb{s} \in \mathbb{S}$ and $\mu \in \mathcal{D}(\mathbb{S})$. The probabilistic semantics corresponds to a transition system where probabilistic transition states are identified with configurations. A small-step transition from a configuration to a distribution on configurations $\langle\!|\vec{C}, s|\!\rangle \to \mu$ where $\mu \in \mathcal{D}(Config)$ can be viewed as a transition in the unlabelled probabilistic transition system such that $\mu(\langle\!|\vec{D}, s'|\!\rangle) = p$ iff $\langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle$.

Define small-step semantics transitions from a configuration c to a *set* of configurations S by

$$c \to_p S \iff p = \sum \{\!|q | c \to_q d, d \in S|\!\}.$$

Inspired by the definition of Larsen and Skou, let us define probabilistic bisimulation on program configurations.

Definition 1 ([13]) *An equivalence relation $R \in Rel(Config)$ is a* probabilistic bisimulation *if whenever $c\ R\ d$ then*

$$\forall S \in Config/R.\ c \to_p S \Longrightarrow d \to_p S.$$

Two configurations are probabilistically bisimilar if there is a *probabilistic bisimulation* which relates them. In the remainder of this article we will use a weakening of the definition of probabilistic bisimulation by relaxing the reflexivity condition.

Definition 2 *A per $R \in Rel(Config)$ is a* partial probabilistic bisimulation *if whenever $c\ R\ d$ then*

$$(i)\ \forall S \in Config/R.\ c \to_p S \Longrightarrow d \to_p S,$$
$$(ii)\ c \to_0 Config \setminus dom(R).$$

The partiality introduced here can be thought of as simply removing "unreachable" parts of the relation. The modification will be useful when we specify security, but otherwise does not significantly change the definition. Note that any probabilistic bisimulation is also a partial probabilistic bisimulation. In the other direction we have:

Proposition 1 *R is a partial probabilistic bisimulation implies R^* is a probabilistic bisimulation.*

From this it follows that any two configurations which are contained in some partial probabilistic bisimulation R are also probabilistically bisimilar, and vice-versa.

Since we are interested in studying programs rather than configurations, we recover an equivalence relation on programs (vectors of commands) as follows.

Definition 3 *A per $R \in Rel(\vec{Com})$ is a* partial probabilistic bisimulation on thread pools *iff $R \times Id_{\mathbf{St}}$ is a partial probabilistic bisimulation on configurations.*

We write $c \sim d$ (resp. $C \sim D$) iff there exists a partial probabilistic bisimulation on configurations (commands) relating c and d (C and D).

It is possible to show that both $\sim$ relations are themselves (partial) probabilistic bisimulations (c.f. [3]). We sketch a proof of this fact in Appendix A.

The partial probabilistic bisimulation on thread pools could form the basis of a compositional semantics – although one must at least restrict it to relations between equal-length vectors to obtain compositionality. We will not pursue this investigation further in this paper since we are not particularly interested in the uniform scheduler.

3.4. Noninterference Based on Partial Probabilistic Bisimulation

Our aim at this point is to construct a modification of probabilistic bisimulation that reflects the "equivalence" on program behaviour visible for the attacker. The intuition behind the construction is that a program is secure iff for any two states which differ only in the values of high variables, two configurations containing the program and each of the states, execute in such a way that their behaviour is indistinguishable (or "low-equivalent") from the attacker's observation of the low parts of the state and the probability with which they occur. Let us formally define such a "low-equivalence" as a partial probabilistic bisimulation. The development parallels Section 3.3, with the modification that the states of equivalent configurations can be different in the high components.

Definition 4 *A per $R \in Rel(\vec{Com})$ is a* partial probabilistic low-bisimulation *iff $R \times (=_L)$ is a partial probabilistic bisimulation on configurations.*

Write $C \sim_L D$ (C and D are *low-bisimilar*) iff there exists a partial probabilistic low-bisimulation relating programs C and D.

There is an alternative and equivalent way to define this partial probabilistic low-bisimulation directly in terms of the operational semantics:

Proposition 2 *A per R is a partial probabilistic low-bisimulation on commands iff whenever $\vec{C}\ R\ \vec{D}$ then*

$$
\begin{aligned}
&\forall s_1 =_L s_2 . \langle \vec{C}, s_1 \rangle \to \langle \vec{C'}, s'_1 \rangle \implies \\
&\quad \exists \vec{D'}, s'_2 . \langle \vec{D}, s_2 \rangle \to \langle \vec{D'}, s'_2 \rangle \\
&\qquad (i) \sum \{\!|p | \langle \vec{C}, s_1 \rangle \to_p \langle \vec{S}, s \rangle, \vec{S} \in [\vec{C'}]_R, s =_L s'_1 |\!\} = \\
&\qquad\quad \sum \{\!|p | \langle \vec{D}, s_2 \rangle \to_p \langle \vec{S}, s \rangle, \vec{S} \in [\vec{D'}]_R, s =_L s'_2 |\!\} \\
&\qquad (ii)\ \vec{C'}\ R\ \vec{D'}, s'_1 =_L s'_2,
\end{aligned}
$$

where $[\vec{E}]_R$ stands for the $R-$equivalence class which contains $\vec{E}$.

With this formulation the (informal) connection to Gray's P-restrictiveness condition becomes clearer.

Analogously to Section 3.3, we obtain the result that $\sim_L$ is itself a partial probabilistic low-bisimulation and it is the greatest fixed point of the corresponding functor. We are now ready for the culmination of this section, the security specification for a thread pool:

$$\vec{C} \text{ is } \textit{secure} \iff \vec{C} \sim_L \vec{C}.$$

The observations about proof techniques for $\sim$ from Appendix A hold for the case of $\sim_L$ as well.

4. Scheduler-independent Noninterference

The security specification above captures the probabilistic and internal timing covert channels. We have so far presumed the uniform scheduler is used for choosing the next thread. However, the scheduler is typically not specified by the language definition, and may vary from implementation to implementation. In different implementations, different probabilistic policies of a scheduler can be used, and these specific policies can be exploited by the attacker. Under the worst case assumption the security condition should be scheduler-independent – in other words we should assume that the attacker chooses the scheduler. In this section, we generalise the security condition to be robust with respect to any particular scheduler used – for a reasonable class of schedulers. The schedulers in question will be defined as a function from the history of the computation (and the low variables) to a probability distribution on the set of live threads.

Let us start with an example that illustrates that the uniform scheduler assumption does not imply security for other schedulers. Suppose that h is a boolean. Consider the following program: $l := h \mid l := \neg h$. While this is a secure program for the uniform scheduler[2] that picks threads with equal probability, it is insecure for any other scheduler. For example, a round-robin scheduler might execute threads in a deterministic order. Although the attacker may not know this in advance, he may easily be able to determine it with reasonable certainty (using a program which does not touch high data), and thus deduce the value of h. This motivates the introduction of a *scheduler-independent low-bisimulation* ($\approx_L$).

