Confidentiality for Probabilistic Multi-Threaded Programs and Its Verification

Tri Minh Ngo, Mariëlle Stoelinga, and Marieke Huisman

University of Twente, Netherlands
tringominh@gmail.com
Marielle.Stoelinga@ewi.utwente.nl
Marieke.Huisman@ewi.utwente.nl

Abstract. Confidentiality is an important concern in today's information society: electronic payment and personal data should be protected appropriately. This holds in particular for multi-threaded applications, which are generally seen the future of high-performance computing. Multi-threading poses new challenges to data protection, in particular, data races may be exploited in security attacks. Also, the role of the scheduler is seminal in the multi-threaded context.
This paper proposes a new notion of confidentiality for probabilistic and non-probabilistic multi-threaded programs, formalized as scheduler-specific probabilistic observational determinism (SSPOD), together with verification methods. Essentially, SSPOD ensures that no information about the private data can be derived either from the public data, or from the probabilities of the public data being changed. Moreover, SSPOD explicitly depends on a given (class of) schedulers.
Formally, this is expressed by using two conditions: (i) each publicly visible variable individually behaves deterministically with probability 1, and (ii) for every trace considering all publicly visible variables, there always exists a matching trace with equal probability. We verify these conditions by a clever combination of new and existing algorithms over probabilistic Kripke structures.

1 Introduction

Confidentiality plays a crucial role in the development of applications dealing with private data, such as Internet banking, medical information systems, and authentication systems. These systems need to enforce strict protection of private data, like credit card details, medical records, etc. The key idea is that secret information should not be derivable from public data. For example, the program `if` $(\mathtt{h} > 0)$ `then` $\mathtt{l} := 0$ `else` $\mathtt{l} := 1$, where `h` is a private variable and `l` is a public variable[1], is considered insecure, because we can derive the value of `h` from the value of `l`. If private data is not sufficiently protected, users refuse

[1] For simplicity, throughout this paper, we consider a simple two-point security lattice, where the data is divided into two disjoint subsets, of private (high) and public (low) security levels, respectively.

to use such applications. Using formal means to establish confidentiality is a promising way to gain the trust of users.

Possibilistic programs. With the trend of multiple cores on a chip and massively parallel systems like general purpose graphic processing units, multi-threading is becoming more standard. Existing confidentiality properties, such as *noninterference* [12] and *observational determinism* [30, 15] are not suitable to ensure confidentiality for multi-threaded programs. They only consider input-output behavior, and ignore the role of schedulers, while multi-threaded programs allow all interactions between threads and intermediate results to be observed [30, 15, 14]. Thus, new methods have to be developed for an observational model where an attacker can access the full code of the program, observe the traces of public data, and limit the set of possible program traces by selecting a scheduler.

Because of the exchange of intermediate results, to ensure confidentiality for multi-threaded programs, it is necessary to consider the whole execution traces, i.e., the sequences of states that occur during the execution of the program [30, 23]. Besides, due to the interactions between threads, the traces of a multi-threaded program depend on the scheduler that is used to execute the program. Therefore, a program's confidentiality is only guaranteed under a particular scheduler, while a different scheduler might make the program reveal secret information, as illustrated by the following example.

$$\{\texttt{if}\ (\texttt{h} > 0)\ \texttt{then}\ \texttt{l1} := 1\ \texttt{else}\ \texttt{l2} := 1\} \| \{\texttt{l1} := 1; \texttt{l2} := 1\} \| \{\texttt{l2} := 1; \texttt{l1} := 1\},$$

where $\|$ is the parallel operator. Under a nondeterministic scheduler, the secret information cannot be derived, because the traces in the cases $\texttt{h} > 0$ and $\texttt{h} \leq 0$ are the same. However, under a scheduler that always executes the leftmost thread first, the secret information is revealed by observing whether `l1` is updated before `l2`, i.e., when `l1` is updated before `l2`, the attacker knows that $\texttt{h} > 0$. However, this program is considered secure by observational determinism [30, 15].

Taking into account the effect of schedulers on confidentiality, we proposed a definition of *scheduler-specific observational determinism* (SSOD) for *possibilistic* multi-threaded programs [14]. Basically, a program respects SSOD if (SSOD-1) for any initial state, traces of each public variable are stuttering-equivalent, and (SSOD-2) for any two initial states I and I' that are indistinguishable w.r.t. the public variables, for every trace starting in I, there *exists* a trace that is stuttering equivalent w.r.t. *all* public variables, starting in I'.

SSOD is scheduler-specific, since traces model the runs of a program under a particular scheduler. When the scheduling policy changes, some traces *cannot occur*, and also, some new traces *might appear*; thus the new set of traces may not respect our requirements. For example, the above program is accepted by SSOD w.r.t. the nondeterministic scheduler, but is rejected under the scheduler that always executes the leftmost thread first.

Probabilistic programs. To extend our earlier results, this paper also considers programs that have probabilistic behaviors. For probabilistic programs, some

threads might be more likely to be executed than others. This opens up the possibility of probabilistic attacks, as in the following example.

$$\texttt{if}\ (\texttt{h} > 0)\ \texttt{then}\ \{\texttt{l1} := 1 \parallel \texttt{l2} := 1\}\ \texttt{else}\ \{\texttt{l1} := 1 \parallel \texttt{l2} := 1\}.$$

This program is secure under a nondeterministic scheduler. However, consider a scheduler that, when $\texttt{h} > 0$, picks thread `l1:=1` first with probability 3/4; otherwise, it chooses between the threads with equal probabilities. With this scheduler, we can learn information about `h` from the probabilities of public data traces. However, the program is still accepted by SSOD w.r.t. this scheduler, because SSOD only considers the existence of traces, not its probability.