4.1. Semantics with Schedulers

In order to obtain a security condition which is robust with respect to schedulers we extend the semantics of the language to explicitly schedule threads.

But what is a scheduler? Abstractly we will take a scheduler to be a mechanism for selecting threads which itself satisfies some noninterference property, i.e., its behaviour is independent of high data. Why should a scheduler have access to the state at all? By allowing the scheduler to depend on the low variable, it becomes possible to model threads with *dynamic priorities*, i.e., where the program can control the scheduler's behaviour using the values of certain distinguished low program variables.

The scheduler needs more than the low part of the state however. A minimum requirement of any reasonable scheduler is that it also knows the number of threads from which it can choose. This information will be part of the *history* of the computation so far. A scheduler will be defined to

[2] Following the execution of the command with the initial state (h_0, l_0), we have $\langle l := h \mid l := \neg h, (h_0, l_0) \rangle \to_{1/2} \langle l := \neg h, (h_0, h_0) \rangle \to_1 \langle \langle \rangle, (h_0, \neg h_0) \rangle$ and $\langle l := h \mid l := \neg h, (h_0, l_0) \rangle \to_{1/2} \langle l := h, (h_0, \neg h_0) \rangle \to_1 \langle \langle \rangle, (h_0, h_0) \rangle$. Since the probabilities of transitions as well as timing are mirrored in the two transition sequences, varying h_0 from True to False will affect no low-observable behaviour (neither low-observable probabilities nor timing).

be a function with arguments (i) the *history* of the computation so far, and (ii) the low part of a state, and yielding as its result a probability distribution on the number of threads currently in the configuration.

Let us first inductively define the set of possible schedule histories $Hist$. A *history* H is a sequence of pairs $H \in (\mathbb{N} \times \mathbb{N})^*$, representing information about the computation steps so far. In each pair of the sequence, the first component is the index of the thread last chosen for computation, and the second component is the total number of threads that remained in the configuration after that thread's computation step. Not all sequences describe valid histories. In any two adjacent pairs in a history, $\ldots(i,m)(j,n)\ldots$, the tread j that is selected must be in the range $0\ldots m-1$, and the number of threads remaining after j's step (n) cannot be smaller than $m-1$, since at most one thread can die in any computation step (although many may be created). Assuming without loss of generality that a computation always starts with a single thread and the empty history, we have the following inductive rules that define $Hist$:

$$\epsilon \in Hist \qquad (0,n) \in Hist$$

$$\frac{H(i,m) \in Hist}{H(i,m)(j,n) \in Hist}(j \le m-1 \le n)$$

Define the number of live processes for a given history by

$$live(\epsilon) = 1, \qquad live(H(i,n)) = n.$$

A scheduler σ is any function that given a history H and the low part l of a state returns a probability distribution on live processes. The dependent type of $\sigma(H,l)$ is

$$\sigma(H,l) \in \mathcal{D}(\{0\ldots live(H)-1\}).$$

Given a scheduler σ not all histories are feasible under σ. Define the set of *σ-feasible* histories $Hist_\sigma \subseteq Hist$ by

$$\epsilon \in Hist_\sigma$$

$$\frac{H \in Hist_\sigma \quad H(i,m) \in Hist \quad \exists l.\,\sigma(H,l)i > 0}{H(i,m) \in Hist_\sigma}$$

At each step of a computation with low-state l, only those threads can be picked for which $\sigma(H,l) > 0$. $Hist_\sigma$ will be used in the definition of σ-specific low-bisimulation – although it should be noted that $Hist_\sigma$ is still, in general, a superset of the histories which are *actually* feasible since not all l are necessarily feasible.

To introduce the scheduler to the semantics, we need to update the computation history with each successive computation step. We extend configurations to contain a history along with a command vector and a state. An initial configuration has the form $\langle\!| \epsilon, C, s |\!\rangle$. Now we replace the semantics of thread pools in Figure 3 by the rule

$$\frac{\langle\!| C_i, s |\!\rangle \rightharpoonup \langle\!| \vec{C}, s' |\!\rangle}{\langle\!| H, \langle C_0 \ldots C_i \ldots C_{n-1}\rangle, s |\!\rangle \rightarrow_p \langle\!| H', \langle C_0 \ldots \vec{C} \ldots C_{n-1}\rangle, s' |\!\rangle}$$

where $H' = H(i, n+|\vec{C}|-1), p = \sigma(H,l)\,i$ and $s = (h,l)$ for some h. The history is updated with the number of the thread being picked and the new thread-pool size. In a context where we are discussing several different schedulers, we will explicitly label the transition arrow thus $\rightarrow^\sigma_p$.

This model of the semantics with schedulers is general enough to describe any scheduler that uses the full history of thread creation/deletion to generate probabilities for picking the threads. Note however that it cannot model a scheduler which uses real-time. Consider some examples of common schedulers. Any scheduler must pick the first (and the only) thread given the empty history, so we can safely omit this case in the following examples.

For any history H, the deterministic *round robin* scheduler is defined by

$$roundr(H(i,n),l)\,x = \begin{cases} 1, & \text{if } x = (i+1)\ mod\, n, \\ 0, & \text{otherwise.} \end{cases}$$

A round-robin scheduler which chooses the same thread for several steps before switching would need to look further back in the history.

The *uniform scheduler* presented in Section 2 is simply

$$uni(H,l)x = 1/live(H),$$

for any $x \in \{0\ldots live(H)-1\}$. As in the example above, the previous history and the value of the low variable are disregarded. Note that the history contains sufficient information to calculate the individual lifetimes of threads, or the creation hierarchy of the threads in order that these may be used to influence – either deterministically or stochastically – the policy of the scheduler.

4.2. Probabilistic Noninterference with Schedulers

Given a scheduler σ, let us define the scheduler-specific partial probabilistic low-bisimulation. This bisimulation is a kind of lock-step execution as we have already seen for $\sim_L$. For two bisimilar configurations, if one makes a transition to an equivalence class then the other configuration should also be able to make a transition with the same probability to the same class. However, the history in the second configuration can vary from the one in the first configuration. Let us capture the degree of how the history can vary by defining a relation $=_\sigma$ on σ-feasible histories. It relates σ-feasible histories which are indistinguishable for σ. Define $=_\sigma \in Rel(Hist_\sigma)$ to be the largest relation satisfying

$$\frac{H =_\sigma H'}{\forall l.\,\sigma(H,l) = \sigma(H',l)} \qquad \frac{H =_\sigma H'}{H(i,m) =_\sigma H'(i,m)}$$

For example, $=_{uni}$ relates any two histories such that the number of live threads is the same.

Using the method of Section 3 define the scheduler-specific partial probabilistic bisimulation on configurations who now include histories. We have now everything we need to define a scheduler-specific partial probabilistic low-bisimulation on thread pools.

Definition 5 *Given a scheduler σ, a per $R \in Rel(\vec{Com})$ is a σ-*specific partial probabilistic low-bisimulation *iff $(=_\sigma) \times R \times (=_L)$ is a σ-specific partial probabilistic bisimulation on configurations.*

Write $\vec{C} \sim_L^\sigma \vec{D}$ ($\vec{C}$ and $\vec{D}$ are *low-bisimilar under σ*) iff there exists a partial probabilistic low-bisimulation relating programs $\vec{C}$ and $\vec{D}$.