To detect vulnerabilities to probabilistic attacks, we define *scheduler-specific probabilistic-observational determinism* (SSPOD). This formalizes the observational determinism property for probabilistic multi-threaded programs, executed under a probabilistic scheduler. Basically, a program respects SSPOD if (SSPOD-1) for any initial state, each public variable individually behaves deterministically with probability 1, and (SSPOD-2) for any two initial states I and I' that are indistinguishable *w.r.t.* the public variables, for every trace starting in I, there *exists* a trace that is stuttering equivalent *w.r.t. all* public variables, starting in I', and the probabilities of these two matching traces are the same.

The first condition of SSPOD requires that all public variable traces individually evolve deterministically. Requiring only that a stuttering-equivalent public variable trace exists is not sufficient to guarantee confidentiality for multi-threaded programs, as extensively discussed in [30, 14]. The first condition avoids leakage of private information based on the observation of public data traces. The second condition of SSPOD requires the existence of a public data trace with equal probabilities. This existential condition avoids refinement attacks where an attacker chooses an appropriate scheduler to control the set of possible traces. The second condition is also sufficient to ensure that any difference in the relative order of updates is coincidental, and thus no private information can be deduced from it. In addition, SSPOD also guarantees that no private information can be derived from the probabilistic distribution of traces, because indistinguishable traces occur with the same probabilities.

Notice that the question which classes of schedulers appropriately model real-life attacks is orthogonal to our results: our definition is parametric on the scheduler. In Section 5, we compare SSPOD with the existing formalizations of confidentiality properties for probabilistic programs [29, 23, 24], and argue that they are either unsuitable to the multi-threaded context, or very restrictive.

Verification. Besides formalizing the property, the paper also discusses how to verify SSPOD. The traditional way to check information flow properties is by using a type system. However, as discussed in [14], type systems are not suited to verify existential properties, as the one in SSPOD. Besides, type systems that have been proposed to enforce confidentiality for multi-threaded programs are often very restrictive. This restrictiveness makes the application programming become impractical; many intuitively secure programs are rejected by this

approach, i.e., $\mathtt{h} := \mathtt{l}; \mathtt{l} := \mathtt{h}$. Instead, in [14], we proposed to use a different approach for SSOD, encoding the information flow property as a temporal logic property. This idea is based on the use of self-composition [6, 15], and allows us to verify the information flow property via model checking. However, the result is rather complex, and thus its verification cannot be handled efficiently by the existing model-checking tools.

Therefore, this paper proposes more efficient algorithms to verify our definition. For this purpose, programs are modeled as probabilistic Kripke structures. For both conditions of SSPOD, we present a verification approach, a clever combination of new and existing algorithms. The first condition is checked by removing all stuttering loops, except the self-loops in final states, and then verifying stuttering equivalence. Verification of stuttering equivalence is implemented by checking whether there exists a functional bisimulation between the executions of the Kripke structure and a witness trace. This is a new algorithm, that is also relevant outside the security context, e.g., as in partial-order reduction for model checking, because stuttering equivalence is a fundamental concept in the theory of concurrent and distributed systems. SSOD-1 can be also verified by a variant of this algorithm. SSPOD-2 is implemented by removing stuttering steps, thereby reducing the problem into an equivalence problem for probabilistic languages [28, 11, 16]. This approach gives a precise verification method for observational determinism. Furthermore, the model checking procedure is also able to produce a counter-example to synthesize attacks for insecure programs, i.e., for programs that fail either of the conditions of SSPOD (similar as in [20]).

Currently, we are implementing our verification techniques in the symbolic model checker LTSmin [7]. SSPOD-1 has been implemented, and we will adapt the existing implementation of [16] for SSPOD-2. Once the implementation is finished, we will apply the tool to case studies.

Organization of the paper. Section 2 presents the preliminaries. Then, Section 3 formalizes the SSPOD property, and Section 4 presents its verification. Section 5 discusses related work. Section 6 concludes, and discusses future work.

2 Preliminaries

2.1 Basics

Sequences. Let X be an arbitrary set. The sets of all *finite sequences*, and all *sequences* of X are denoted by X^*, and X^ω, respectively. The empty sequence is denoted by ε. Given a sequence $\sigma \in X^*$, we denote its last element by $last(\sigma)$. A sequence $\rho \in X^*$ is called a *prefix* of σ, denoted by $\rho \sqsubseteq \sigma$, if there exists another sequence $\rho' \in X^\omega$ such that $\rho\rho' = \sigma$.

Probability distributions. A *probability distribution* μ over a set X is a function $\mu \in X \to [0, 1]$, such that the sum of the probabilities of all elements is 1, i.e., $\sum_{x \in X} \mu(x) = 1$ over a set X. If X is uncountable, then $\sum_{x \in X} \mu(x) = 1$ implies

that $\mu(x) > 0$ for countably many $x \in X$. We denote by $\mathcal{D}(X)$ the set of all probability distributions over X. The *support* of a distribution $\mu \in \mathcal{D}(X)$ is the set $supp(\mu) = \{x \in X \mid \mu(x) > 0\}$ of all elements with a positive probability. For an element $x \in X$, we denote by $\mathbf{1}_x$ the probability distribution that assigns probability 1 to x and 0 to all other elements.

2.2 Probabilistic Kripke Structures

We consider probabilistic Kripke structures (PKS) that can be used to model semantics of probabilistic programs in a standard way [13]. PKSs are like standard Kripke structures [17], except that each transition $c \to \mu$ leads to a probability distribution μ over the next states, i.e., the probability to end up in state c' is $\mu(c')$. Each state may enable several probabilistic transitions, modeling different execution orders to be determined by a scheduler. For technical convenience, our PKSs label states with arbitrary-valued variables from a set Var, rather than with Boolean-valued atomic propositions. Thus, each state c is labeled by a labeling function $V(c) : Var \to Val$ that assigns a value $V(c)(v) \in Val$ to each variable $v \in Var$. We assume that Var is partitioned into sets of low variables L and high variables H, i.e., $Var = L \cup H$, with $L \cap H = \emptyset$.