Note that for the uniform scheduler uni, we have $\sim_L^{uni} = \sim_L$, where $\sim_L$ is the low-bisimulation from Section 3. Given a scheduler σ and a program $\vec{C}$,

$$\vec{C} \text{ is } \sigma\text{-}secure \iff \vec{C} \sim_L^\sigma \vec{C}.$$

4.3. Scheduler-independent Noninterference

As we have argued, a realistic security condition should be scheduler-independent. We are now ready to define the *scheduler-independent* low-bisimulation $\approx_L$ and the scheduler-independent security (SI-security). Define

$$\vec{C} \approx_L \vec{D} \iff \forall \sigma. \vec{C} \sim_L^\sigma \vec{D}, \text{ and}$$

$$\vec{C} \text{ is } SI\text{-}secure \iff \vec{C} \approx_L \vec{C}.$$

Trivially, $\approx_L \subseteq \sim_L$. The example in the beginning of this section shows that the reverse inclusion does not hold. As a corollary, we have $\vec{C}$ is SI-secure implies $\vec{C}$ is secure.

On a side note, the quantification over schedulers is somewhat reminiscent of the "strategies" in the adversary model used by Syverson and Gray [18], which are also functions from computation histories to distributions. Note however that we are not using them to model attacker behaviour (e.g., I/O) – but rather an internal component of the system.

We define yet another low-bisimulation along with a security condition as an aid to prove various analyses secure. This condition is potentially the strongest of all considered in the paper; and proofs of the soundness of particular analyses go through easiest just with this one. Call the new low-bisimulation *strong*.

In order to achieve a compositional bisimulation we want to restrict the strong low-bisimulation so that any two strongly low-bisimilar thread pools must be of equal size and must create/kill exactly the same number of processes at each step under any scheduler. This means that we do not need the machinery for explicit scheduling. In fact, the new low-bisimulation is purely nondeterministic.

Definition 6 *Define the* strong *low-bisimulation $\cong_L$ to be the union of all symmetric relations R on thread pools of equal size, such that whenever $\langle C_0 \dots C_{n-1}\rangle$ R $\langle D_0 \dots D_{n-1}\rangle$ then*

$$\forall s_1 =_L s_2 \forall i. \langle\!| C_i, s_1 |\!\rangle \twoheadrightarrow \langle\!| \vec{C'}, s'_1 |\!\rangle \implies \\ \exists \vec{D'}, s'_2. \langle\!| D_i, s_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{D'}, s'_2 |\!\rangle, \vec{C'} \; R \; \vec{D'}, s'_1 =_L s'_2.$$

The strong security specification goes:

$$\vec{C} \text{ is } strongly\ secure \iff \vec{C} \cong_L \vec{C}.$$

We are going to use this specification of security in the proofs of analyses correctness. Let us first show that the strong low-bisimulation is indeed stronger than the scheduler-independent one.

Proposition 3 *$\vec{C}$ is strongly secure $\implies$ $\vec{C}$ is SI-secure.*

Proof. In order to show $\cong_L \subseteq \approx_L$, let us check that $\cong_L$ is a σ-specific partial probabilistic low-bisimulation for all σ. Assuming $\vec{C} \cong_L \vec{D}$, we need to show that for any σ:

$$\forall H_1 =_\sigma H_2, s_1 =_L s_2. \langle\!| H_1, \vec{C}, s_1 |\!\rangle \to \langle\!| H'_1, \vec{C'}, s'_1 |\!\rangle \implies \\ \exists H'_2, \vec{D'}, s'_2.\ \langle\!| H_2, \vec{D}, s_2 |\!\rangle \to \langle\!| H'_2, \vec{D'}, s'_2 |\!\rangle \\ (i) \sum \{\!| p \,|\, \langle\!| H_1, \vec{C}, s_1 |\!\rangle \to_p \langle\!| H, \vec{S}, s |\!\rangle, \vec{S} \in [\vec{C'}]_{\cong_L}, \\ H =_\sigma H'_1, s =_L s'_1 |\!\} = \\ \sum \{\!| p \,|\, \langle\!| H_2, \vec{D}, s_2 |\!\rangle \to_p \langle\!| H, \vec{S}, s |\!\rangle, \vec{S} \in [\vec{D'}]_{\cong_L}, \\ H =_\sigma H'_2, s =_L s'_2 |\!\}, \\ (ii)\ H'_1 =_\sigma H'_2, \vec{C'} \cong_L \vec{D'}, s'_1 =_L s'_2.$$

Given arbitrary $H_1 =_\sigma H_2$ and $s_1 =_L s_2$, suppose $\vec{C} = \langle C_0 \dots C_{n-1}\rangle$ and $\langle\!| H_1, \vec{C}, s_1 |\!\rangle \to \langle\!| H'_1, \vec{C'}, s'_1 |\!\rangle$. Then there exists i such that the probabilistic transition was triggered by a deterministic transition within C_i using the rule

$$\frac{\langle\!| C_i, s_1 |\!\rangle \twoheadrightarrow \langle\!| \vec{I}, s'_1 |\!\rangle}{\langle\!| H_1, \vec{C}, s_1 |\!\rangle \to_p \langle\!| H'_1, \langle C_0 \dots \vec{I} \dots C_{n-1}\rangle, s'_1 |\!\rangle}$$

where $H'_1 = H_1(i, n + |\vec{I}| - 1)$, $p = \sigma(H_1, l)\, i$ and $s_1 = (h_1, l)$ for some h_1 and $p > 0$. From the definition of $\cong_L$ deduce that $\vec{D}$ has the form $\langle D_0 \dots D_{n-1}\rangle$ such that $\exists \vec{J}, s'_2. \langle\!| D_i, s_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{J}, s'_2 |\!\rangle$ and $\vec{I} \cong_L \vec{J}, s'_1 =_L s'_2$. Hence,

$$\frac{\langle\!| D_i, s_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{J}, s'_2 |\!\rangle}{\langle\!| H_2, \vec{D}, s_2 |\!\rangle \to_q \langle\!| H'_2, \langle D_0 \dots \vec{J} \dots D_{n-1}\rangle, s'_2 |\!\rangle}$$

where $H'_2 = H_2(i, n + |\vec{J}| - 1)$, $q = \sigma(H_2, l)\, i$ and $s_2 = (h_2, l)$ for some h_2 and $q > 0$. Observe that $H_1 =_\sigma H_2$ immediately yields $p = q$. Note that $|\vec{I}| = |\vec{J}|$ implies $H'_1 = H'_2$ which, in turn, gives $H'_1 =_\sigma H'_2$. Since $C_i \cong_L D_i$ for all $i = 0, \ldots, n-1$ and $\vec{I} \cong_L \vec{J}$, looking ahead in Section 5, apply the secure congruence theorem (Theorem 1) to obtain $\vec{C'} = \langle C_0 \ldots \vec{I} \ldots C_{n-1} \rangle \cong_L \langle D_0 \ldots \vec{J} \ldots D_{n-1} \rangle = \vec{D'}$. This completes the proof of the condition (ii). The sum equality of (i) holds since there is a one-to-one correspondence between the two multisets that are summed over. $\Box$