Definition 1 (Probabilistic Kripke structure). *A* probabilistic Kripke structure $\mathcal{A}$ *is a tuple* $\langle \mathcal{S}, I, Var, Val, V, \to \rangle$ *consisting of (i) a set* $\mathcal{S}$ *of* states*, (ii) an initial state* $I \in \mathcal{S}$*, (iii) a finite set of variables* Var*, (iv) a countable set of values* Val*, (v) a* labeling function $V : \mathcal{S} \to (Var \to Val)$*, (vi) a* transition relation $\to \subseteq \mathcal{S} \times \mathcal{D}(\mathcal{S})$*. We assume that* $\to$ *is non-blocking, i.e.,* $\forall c \in \mathcal{S}. \exists \mu \in \mathcal{D}(\mathcal{S}). c \to \mu$.

A PKS is *fully probabilistic* if each state has at most one outgoing transition, i.e., if $c \to \mu$ and $c \to \mu'$ implies $\mu = \mu'$. Given a set $Var' \subseteq Var$, the *projection* $\mathcal{A}_{|_{Var'}}$ of $\mathcal{A}$ on Var', restricts the labeling function V to labels in Var'. Thus, we obtain $\mathcal{A}_{|_{Var'}}$ from $\mathcal{A}$ by replacing V by $V_{|_{Var'}} : \mathcal{S} \to (Var' \to Val)$.

Semantics of probabilistic programs. A program C over a variable set Var can be expressed as a PKS $\mathcal{A}$ in a standard way: The states of $\mathcal{A}$ are tuples $\langle C, s \rangle$ consisting of a program fragment C and a valuation $s : Var \to Val$. The transition relation $\to$ follows the small-step semantics of C. If a program terminates in a state c, we include a special transition $c \to \mathbf{1}_c$, ensuring that $\mathcal{A}$ is non-blocking.

Paths and traces. A *path* π in $\mathcal{A}$ is an infinite sequence $\pi = c_0 c_1 c_2 \ldots$ such that (i) $c_i \in \mathcal{S}, c_0 = I$, and (ii) for all $i \in \mathbb{N}$, there exists a transition $c_i \to \mu$ with $\mu(c_{i+1}) > 0$. We define $Path(\mathcal{A})$ as the set of all infinite paths of $\mathcal{A}$; and $Path^*(\mathcal{A}) = \{\pi' \sqsubseteq \pi \mid \pi \in Path(\mathcal{A})\}$ as the set of all finite paths in $Path(\mathcal{A})$.

The *trace* T of a path π records the valuations along π. Formally, $T = trace(\pi) = V(c_0)V(c_1)V(c_2)\ldots$. Trace T is a *lasso* iff it ends in a loop, i.e., if $T = T_0 \ldots T_i (T_{i+1} \ldots T_n)^\omega$, where $(T_{i+1} \ldots T_n)^\omega$ denotes a loop. Let $Trace(\mathcal{A})$ denote the set of all infinite traces of $\mathcal{A}$. Two states c and c' are *low-equivalent*, denoted $c =_L c'$, iff $V(c)_{|_L} = V(c')_{|_L}$.

2.3 Probabilistic schedulers

A probabilistic scheduler is a function that implements a scheduling policy [23], i.e., that decides with which probabilities the threads are selected. To make our security property applicable for many schedulers, we give a general definition. We allow a scheduler to use the full history of computation to make decisions: given a path ending in some state c, a scheduler chooses which of the probabilistic transitions enabled in c to execute. Since each transition results in a distribution, a probabilistic scheduler returns a distribution of distributions[2].

Definition 2. *A scheduler δ for PKS $\mathcal{A} = \langle \mathcal{S}, I, Var, Val, V, \rightarrow \rangle$ is a function $\delta : Path^*(\mathcal{A}) \rightarrow \mathcal{D}(\mathcal{D}(\mathcal{S}))$, such that, for all finite paths $\pi \in Path^*(\mathcal{A})$, $\delta(\pi)(\mu) > 0$ implies $last(\pi) \rightarrow \mu$.*

The effect of a scheduler δ on a PKS $\mathcal{A}$ can be described by a PKS $\mathcal{A}_\delta$: the set of states of $\mathcal{A}_\delta$ is obtained by unrolling the paths in $\mathcal{A}$, i.e., $\mathcal{S}_{\mathcal{A}_\delta} = Path^*(\mathcal{A})$ such that states of $\mathcal{A}_\delta$ contain a full history of execution. Besides, the unreachable states of $\mathcal{A}$ under the scheduler δ are removed by the transition relation $\rightarrow_\delta$.

Definition 3. *Let $\mathcal{A} = \langle \mathcal{S}, I, Var, Val, V, \rightarrow \rangle$ be a PKS and let δ be a scheduler for $\mathcal{A}$. The PKS associated to δ is $\mathcal{A}_\delta = \langle Path^*(\mathcal{A}), I, Var, Val, V_\delta, \rightarrow_\delta \rangle$, where $V_\delta : Path^*(\mathcal{A}) \times Var \rightarrow Val$ is given by $V_\delta(\pi) = V(last(\pi))$, and the transition relation is given by $\pi \rightarrow_\delta \mu$ iff $\mu(\pi c) = \sum_{\nu \in supp(\delta(\pi))} \delta(\pi)(\nu) \cdot \nu(c)$ for all π, c.*

Since all nondeterministic choices in $\mathcal{A}$ have been resolved by δ, $\mathcal{A}_\delta$ is fully probabilistic. The probability $P(\pi)$ given to a finite path $\pi = \pi_0 \pi_1 \dots \pi_n$ is determined by $\delta(\pi_0)(\pi_1) \cdot \delta(\pi_0 \pi_1)(\pi_2) \cdots \delta(\pi_0 \pi_1 \dots \pi_{n-1})(\pi_n)$. The probability of a finite trace T is obtained by adding the probabilities of all paths associated with T. Based on this observation, we can associate a probability space $(\Omega, \mathcal{F}, \mathbf{P}_\delta)$ over sets of traces. Following the standard definition, we set $\Omega = (Var \rightarrow Val)^\omega$, $\mathcal{F}$ contains all measurable sets of traces, and $\mathbf{P}_\delta : \mathcal{F} \rightarrow [0, 1]$ is a probability measure on $\mathcal{F}$. Thus, $\mathbf{P}_\delta(X)$ is the probability that a trace inside set $X \in \mathcal{F}$ occurs. We refer to [26] for technical details. Notice that Ω and $\mathcal{F}$ depend only on $\mathcal{A}$, not on $\mathcal{A}_\delta$.