We suspect that the reverse inclusion may also hold in case the SI-security condition is required to be compositional. Notice for example that the quantification over all schedulers in the SI-security condition forces the restriction on the number of threads of two bisimilar programs, i.e., if $\vec{C} \approx_L \vec{D}$ then $|\vec{C}| = |\vec{D}|$. Indeed, otherwise there is a scheduler that can use the difference in the number of threads to modify the low variable which can be used to leak information. For example, it holds that $\mathsf{skip} \mid \mathsf{skip} \approx_L \mathsf{skip}; \mathsf{skip}$ – for any scheduler the two programs execute deterministically in lock-step. However, the parallel composition of the each of the two with the thread $l := 5$ will not preserve the bisimulation $l := 5 \mid \mathsf{skip} \mid \mathsf{skip} \not\approx_L l := 5 \mid (\mathsf{skip}; \mathsf{skip})$ since the probability of l to become 5 depends on the number of threads and many schedulers will reflect it in the semantics. This applies, for example, to the uniform scheduler: $l := 5 \mid \mathsf{skip} \mid \mathsf{skip} \not\sim_L l := 5 \mid (\mathsf{skip}; \mathsf{skip})$.

5. Hook-up Properties

In this section, we investigate the hook-up properties of strong security. Strong bisimulation is constructed to be suitable for compositional reasoning and yet to be general enough to be able to prove security of known analyses. The type-based program analyses that we have considered all satisfy the strongest notion of security. For the rest of the paper "secure" will mean strongly secure.

In the standard security terminology, the *hook-up* property [14] says that if two programs are secure then their composition (parallel, sequential or some other kind) is secure is well. We will show what kinds of composition are allowed. These properties are important since they are the key to the utility of the specification for the purpose of proving the correctness of syntax-directed program analyses.

The low-bisimulation is not a congruence. (Take, for example, any ground context that is insecure.) The low-bisimulation is preserved by *secure contexts*, i.e., contexts built with secure components. Let $[\vec{\bullet}]$ be a hole for a command vector and $[\bullet]$ be a hole for a singleton command. A context $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is secure iff it has one of these forms:

$$\begin{aligned}\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] ::= {} & \mathsf{skip} \mid h := Exp \mid l := Exp \ (Exp \text{ is low}) \\ & \mid [\bullet_1]; [\bullet_2] \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low}) \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low}) \\ & \mid \mathsf{fork}([\bullet_1][\vec{\bullet_2}]) \mid \langle [\vec{\bullet_1}][\vec{\bullet_2}] \rangle\end{aligned}$$

where a (boolean or arithmetic) expression Exp is defined to be *low* iff $\forall s_1 =_L s_2 . \exists n. \langle Exp, s_1 \rangle \downarrow n \ \& \ \langle Exp, s_2 \rangle \downarrow n$. Otherwise, the expression is *high*.

Theorem 1 (Secure congruence) *If $\vec{C_1} \cong_L \vec{C'_1}$, $\vec{C_2} \cong_L \vec{C'_2}$ and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then it is the case that $\mathbb{C}[\vec{C_1}, \vec{C_2}] \cong_L \mathbb{C}[\vec{C'_1}, \vec{C'_2}]$. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] =$ $\mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[\vec{C_1}, \vec{C_2}] \cong_L \mathbb{C}[\vec{C'_1}, \vec{C'_2}]$ provided $\vec{C_1} \cong_L \vec{C_2}$.*

Proof. Cases on $\mathbb{C}$. Consider first the single-threaded case $\mathbb{C}[\vec{C_1}, \vec{C_2}] = C$ and $\mathbb{C}[\vec{C'_1}, \vec{C'_2}] = C'$. Let us start off with the cases $\mathsf{skip}, x := e, \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ $(B : low)$, $\mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ $(B : high)$ and $\mathsf{fork}([\bullet_1][\vec{\bullet_2}])$.

In the proof below, let us make use of the so-called "up-to" technique for proving two programs $\vec{I}$ and $\vec{J}$ bisimilar. In such a technique, one constructs a relation $R \in Rel(\vec{Com})$ such that $\vec{I}\ R\ \vec{J}$ and $R \subseteq F_{\cong_L}(R \cup \cong_L)$ where $F_{\cong_L}(\cdot)$ is the corresponding function on relations such that R is a strong low-bisimulation whenever R is a subset of $F(R)$ (c.f. Appendix A).

The standard nondeterministic bisimulation theory guarantees us then that $R \subseteq \cong_L$ and thereby $\vec{I} \cong_L \vec{J}$. It is indeed enough to have the singleton relation $R = \{(C, C')\}$ for a proof of the first cases in question. Let us inspect these cases of $\mathbb{C}$:

skip
: Start with s_1 and s_2 $(s_1 =_L s_2)$. With either state, the computation of skip terminates in one step with the same states. So, the final states are low-equal as well.

$l := Exp$ **(Exp is low)**
: Since Exp is a low expression, the computation of the command with different initial states s_1 and s_2 $(s_1 =_L s_2)$ terminates in one step with the same states that are still low-equal.

$h := Exp$
: Exp can be arbitrary. Computing the command will not change the low component of the states.

if B **then** $[\bullet_1]$ **else** $[\bullet_2]$ $(B : low)$
: Start with $\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s_1 \rangle$ and $\langle \mathsf{if}\ B\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2, s_2 \rangle$ $(s_1 =_L s_2)$. B has the same value in either state. Thus, one step of computation will lead to the configuration with bisimilar commands (C_1 and C'_1, or C_2 and C'_2) and low-equal states. Further, we know $C_1 \cong_L C'_1$ and $C_2 \cong_L C'_2$.

if B **then** $[\bullet_1]$ **else** $[\bullet_2]$ $(B : high)$
: Start with s_1 and s_2 $(s_1 =_L s_2)$. Here the value of B might differ for s_1 and s_2, but since $C_i \cong_L C'_j$ for $i, j = 1, 2$ one step of computation ends up in bisimilar commands and and low-equal states in this case as well.

$\mathsf{fork}([\bullet_1][\vec{\bullet_2}])$

Trivial, as computation does not depend on h.

We have three cases left:

$[\bullet_1]; [\bullet_2]$

C_1 (and C_1') might spawn new processes, so we need to have all possible bisimilar postfix vectors after C_2 (C_2') in R. $R = \{((E; C_2)\vec{G}, (E'; C_2')\vec{G'}) | E \cong_L E', \vec{G} \cong_L \vec{G'}\}$. Then $R \subseteq F_{\cong_L}(R \cup \cong_L)$.