2.4 Stuttering-free PKSs and Stuttering Equivalence

Stuttering steps and *stuttering equivalence* [21, 15] are the basic ingredients of our confidentiality properties. In the non-probabilistic case, a stuttering step is a transition $c \rightarrow c'$ that leaves the labels unchanged, i.e., $V(c') = V(c)$. In the probabilistic scenario, a transition stutters if, with positive probability, at least one of the reached states has the same label. A *stuttering-free* PKS allows stuttering transitions only as the self-loops in final states.

[2] Thus, we assume a discrete probability distribution over the uncountable set $\mathcal{D}(\mathcal{S})$; only the countably many transitions occurring in $\mathcal{A}$ can be scheduled with a positive probability.

Definition 4 (Stuttering-free PKS). *A* stuttering step *is a transition $c \to \mu$ with $V(c) = V(c')$ for some $c' \in supp(\mu)$. A PKS is called* stuttering-free *if for all stuttering steps $c \to \mu$, we have that $\mu = \mathbf{1}_c$ and no other transition leaving from c, i.e., if $c \to \mu'$, this implies $\mu = \mu'$.*

Two sequences are stuttering equivalent if they are the same after we remove adjacent occurrences of the same label, e.g., $(\mathbf{aaabcccd})^\omega$ and $(\mathbf{abbcddd})^\omega$.

Definition 5 (Stuttering equivalence). *Let X be a set. Stuttering equivalence, denoted $\sim$, is the largest equivalence relation over $X^\omega \times X^\omega$ such that for all $T, T' \in X^\omega$, $\boldsymbol{a}, \boldsymbol{b} \in X$: $\boldsymbol{a}T \sim \boldsymbol{b}T' \Rightarrow \boldsymbol{a} = \boldsymbol{b} \wedge (T \sim T' \vee \boldsymbol{a}T \sim T' \vee T \sim \boldsymbol{b}T')$. A set $Y \subseteq X$ is* closed under stuttering equivalence *if $T \in Y \wedge T \sim T'$ imply $T' \in Y$.*

3 Scheduler-Specific Probabilistic-Observational Determinism

A program is confidential w.r.t. a particular scheduler iff no secret information can be derived from the observation of public data traces, the order of public data updates, or from the probabilities of traces. This is captured formally by the definition of *scheduler-specific probabilistic-observational determinism*.

As shown in [30, 14], to be secure, a multi-threaded program must enforce an order on the accesses to a single low variable, i.e., the sequence of operations performed at a single low variable is deterministic. Therefore, SSPOD's first condition requires that for any initial state, traces of each low variable that do not end in a non-final stuttering loop are stuttering equivalent with probability 1. This condition ensures that no secret information can be derived from the observation of public data traces, because when all low variables individually evolve deterministically, the values of low variables do not depend on the values of high variables. However, a consequence of SSPOD's first condition is that harmless programs such as `l:=0 ‖ l:=1` are also rejected.

SSPOD also requires that, given any two initial low-equivalent states I and I', for every trace starting in I, there *exists* a trace that is stuttering equivalent *w.r.t. all* low variables, starting in I', and the probabilities of these two matching traces are the same. This condition ensures that secret information cannot be derived from the relative order of updates of any two low variables, or from any probabilistic attack, because there is always a matching trace with the same probability of occurrence.

Let $(\Omega, \mathcal{F}, \mathbf{P}_\delta)$ denote the probability space of $\mathcal{A}_\delta$ with an initial state I. Notice that the probability of a trace to end up in a non-final stuttering loop is 0, because a non-final loop must contain at least one state with a transition that goes out of the loop; thus, it contains a transition with a probability less than 1. Thus, if X is a set of traces that ends in a non-final stuttering loop and are closed under stuttering equivalence, $\mathbf{P}_\delta[X]$ might be 0. Therefore, SSPOD is formally defined as follows.

Definition 6 (SSPOD). *Given a scheduler δ, a program C respects SSPOD w.r.t. L and δ, iff for any initial state I,*

SSPOD-1 *For any $l \in L$, let $X \in \mathcal{F}$ be any set of traces closed under stuttering equivalence w.r.t. l, we have $\mathbf{P}_\delta[X] = 1$ or $\mathbf{P}_\delta[X] = 0$.*

SSPOD-2 *For any initial state I' that is low-equivalent with I, for all sets of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence w.r.t. L, we have $\mathbf{P}_\delta[X] = \mathbf{P}'_\delta[X]$, where $(\Omega, \mathcal{F}, \mathbf{P}'_\delta)$ denote the probability space of $\mathcal{A}_\delta$ with I'.*

Program C is scheduler-specific probabilistic-observational deterministic w.r.t. a set of schedulers Δ if it is so w.r.t. any scheduler $\delta \in \Delta$.

4 Verification of SSPOD

This section discusses how we algorithmically verify the two conditions of SSPOD. As mentioned above, we use a combination of new and existing algorithms. Moreover, the new algorithm is general, and also applicable in other, non-security related contexts. We assume that data domains are finite and schedulers use finite memory. Therefore, the algorithms work only on finite fully probabilistic PKSs, which can be viewed as finite Markov Chains.