$\mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]$ (B is low)

Choose R to be the relation $\{((E; \mathsf{while}\ B\ \mathsf{do}\ C_1)\vec{G}, (E'; \mathsf{while}\ B\ \mathsf{do}\ C_1')\vec{G'}) | E \cong_L E', \vec{G} \cong_L \vec{G'}\}$. Start with s_1 and s_2 ($s_1 =_L s_2$). B has the same value in either state. Thus, one step of computation will lead both configurations either to the termination of the loop (with low-equal states) or to configurations with commands $C_1; \mathsf{while}\ B\ \mathsf{do}\ C_1$ and $C_1'; \mathsf{while}\ B\ \mathsf{do}\ C_1'$ and low-equal states. C_1 and C_1' might also spawn new processes.

$\langle[\vec{\bullet_1}][\vec{\bullet_2}]\rangle$

Finally, the case of multiple threads. Take $R = \{(\vec{E_1}\vec{E_2}, \vec{G_1}\vec{G_2}) | \vec{E_i} \cong_L \vec{G_i}, i = 1, 2\}$. Suppose $\vec{E_i} = \langle E_i^0 \ldots E_i^{n_i - 1}\rangle$ and $\vec{G_i}' = \langle G_i^0 \ldots G_i^{m_i - 1}\rangle$ for $i = 1, 2$. Let us prove $R \subseteq F_{\cong_L}(\cong_L)$ by showing:

$$(i) |\vec{E_1}\vec{E_2}| = |\vec{G_1}\vec{G_2}|$$

$$(ii)\ \forall s_1 =_L s_2 \forall i, j. \langle\!|E_i^j, s_1|\!\rangle \twoheadrightarrow \langle\!|\vec{E}, s_1'|\!\rangle \implies \exists \vec{G}, s_2'. \langle\!|G_i^j, s_2|\!\rangle \twoheadrightarrow \langle\!|\vec{G}, s_2'|\!\rangle, \vec{E} \cong_L \vec{G}, s_1' =_L s_2'.$$

The (i)-item is straightforward, as $|\vec{E_i}| = |\vec{G_i}|$ for $i = 1, 2$. The (ii)-item follows from the two separate (ii)-items (when i is fixed to be either 1 or 2) which come from the unwinding of the definition of the strong bisimulation for each of the $\vec{E_i} \cong_L \vec{G_i}$. □

An immediate corollary is the hook-up property result.

Corollary 1 (Hook-up) *If $\vec{C_1}, \vec{C_2}$ are secure and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure provided $\vec{C_1} \cong_L \vec{C_2}$.*

Proof. Observe that $\vec{C_1}, \vec{C_2}$ are secure implies $\vec{C_i} \cong_L \vec{C_i}$ for $i = 1, 2$. Thus by the security congruence theorem $\mathbb{C}[\vec{C_1}, \vec{C_2}] \cong_L \mathbb{C}[\vec{C_1}, \vec{C_2}]$, i.e., $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure. □

6. Proving Analyses Secure

The compositionality of the security condition can be fruitfully exploited when proving the correctness of various compositional program analyses. In this section we develop one such analysis.

A popular form for presenting program analysis is as a nonstandard type system – as exemplified in the present context by Volpano, Smith and Irvine's type system [20] expressing Denning's flow analysis [6]. The type system was extended to a statically-threaded language with a uniform scheduler in [19]. The style of such type systems is to give security types to expressions (*high* or *low*) and commands (*high cmd* or *low cmd*) and make sure that these types do not mismatch when composing the commands.

The hook-up properties of Corollary 1 are sufficient to make the correctness proof of Volpano and Smith's analysis extremely simple – and the proof thereby includes correctness with respect to, e.g., round-robin schedulers and schedulers using dynamic priorities (via dependence on low "priority" variables in the program).

As an example we will consider a useful improvement of Volpano and Smith's system (for our slightly richer language) based on ideas from a recent analysis developed by Johan Agat [1] which shows how timing channels can be eliminated from sequential programs by a combination of typing and code transformation.

The improvement in question concerns the treatment of the security of conditional expressions in the case when the boolean depends on high variables. Consider a command of the form $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ where $B : \mathit{high}$. Volpano and Smith's analysis requires that (i) no low assignments are made within C_1 or C_2, (ii) no while loops occur within the branches, and (iii) the whole conditional is executed atomically (i.e., without interruption by another thread.) The last of these points is enforced in the Volpano-Smith system by an additional language construct called protect. A disadvantage of using protect is that it is not a typical synchronisation primitive of modern concurrent languages.[3]

We will instead use a technique due to Agat to avoid the need for protect statements, and obtain an analysis that is more permissive than Volpano and Smith's analysis.

6.1. Agat's Approach

As Agat observed in the sequential case, each of the restrictions on a high conditional can be lifted to some degree by observing that it is sufficient for C_1 and C_2 to be low-bisimilar. The hook-up result shows that this is true in the present setting also.

Agat's approach is to *transform* the branches of the conditional so that they become bisimilar in a manner that is easily checked by a syntax directed set of rules. The essence

[3]The protect statement also appears prohibitively expensive to implement using, e.g., locks, since in order to block all threads during a protect, *every* atomic command C in the program must be transformed into getlock; C; releaselock.

of the transformation is to pad the branches of the conditionals. In our case this padding will add skip instructions and possibly dummy forks. Agat's aim was the elimination of *external* timing attacks; we can also argue this here, although we simply focus on the internal timing leaks.

In the rest of the section we present the combined analysis and transformation system, prove its correctness with respect to strong security and show how this approach eliminates the need for protect's.

6.2. The Type System

The analysis is based on a type system that transforms a given program into a new program. If the initial program is free of direct and indirect insecure flows then it might be accepted by the system and transformed into a program free of timing leaks. Otherwise the initial program is rejected. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$, where $\vec{C}$ is a program, C' is the result of its transformation and $\vec{Sl}$ is the "type" of $\vec{C'}$. The types of programs are their *low slices*. A low slice is essentially a copy of a secure program in which assignments to high variables have been replaced by skip's. The slice Sl does not have any occurrences of h, but has the same structure as C' and models the timing behaviour of C', as observable by other threads.

The typing and transformation rules are presented in Figure 4. The variables h and l have the types $high$ and low respectively. Integer literals n may be considered as either $high$ or low. An arbitrary expression Exp may be considered as $high$. In all but If$_{high}$ rules, the transformed program is constructed compositionally using the same constructs as the original program. The information about the low slice of the new program is recorded in the typing. Command skip is its own low slice and therefore its own type. The rule Assign$_{low}$ prevents direct insecure information flows – the assignment $l := h$ is not typable. The rule Assign$_{high}$ types an assignment to the high variable with the low slice skip. The rules Arithm$_{low}$, Seq, If$_{low}$, Par and Fork propagate types compositionally. The guard of the while-loop in the rule While has to be low in order to prevent the timing (and nontermination) flow from the loop's guard.

The rule If$_{high}$ prevents indirect insecure flows and timing flows. Let $al(C)$ be a predicate returning True whenever there is a syntactic occurrence of an assignment to low variable l in the command C and returning False otherwise. The condition $al(Sl_1) = al(Sl_2) =$ False prevents the indirect leaks.

The interesting rule is the case when the guard of an if command is high. In this case both branches must be typable (i.e., they must have a low slice). For the transformed program to be secure it is also necessary that the two branches be low-bisimilar. This is achieved by cross-copying the low slice of one branch into the other. The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

6.3. An Example Code Transformation

Let us give a simple example of the application of the transformation system. The example is based on one in [1], and is used to illustrate the effect of the transformation. The heart of RSA encryption is the computation of $a^b \ mod \ n$, where a is an integer representing the plaintext and b is the integer encryption key. To efficiently compute $a^b \ mod \ n$, the modular exponentiation algorithm can be used, but, as shown by Kocher [12], a careless implementation will leak b through timing. On the left in Figure 5, we give an example of such an implementation. (Assume we have an if without an else with the obvious semantics.) All variables except for i and k in the program are high. The result of computation is stored in d.

We represent b as a $k+1$ element long array of secret booleans, with the most significant bit of b at $b[k]$. The initial program is not secure since the two branches of the if have different timing behaviours (they are not low-bisimilar). For some schedulers, a (secure) thread running in parallel with this code may compute different low outputs depending on the values of the b_i (we omit a concrete example for reasons of space). The transformation algorithm presented in Figure 4 (with the minor extension necessary to handle arrays with a low index) will close these timing leaks by transforming the program into the one given in the middle of Figure 5. The security of this padded program relies on atomic execution of the assignment operators. Finally, the low slice of the padded program is shown on the right in Figure 5.

6.4. Correctness of the Analysis

Let us prove the security of Agat's analysis extended to the multi-threaded language. The following theorem is a straightforward use of the secure congruence properties of the security condition proved in Theorem 1.

Theorem 2 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \approxeq_L \vec{Sl}$

Proof. Proceed by structural induction on the thread pool $\vec{C'}$. Let us consider the cases by which $\vec{C'} : \vec{Sl}$. In the cases skip : skip, $l := Exp : l := Exp$ $(Exp : low)$ and $h := Exp$: skip the program $\vec{C'}$ is low-bisimilar to it's low slice. The cases $C_1; C_2$, while B do C_1 : while B do Sl' $(B : low)$, $\langle C_0 \ldots C_{n-1}\rangle : \langle Sl_0 \ldots Sl_{n-1}\rangle$, fork$(C_1\vec{C_2})$ and if B then C_1 else C_2 : if B then Sl_1 else Sl_2 $(B : low)$ provide secure contexts $[\bullet_1]; [\bullet_2]$, while B do $[\bullet]$, $\langle[\bullet_0] \ldots [\bullet_{n-1}]\rangle$, fork$([\bullet_1][\vec{\bullet_2}])$ and if B then $[\bullet_1]$ else $[\bullet_2]$ $(B : low)$ respectively. By the the

[Var]	$h : high \qquad l : low$
[Exp]	$n : \tau \qquad Exp : high$
[Arithm$_{low}$]	$\dfrac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low}$
[Skip]	$\text{skip} \hookrightarrow \text{skip} : \text{skip}$
[Assign$_{low}$]	$\dfrac{Exp : low}{l := Exp \hookrightarrow l := Exp : l := Exp}$
[Assign$_{high}$]	$h := Exp \hookrightarrow h := Exp : \text{skip}$
[Seq]	$\dfrac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : Sl_1; Sl_2}$
[While]	$\dfrac{B : low \quad C \hookrightarrow C' : Sl}{\text{while } B \text{ do } C \hookrightarrow \text{while } B \text{ do } C' : \text{while } B \text{ do } Sl}$
[Par]	$\dfrac{C_0 \hookrightarrow C_0' : Sl_0 \ \ldots \ C_{n-1} \hookrightarrow C_{n-1}' : Sl_{n-1}}{\langle C_0 \ldots C_{n-1} \rangle \hookrightarrow \langle C_0' \ldots C_{n-1}' \rangle : \langle Sl_0 \ldots Sl_{n-1} \rangle}$
[Fork]	$\dfrac{C_1 \hookrightarrow C_1' : Sl_1 \quad \vec{C_2} \hookrightarrow \vec{C_2'} : \vec{Sl_2}}{\text{fork}(C_1 \vec{C_2}) \hookrightarrow \text{fork}(C_1' \vec{C_2'}) : \text{fork}(Sl_1 \vec{Sl_2})}$
[If$_{low}$]	$\dfrac{B : low \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\text{if } B \text{ then } C_1 \text{ else } C_2 \hookrightarrow \text{if } B \text{ then } C_1' \text{ else } C_2' : \text{if } B \text{ then } Sl_1 \text{ else } Sl_2}$
[If$_{high}$]	$\dfrac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \text{False}}{\text{if } B \text{ then } C_1 \text{ else } C_2 \hookrightarrow \text{if } B \text{ then } C_1'; Sl_2 \text{ else } Sl_1; C_2' : \text{skip}; Sl_1; Sl_2}$

Figure 4. Transformation to eliminate timing leaks

Original program $(i, k : low)$

```
$c := 0;\ d := 1;\ i := k;$
while $i \geq 0$
   do $i := i - 1;$
      $c := 2 \cdot c;$
      $d := (d \cdot d)\ mod\ n;$
      if $b_i = 1$
         then $c := c + 1;$
              $d := (d \cdot a)\ mod\ n$
```

$\hookrightarrow$

Transformed program $(i, k : low)$

```
$c := 0;\ d := 1;\ i := k;$
while $i \geq 0$
   do $i := i - 1;$
      $c := 2 \cdot c;$
      $d := (d \cdot d)\ mod\ n;$
      if $b_i = 1$
         then $c := c + 1;$
              $d := (d \cdot a)\ mod\ n$
         else skip;
              skip
```

:

Low slice

```
skip; skip; $i := k;$
while $i \geq 0$
   do $i := i - 1;$
      skip;
      skip;
      skip;
      skip;
      skip
```

Figure 5. Producing a secure modular exponentiation thread

induction hypothesis the sub-commands have low-bisimilar low slices, so the conclusion follows by the hook-up property.

The last case is if B then $C_1'; Sl_2$ else $Sl_1; C_2'$: skip; $Sl_1; Sl_2$. In order to apply the special if-on-high case of the hook-up property we need to show $C_1'; Sl_2 \approxeq_L Sl_1; C_2'$. By the induction hypothesis, $C_i' \approxeq_L Sl_i$ $(i = 1, 2)$. Now since $[\bullet_1]; [\bullet_2]$ is a secure context, we have that $C_1'; Sl_2 \approxeq_L Sl_1; C_2'$. Finally, this is sufficient to argue that $\{(\text{if } B \text{ then } C_1'; Sl_2 \text{ else } Sl_1; C_2', \text{skip}; Sl_1; Sl_2)\}$ is a strong bisimulation up to $\approxeq_L$ (which requires just a case analysis on the outcome of B). □

Corollary 2 (Security of the Analysis) *$\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'}$ is secure.*

Proof. Theorem 2 gives $\vec{C'} \approxeq_L \vec{Sl}$. The symmetry and transitivity of $\approxeq_L$ entails $\vec{C'} \approxeq_L \vec{C'}$, i.e., $\vec{C'}$ is secure. □

6.5. Soundness of the Transformation

We have shown that the result of the transformation is secure, but what of its relation to the original program? We argue below that the transformed program is a possibilistic refinement of the original program, and that under certain additional assumptions, that it is equivalent.