4.1 Verification of SSPOD-1

Algorithm. Given a program C, and a scheduler δ, SSPOD-1 requires that after projecting $\mathcal{A}_\delta$ on any low variable l, all traces that do not stutter forever in a non-final stuttering loop must be stuttering equivalent with probability 1. To verify this, we pick one arbitrary trace and ensure that all other traces are stuttering equivalent to this trace. Concretely, for each $l \in L$, we carry out the following steps.

SSPOD-1 on l

1: Project $\mathcal{A}_\delta$ on l, yielding $\mathcal{A}_{\delta\,|_l}$.
2: Remove all stuttering loops in $\mathcal{A}_{\delta\,|_l}$.
3: Re-establish the self-loops for final states of $\mathcal{A}_{\delta\,|_l}$. This yields a stuttering-loop free PKS, denoted $\mathcal{R}_{\delta\,|_l}$.
4: Check whether all traces of $\mathcal{R}_{\delta\,|_l}$ are stuttering equivalent by:
  4.1: Choose a *witness* trace by:
    4.1.1: Take an arbitrary lasso T of $\mathcal{R}_{\delta\,|_l}$.
    4.1.2: Remove stuttering steps and minimize T.
  4.2: Check stuttering trace equivalence between $\mathcal{R}_{\delta\,|_l}$ and T by checking if there exists a functional bisimulation between them.

This algorithm works, since we transform the probabilistic property SSPOD-1 into a possibilistic one. Key insight is that the probability of a trace that stutters forever in a non-final stuttering loop is 0. Therefore, after removing all non-final stuttering loops, it is sufficient to determine whether all traces are stuttering equivalent.

To perform Step 1, we label every state with the value of l in that state. To remove the stuttering loops in Step 2, we use a classical algorithm for finding strongly connected components w.r.t. stuttering steps [1], and collapse these components into a single state. To ensure that the transition relation remains non-blocking, Step 3 re-establishes the self-loops for final states.

Step 4.1.1 is implemented via a classical cycle-detection algorithm based on depth-first search (Appendix A). The initial state of a lasso is also the initial state of PKS. The algorithm essentially proceeds by picking arbitrary next steps, and terminates when it hits a state that was picked before. Step 4.1.2 is done via the standard strong bisimulation reduction. For example, the minimal form of a lasso $\mathbf{abb}(\mathbf{cb})^\omega$ is $\mathbf{a}(\mathbf{bc})^\omega$. This minimal lasso is called the *witness trace*.

Step 4.2 checks stuttering trace equivalence between a PKS $\mathcal{A}$ and the witness trace T by checking if there exists a functional bisimulation between them, i.e., a bisimulation that is a function, thus mapping each state in $\mathcal{A}$ to a single state in T. This is done by exploring the state space of $\mathcal{A}$ in a breadth-first search (BFS) order and building the mapping *Map* during exploration. We name each state in T by a unique symbol $u \in \mathcal{U}$, i.e., u_i denotes T_i. Let $succ(T, u)$ denote the successor of u on T.

We map the $\mathcal{A}$'s initial state to u_0, i.e., $Map[init\_state] = u_0$. Each iteration of the algorithm examines the successors of the state stored in the variable *current*. Assume that $Map[current]$ is u, consider a successor $c \in succ(\mathcal{A}, current)$. The *potential_map* of c is u if $current \rightarrow c$ is a stuttering transition; otherwise, it is $succ(T, u)$. The algorithm returns *false*, i.e., $continue = false$, if (i) c and *potential_map* have different valuations, (ii) c is a final state of $\mathcal{A}$, while *potential_map* is not the final state of T, or (iii) c has been checked before, but its mapped state is not *potential_map*.

If none of these cases occurs and c was not checked before, c is added to Q, and mapped to *potential_map*. Basically, a state c of $\mathcal{A}$ is mapped to u, i.e., $Map[c] = u$, iff the trace from the initial state to state c in $\mathcal{A}$ and the prefix of T upto u are stuttering equivalent.

Let $c \sim_V c'$ denote that c and c' have the same valuation, i.e., $V(c) = V(c')$; $final(\mathcal{A}, c)$ denote that c is a final state in $\mathcal{A}$; and $final(T, u)$ denote that u is the final state in T. The algorithm also uses a FIFO queue Q of *frontier* states. The termination of Algorithm 4.2 follows from the termination of BFS over a finite $\mathcal{A}$.

4.2: Stuttering Trace Equivalence($\mathcal{A}$, T)

for all states $c \in \mathcal{S}$ **do** $Map[c] := \perp$;
$continue := true$;
$Q := empty\_queue()$; $enqueue(Q, init\_state)$;
$Map[init\_state] := u_0$; `// u0 is T0`
while $!empty(Q) \wedge continue$ **do**
    $current := dequeue(Q)$;
    $u := Map[current]$;
    for all states $c \in succ(\mathcal{A}, current)$ **do**
        $potential\_map := (c \sim_V current)\ ?\ u : succ(T, u)$;

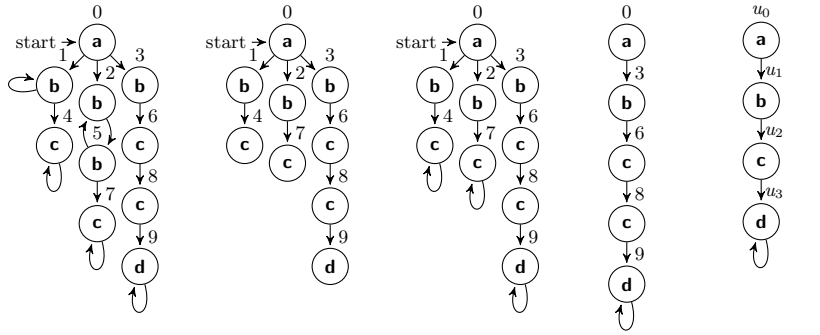

Fig. 1: Step 1 - Step 4.1

case $c \not\sim_V$ *potential_map* $\rightarrow$ *continue* := *false*;
$[\!]$ *final*$(\mathcal{A}, c) \wedge \neg$*final*$(T,$ *potential_map*$)$ $\rightarrow$ *continue* := *false*;
$[\!]$ $Map[c] = \bot$ $\rightarrow$ *enqueue*(Q, c);
$Map[c]$:= *potential_map*;
$[\!]$ $Map[c] \neq$ *potential_map* $\rightarrow$ *continue* := *false*;
return *continue*;