Clearly, the padding introduced by the transformation can change the timing and thereby the program's probabilistic behaviour, but otherwise it is just additional "stuttering". A more serious problem is that it might introduce extra nontermination, in the case that a nonterminating loop is cross-copied into a branch that would otherwise have terminated.

To make the first point precise, let us define a *possibilistic (bi)simulation* on programs. To obtain a possibilistic transition relation we can simply erase the probability from the uniform scheduler semantics defined in Section 2.

Definition 7 *Define the possibilistic simulation $\preceq$ (resp., bisimulation $\simeq$) to be the the union of all (resp. symmetric) relations R on thread pools such that whenever $\vec{C}\ R\ \vec{D}$ then for all s, s', C', there exists a D' such that*

$$\langle \vec{C}, s \rangle \to \langle \vec{C'}, s' \rangle \Longrightarrow \langle \vec{D}, s \rangle \to^* \langle \vec{D'}, s' \rangle \textit{ and } \vec{C'}\ R\ \vec{D'}.$$

Intuitively, $\vec{C} \preceq \vec{D}$ holds whenever $\vec{C}$ performs a (maybe infinitely) slowed down version of the computation of $\vec{D}$.

Proposition 4 *$\preceq$ is reflexive, transitive and preserved by all contexts (not necessarily secure), i.e., $\preceq$ is a precongruence.*

Proposition 5 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \preceq \vec{C}$.

Proof. Induction on the height of the transformation derivation. The cases Skip, Assign$_{low}$, Assign$_{high}$ follow from the reflexivity of $\preceq$. The cases Seq, While, Par, Fork and If$_{low}$ are implied by the induction hypotheses and the precongruence of $\preceq$. The remaining case is If$_{high}$, i.e., if B then $C_1'; Sl_2$ else $Sl_1; C_2'$: skip; $Sl_1; Sl_2$. By induction we have $C_1' \preceq C_1$ and $C_2' \preceq C_2$. The side condition guarantees that there are no assignments in the slices Sl_1 and Sl_2 of the branches C_1 and C_2 of the if. Thus Sl_1 and Sl_2 contain only dummy computation without changing the state. The insertion of such a computation might only slow down the computation. Thus, $C_1'; Sl_2 \preceq C_1$ and $Sl_1; C_2' \preceq C_2$. The precongruence of $\preceq$ yields $C' \preceq C$. □

6.6. Elimination of the protect

Let us define the notion of a protected program from [19]. A program $\vec{C}$ is *protected* iff any conditional on the high variable is wrapped in a protect term. A protect cannot have other protect's or while's within its scope of protection. We are now ready to sketch the protect elimination result.

Theorem 3 (Protect elimination) *If $\vec{C_p}$ is well-typed and protected in Volpano-Smith's type system then $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$ where $\vec{C}$ is the result of erasing the* protect*'s from $\vec{C_p}$, for some secure $\vec{C'}$ and some $\vec{Sl}$ such that $\vec{C} \simeq \vec{C'}$.*

Proof. (Sketch) Although there are only two types of commands in Volpano-Smith's system (*low cmd* and *high cmd*), the type inference is still performed compositionally and similarly to the transformation in Figure 4. Hence we are able to type all programs that are typable in Volpano-Smith's system. Whenever $\vec{C_p}$ is well-typed then $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$. By Proposition 5 we immediately have $\vec{C'} \preceq \vec{C}$.

The program $\vec{C}$ has no occurrences of fork (no dynamic thread creation in Volpano-Smith's language). Therefore, the padding transformation cannot introduce new fork's. While-loops cannot be introduced since the only place where the transformation generates new code is the if-on-high case, and since such terms are protected they contain no occurrences of while. What the transformation might introduce in the branches of a high if is a sequential composition of low if's and skip's (these are the only possibilities for the low slices of the branches). The insertion of these commands does not affect the possibilistic semantics or the termination properties of the initial program (we omit the routine proofs of these laws). Using this intuition, the rest of the proof of $\vec{C'} \simeq \vec{C}$ follows similarly to Proposition 5. □

7. Conclusions

We have developed extensional semantics-based specifications of secure information flow for multi-threaded programs. The specifications capture probabilistic covert channels that arise from the scheduling of concurrent threads. We have argued that under the worst case assumption the security condition should be scheduler-independent.

By defining security specifications as noninterference based on a probabilistic bisimulation, we have achieved compositional reasoning (build secure programs from secure components) and precision over approaches proposed in the literature, without the mathematical overheads of a denotational approach using, e.g., probabilistic powerdomains. Proving type-system based analyses is simple since such analyses are usually compositional as well.

In this paper, we have been considering unsynchronised multi-threaded concurrency with dynamic thread creation. Future work must include the study of synchronisation primitives.

Another simplification is in our treatment of I/O. Implicitly, we permit the attacker (the external supplier of the program) to observer intermediate low outputs, and even provide the initial low input, but not to selectively provide inputs to the program as it runs. Blocking input gives the attacker(s) the ability to externally control the timing behaviour of threads. To study the essence of these problems, it might be appropriate to extend Focardi and Gorrieri's process-calculus-based study [8]) to the probabilistic case. This should also enable a more rigorous connection to be made to Gray's P-Restrictiveness.

References

[1] J. Agat. Transforming out timing leaks. In *POPL'00: The 27:th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53. ACM, January 2000.

[2] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[3] C. Baier and M. Z. Kwiatkowska. Domain equations for probabilistic processes. *Mathematical Structures in Computer Science*, June 1999. To appear.

[4] J.-P. Banatre, C. Bryce, and D. Le Metayer. An approach to information security in distributed systems. In *Proceedings of the 5th IEEE International Workshop on Future Trends in Distributed Computing Systems*, pages 384–394, 1995.

[5] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[6] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

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

[8] R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security*, 3(1):5–33, 1994/1995.

[9] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, April 1982.

[10] J. Gray III. Probabilistic interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990.

[11] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 1998.

[12] P. C. Kocher. Timing attacks on implementations of diffiehellman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

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

[14] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. 1987.

[15] J. McLean. The specification and modeling of computer security. *Computer*, 23(1), January 1990.

[16] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming, ESOP'99*, LNCS 1576, pages 40–58, Amsterdam, Springer-Verlag, March 1999.

[17] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 January 1998.

[18] P. Syverson and J. Gray III. The epistemic representation of information flow security in probabilistic systems. In *Proc. 8th IEEE Computer Security Foundations Workshop*, 1995.