Example 1. Figure 1 illustrates Step 1 - Step 4.1 on a PKS $\mathcal{A}$ consisting of 10 states. Step 1 projects $\mathcal{A}$ on a low variable l. The symbols **a**, **b**, **c** etc. denote state contents, i.e., states with the same value of l are represented by the same symbol. Step 2 removes all stuttering loops, while Step 3 re-establishes the self-loops for final states. Step 4.1 takes an arbitrary trace of $\mathcal{A}$ and then minimizes it. Each state of the witness trace T is denoted by a unique symbol u_i. Figure 2 illustrates Step 4.2. Initially, all states of $\mathcal{A}$ are mapped to a special symbol $\bot$ that indicates unchecked states. To keep states readable, we skip the valuation. Next, state 0 is enqueued, and mapped to u_0. Next, the algorithm examines all unchecked successors of state 0, i.e., states 1, 2, 3. Each of them follows a non-stuttering step, thus their *potential_map*s are all u_1. Since states 1, 2, and 3 have the same valuation as *potential_map*, i.e., **b**, they are all enqueued, and mapped to u_1. Next, the successor of state 1, i.e., state 4, is considered. The transition $1 \rightarrow 4$ is non-stuttering, thus *potential_map* $= u_2$. State 4 has the same valuation as *potential_map*, but it is a final state of $\mathcal{A}$, while *potential_map* is not the final state of T. Thus, *continue* = *false*. The PKS $\mathcal{A}$ and the witness trace T are not stuttering trace equivalent, because there exists a trace that stutters in state 4 forever. The algorithm terminates.

Theorem 1. *Algorithm 4.2 returns true iff there exists a bisimulation between $\mathcal{A}$ and T.*

Proof. See Appendix B.

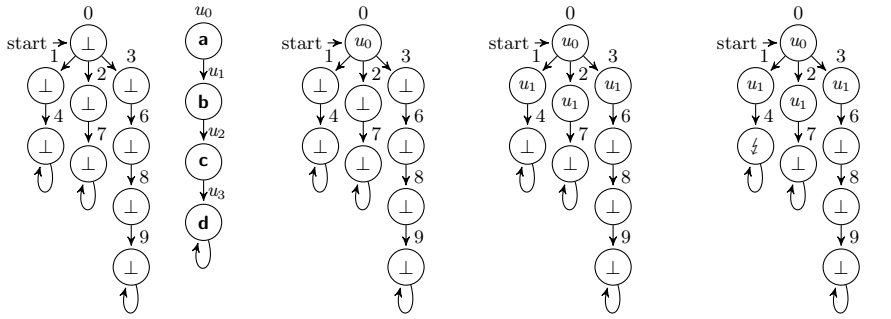

Fig. 2: Step 4.2

Overall Correctness. Step 1 only changes the labels of states of a PKS. Thus, the probability space of the PKS is unchanged. Hence, after projecting $\mathcal{A}_\delta$ on l, we can reformulate SSPOD-1 in terms of $\mathcal{A}_{\delta\,|_l}$. Let $(\Omega, \mathcal{F}, \mathbf{P}_{\delta,l})$ denote the probability space of $\mathcal{A}_{\delta\,|_l}$. First, we reformulate SSPOD-1, which talks about projected traces in $\mathcal{A}$, in terms of the traces in the projected $\mathcal{A}_{\delta,l}$

Theorem 2. *For any $l \in L$, and for a set of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence, if $\mathbf{P}_{\delta,l}[X] = 1$ or $\mathbf{P}_{\delta,l}[X] = 0$, then SSPOD-1 holds.*

Proof. See Appendix C.

The key step (Step 2) in our algorithm is the reduction of a probabilistic property to a non-probabilistic property: after removing all stuttering loops, if all traces of $\mathcal{A}_{\delta\,|_l}$ are stuttering equivalent, then $\mathbf{P}_{\delta,l}[X] = 1$. Thus, SSPOD-1 holds. The correctness of this step follows from a result from Baier and Kwiatkowska [5]: whenever *all fair traces* of a PKS fulfill a certain property φ, then φ holds *with probability* 1. In our context, we define the fairness of traces w.r.t. *non-stuttering transitions*. A non-stuttering transition is *enabled* in a state T_i iff there exists a finite sequence of transitions from T_i that leads to T_j such that $V(T_j) \neq V(T_i)$. A non-stuttering transition is said to be *taken* in a state T_i of T iff $\exists j > 0.\, T_i \neq T_{i+j}$. A trace is *strongly fair* w.r.t. non-stuttering transitions if given that a non-stuttering transition is enabled infinitely often, it is taken infinitely often. Thus, a trace that stutters in a non-final stuttering loop forever is unfair. Let $\mathit{Fair}(\mathcal{A})$ denote the set of fair traces of $\mathit{Trace}(\mathcal{A})$. Applying the result from [5], we obtain:

Theorem 3. *Given a finite $\mathcal{A}_{\delta\,|_l}$ and a set of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence and do not stutter forever in a non-final stuttering loop, if $\forall T, T' \in \mathit{Fair}(\mathcal{A}_{\delta\,|_l}).\, T \sim T'$, then $\mathbf{P}_{\delta,l}[X] = 1$.*

We show that after removing all stuttering loops, and re-establishing the self-loops for final states, the set of fair traces of $\mathcal{A}$ is preserved.

Theorem 4. *Given a PKS $\mathcal{A}$, let $\mathcal{R}$ denote the PKS that is obtained after removing all stuttering loops and re-establishing the self-loops for final states. Then $Fair(\mathcal{A}) = Trace(\mathcal{R})$.*

Proof. See Appendix D.

Combining these results, we obtain.

Theorem 5. *For any $l \in L$, if all traces of $\mathcal{R}_{\delta\,|_l}$ are stuttering equivalent, then SSPOD-1 holds.*

Overall Complexity. Step 1 labels every state of $\mathcal{A}$ by the value of l in that state. This is done in time complexity $O(n)$, where n is the number of states of $\mathcal{A}$. Step 2 uses an $O(m)$-algorithm to find the strongly connected components, where m is the number of transitions of $\mathcal{A}$. The time complexity of Step 4.1 is also $O(m)$. The core of Step 4.2 is the BFS algorithm, whose running time is $O(n+m)$. Therefore, for a single low variable l, the total time complexity of the verification is linear in the size of $\mathcal{A}$, i.e., $O(n+m)$, and for any initial state, the total complexity of the verification of SSPOD-1 (for all $l \in L$) is $|L|\,O(n+m)$.

4.2 Verification of SSPOD-2

Algorithm. SSPOD-2 states that, given a program C, for any two initial low-equivalent states I and I', if we project on the set of low variables L, the probabilistic languages arising from the executions of I and I' should be the same. A number of efficient algorithms for checking equivalence between probabilistic languages have been developed, the classical ones in [10, 28], and the improved variants in [11, 16]. However, none of the existing algorithms exactly fit our purposes, since either they do not abstract from stuttering steps [28, 11, 16], or they consider a different variation of probabilistic language inclusion [10].

Therefore, to verify SSPOD-2, our algorithm first transforms the PKS into an equivalent one, without *stuttering steps*, and then we use the latest and most efficient algorithm from Kiefer et al. [16] to check equivalence of these probabilistic languages. The basic idea of this algorithm is to present the language of a PKS by a polynomial in which each monomial presents an input word of the language and the coefficient of the monomial represents the weight of the word, i.e., the probability of the execution of the word. This method reduces the language equivalence problem to polynomial identity testing.

SSPOD-2

1: Project both $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ (modeling the executions starting in I and I') on the set L, yielding $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$.

2: Remove all stuttering steps from $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$, yielding stuttering-free PKSs $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$.

3: Check the equivalence of the stuttering-free probabilistic languages between $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$, using Kiefer et al. [16].

Overall Correctness. After projecting both $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ on L, we can reformulate SSPOD-2 in terms of $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$. Let $(\Omega, \mathcal{F}, \mathbf{P}_{\delta,L})$ and $(\Omega, \mathcal{F}, \mathbf{P}'_{\delta,L})$ denote the probability space of $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$, respectively.

Theorem 6. *SSPOD-2 holds iff for all sets of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence, we have $\mathbf{P}_{\delta,L}[X] = \mathbf{P}'_{\delta,L}[X]$.*

Proof. See Appendix E.

Let $\mathcal{R}$ denote a stuttering-free PKS that is obtained by applying Step 2 on a given $\mathcal{A}$. Let $P_\mathcal{A}$ and $P_\mathcal{R}$ be the probabilistic transition functions of $\mathcal{A}$ and $\mathcal{R}$, respectively. Step 2 removes all stuttering steps by changing $P_\mathcal{A}$ to $P_\mathcal{R}$ given by the following equations.

$$P_\mathcal{R}(c, c') = \begin{cases} P_\mathcal{A}(c, c') & \text{if } V(c) \neq V(c') \\ \sum_{c'': V(c)=V(c'')} P_\mathcal{A}(c, c'')\, P_\mathcal{R}(c'', c') & \text{otherwise.} \end{cases}$$

Thus, for non-stuttering steps, $P_\mathcal{A}$ and $P_\mathcal{R}$ are the same; for stuttering steps, $P_\mathcal{R}$ accumulates the probabilities of moving to c' via some stuttering steps $c \to c''$. Thus, $P_\mathcal{R}$ accounts for the transition probabilities of stuttering steps in $\mathcal{A}$ into the transition probabilities of non-stuttering steps in $\mathcal{R}$. Therefore, removing stuttering steps does not change the probabilities of sets of traces that are closed under stuttering equivalence.

Theorem 7. *Let $X \in \mathcal{F}$ be a set of traces that are closed under stuttering equivalence, then $\mathbf{P}_\mathcal{A}[X] = \mathbf{P}_\mathcal{R}[X]$.*

Combining all results, it is obvious that to check SSPOD-2, we can check for probabilistic language equivalence between $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$.

Overall Complexity. Step 1 is done in time complexity $O(n)$, where n is the number of states of two PKSs. Step 2 essentially calculates a reachability probability, and is defined as a system of n linear equations over n variables. This equation system can be solved in $O(n^3)$. Step 3 can be done in $O(nm)$, where m is the number of transitions [16]. Thus, the overall complexity is $O(n^3)$ for each pair I and I'.

5 Related Work

The idea of observational determinism originates from Roscoe [22], who was the first to identify the need for determinism to ensure confidentiality for concurrent processes. This observation has resulted in several subtly different definitions of observational determinism for possibilistic multi-threaded programs (e.g., [30, 15, 27, 14]), see [14] for a detailed comparison. Notice that, SSOD is the only one to consider the effect of schedulers on confidentiality.