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

[20] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

A. Observations on Partial Probabilistic Bisimulation

This appendix presents some observations on properties of partial probabilistic bisimulations. The union of all partial probabilistic bisimulations can itself be shown to be a partial probabilistic bisimulation. This follows from a standard Tarski fixed-point argument (although to our knowledge this argument has not been used elsewhere for probabilistic bisimulation). The key is to note that pers on a given set form a complete lattice (with meet given by set-intersection, and join given by the transitive closure of the set-union), and that the definition can be phrased in terms of a continuous functional.

Recall that an unlabelled probabilistic transition system is a set of states $\mathbb{S}$ and a set of transitions of the form $\mathbb{s} \to \mu$ where $\mathbb{s} \in \mathbb{S}$ and $\mu \in \mathcal{D}(\mathbb{S})$. Let us restrict our attention to probabilistic transition systems such that $\mathbb{s} \to \mu$ and $\mathbb{s} \to \mu'$ implies $\mu = \mu'$ (which is clearly the case for the transition system that corresponds to the probabilistic semantics of the multi-threaded language).

Fix such a probabilistic transition system. Let F denote the function from pers to pers (over $\mathbb{S}$) given by:

$$\mathbb{s}\ F(R)\ \mathbb{s}' \iff \begin{array}{l} (i)\ \forall S \in \mathbb{S}/R.\, \mathbb{s} \to_p S \Longrightarrow \mathbb{s}' \to_p S \\ (ii)\ \mathbb{s} \to_0 \mathbb{S} \setminus \mathit{dom}(R) \end{array}$$

It is straightforward to check that $F(R)$ is indeed transitive and symmetric. The following two propositions state the monotonicity and co-continuity of F.

Proposition 6 *Suppose $P, Q \in \mathit{Rel}(\mathbb{S})$ are pers. Then $P \subseteq Q \Longrightarrow F(P) \subseteq F(Q)$.*

Proof. Given $\mathbb{s}\ F(P)\ \mathbb{s}'$, we need to show $\mathbb{s}\ F(Q)\ \mathbb{s}'$. Suppose $\mathbb{s} \to_q S_Q$ for some $S_Q \in \mathbb{S}/Q$. Since $P \subseteq Q$, we can partition S_Q into $\cup_i S_P^i \cup S$, where the S_P^i are equivalence classes of P, and S is a subset of $\mathbb{S} \setminus \mathit{dom}(P)$. Suppose $\mathbb{s} \to_{p_i} S_P^i$. Note $\mathbb{s} \to_0 S$. By $\mathbb{s}\ F(P)\ \mathbb{s}'$ we have $\mathbb{s}' \to_{p_i} S_P^i$ and $\mathbb{s}' \to_0 S$. This implies $\mathbb{s}' \to_q S_Q$ by simple summing. Conclude $\mathbb{s}\ F(Q)\ \mathbb{s}'$. □

Proposition 7 *For a nonincreasing ω-chain of pers $R_0 \supseteq \ldots \supseteq R_i \supseteq \ldots$, F preserves co-limits, i.e.,*

$$F(\cap_{i<\omega} R_i) = \cap_{i<\omega} F(R_i).$$

Proof. Since $\cap_{i<\omega} R_i \subseteq R_j$ for any j, apply the monotonicity of F to get, for all j, $F(\cap_{i<\omega} R_i) \subseteq F(R_j)$ which implies the $\subseteq$ part of the set equality. For the reverse inclusion, we need to show that if for some $\mathbb{s}$ and $\mathbb{s}'$ we have $\mathbb{s}\ (\cap_{i<\omega} F(R_i))\ \mathbb{s}'$ then $\mathbb{s}\ F(\cap_{i<\omega} R_i)\ \mathbb{s}'$. The assumption that $\mathbb{s}\ (\cap_{i<\omega} F(R_i))\ \mathbb{s}'$ entails $\mathbb{s}\ F(R_i)\ \mathbb{s}'$ for all i, which can be rewritten as

$$\begin{array}{l} (i)\ \forall S \in \mathbb{S}/R_i.\, \mathbb{s} \to_p S \Longrightarrow \mathbb{s}' \to_p S \\ (ii)\ \mathbb{s} \to_0 \mathbb{S} \setminus \mathit{dom}(R_i) \end{array}$$

for all i. Suppose now for some $S \in \mathbb{S}/\cap_{i<\omega} R_i$ we have $\mathbb{s} \to_p S$. There exists a sequence of pers $\{S_i\}_{i<\omega}$ such that $S_i = \mathbb{S}/R_i$ and $S = \cap_{i<\omega} S_i$. Note that $S_0 \supseteq \ldots \supseteq S_i \supseteq \ldots \supseteq S$. Consider the sequence of nonincreasing probabilities $\{p_i\}_{i<\omega}$ such that $\mathbb{s} \to_{p_i} S_i$. The fact that the summing operation over countable sets is continuous guarantees $p = \inf\{p_i\}_{i<\omega}$. Thus, $\mathbb{s}' \to_{p_i} S_i$ for all i, implying $\mathbb{s}' \to_p S$. Analogously, $\forall i.\, \mathbb{s} \to_0 \mathbb{S} \setminus \mathit{dom}(R_i)$ implies $\mathbb{s} \to_0 \mathbb{S} \setminus \mathit{dom}(\cap_{i<\omega} R_i)$ which concludes the proof of $\mathbb{s}\ F(\cap_{i<\omega} R_i)\ \mathbb{s}'$. □

Let us define $\sim\ = \cup\{R \mid R \text{ is a per}, R \subseteq F(R)\}$. We are now in a position to establish the fixed-point result.

Proposition 8 (Fixed point) *The relation $\sim$ is the greatest fixed point of F in the lattice of pers. It can be alternatively represented by $\sim\ = \cap_{i<\omega} F^i(\mathbb{S} \times \mathbb{S})$.*

Proof. The proof is a standard argument, by appeal to the Knaster-Tarski fixed-point theorem (see, e.g., [5]). The alternative representation statement holds due to the co-continuity of F. Observe that since, by induction, $F^i(\mathbb{S} \times \mathbb{S})$ is a per for any i then so is $\sim$ itself. □

When proving a per R to be a partial probabilistic bisimulation (or $R \subseteq\ \sim$) one can use the the following analogue of the standard "up-to" technique.

Proposition 9 *Given a per $R \in \mathit{Rel}(\mathbb{S})$,*

$$R \subseteq F((R \cup \sim)^+) \Longrightarrow R \subseteq\ \sim.$$

Proof. Assume $R \subseteq F((R \cup \sim)^+)$. Note that $\sim\ \subseteq F(\sim) \subseteq F((R \cup \sim)^+)$ due to the fixed-point proposition and F's monotonicity. Thus, $R \cup \sim\ \subseteq F((R \cup \sim)^+)$. Since the relation on the right-hand side is a per, it must be true that $(R \cup \sim)^+ \subseteq F((R \cup \sim)^+)$. By the definition of $\sim$, we have $(R \cup \sim)^+ \subseteq\ \sim$ which, in particular, implies $R \subseteq\ \sim$. □