When programs have probabilistic behaviors, to prevent information leakage under probabilistic attacks, several notions of probabilistic noninterference have

been proposed [29, 23, 24]. The first is from Volpano and Smith [29]. It is based on a lock-step execution of probability distributions on states, i.e., given any two initial states that are indistinguishable w.r.t. low variables, the executions of the program from these two initial states, after projecting out high variables, are exactly the same. As shown by Sabelfeld and Sands [23], this definition is not precise, and overly restrictive. Sabelfeld and Sands's definition of probabilistic noninterference is based on a *partial probabilistic low-bisimulation* [23], which requires that given any two initial states that are indistinguishable w.r.t. low variables, for any trace that starts in an initial state, *there exists* a trace that starts in the other initial state and passes through the same equivalence classes of states at the same time, with the same probability. This definition is restrictive *w.r.t.* timing, i.e., it cannot accommodate threads whose running time depends on high variables. Thus, it rejects many harmless programs, while SSPOD accepts, such as $\texttt{if}\ (\texttt{h} > 0)\ \texttt{then}\ \{\texttt{l1} := 3;\ \texttt{l1} := 3;\ \texttt{l2} := 4\}\ \texttt{else}\ \{\texttt{l1} := 3;\ \texttt{l2} := 4\}$.

To overcome these limitations, Smith proposes to use a weak probabilistic bisimulation [24]. Weak probabilistic bisimulation allows two traces to be equivalent when they reach the same outcome, but one runs slower than the other. However, this still demands that any two bisimilar states must reach indistinguishable states with the same probability. This condition of probabilistic bisimulation is more restrictive than SSPOD, because when trace occurrences do not depend on high variables, probabilistic noninterference still rejects the program.

Moreover, all bisimulation-based definitions mentioned above do not require the deterministic behavior of each low variable. However, we insist that a multi-threaded program must enforce a deterministic orderings on the accesses to low variables, see [14]. Finally, probabilistic noninterference [23, 24] also put restrictions on unreachable states, e.g., $\texttt{l} := 1;\ \texttt{if}\ (\texttt{l} == 0)\ \texttt{then}\ \texttt{l} := \texttt{h}\ \texttt{else}\ \texttt{skip}$ is secure but rejected, because the bisimulation also considers the case when the conditional statement is executed from an unreachable state where $\texttt{l}$ equals 0, see [8]. Mantel et al. [18] overcome this limitation by explicitly using assumptions and guarantees about how threads access the shared memory. Notice that SSPOD does not have this property, thus SSPOD is less restrictive.

Mantel et al. [19] also consider the effect of schedulers on confidentiality. However, their observational model is different from ours. They assume that the attacker can only observe the initial and final values of low variables on traces. Thus, their definitions of confidentiality are noninterference-like.

Palamidessi et al., Chen et al., Smith, and Zhu et al. [2–4, 9, 25, 31] investigate a quantitative notion of information leakage for probabilistic systems. Quantitative analysis offers a method to compute *bounds* on how much information is leaked. This information can be used to compare with the threshold, and thus suggesting whether the program is accepted or not. Therefore, we can tolerate the *minor* leakage. Thus, this line of researches is complementary to ours.

6 Conclusion

Summary. This paper introduces the notion of *scheduler-specific probabilistic observational determinism*, together with an algorithmic verification technique.

SSPOD captures the notion of confidentiality for probabilistic multi-threaded programs. The definition extends an earlier proposal for possibilistic confidentiality of such programs, and makes it usable in a larger context. It is important to consider probabilistic multi-threaded programs, because this captures the realistic behavior of programs.

We also propose an algorithmic verification technique for it. The verification is using a combination of new and existing algorithms. The new algorithm solves a standard problem, which makes it applicable also in a broader context. We believe that the idea of adapting known model checking algorithms will also be appropriate for other security properties, such as integrity and availability.

Future work. We see several directions for future work. We plan to continue the study of other security properties, i.e., anonymity, integrity, and availability. We believe that our algorithmic approach is also appropriate to efficiently and precisely verify these security properties.

Further, we also plan to relax our definitions of confidentiality by quantifying the information flow and determining how much information is being leaked. The existing models of quantitative analysis do not address which measure is suitable to quantify information leakage for multi-threaded programs, thus a new approach has to be developed.

Acknowledgment: Our work is supported by NWO under grant 612.067.802 (SLALOM) and grant Dn 63-257 (ROCKS).

References

1. A.V. Aho and J.E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1st edition, 1974.
2. M. S. Alvim, M. E. Andrés, K. Chatzikokolakis, and C. Palamidessi. On the relation between differential privacy and quantitative information flow. In *ICALP (2)*, volume 6756 of *LNCS*. Springer, 2011.
3. M. S. Alvim, M. E. Andrés, K. Chatzikokolakis, and C. Palamidessi. Quantitative information flow and applications to differential privacy. In *FOSAD Tutorial Lectures*, volume 6858 of *LNCS*. Springer, 2011.
4. M. Andres, E., C. Palamidessi, A. Sokolova, and P. Van Rossum. Information hiding in probabilistic concurrent systems. *Journal of Theoretical Computer Science*, 412(28):3072–3089, 2011.
5. C. Baier and M. Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters*, 66:71–79, 1998.
6. G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Press, 2004.
7. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *CAV '10*, volume 6174 of *LNCS*, pages 354–359, 2010.

8. H.-C. Blondeel. Security by logic: characterizing non-interference in temporal logic. Master's thesis, KTH Sweden, 2007.
9. H. Chen and P. Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *PLAS '07*, 2007.
10. L. Christoff and I. Christoff. Efficient algorithms for verification of equivalences for probabilistic processes. In *CAV '91*, LNCS, pages 310–321. Springer-Verlag, 1992.
11. L . Doyen, T.A. Henzinger, and J.F. Raskin. Equivalence of labeled Markov chains. *Int. J. Found. Comput. Sci.*, 19(3):549–563, 2008.
12. J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.
13. A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction. In *In TACAS'06*, 2006.
14. M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In *FoVeOOS'11*, 2012.
15. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterization of observation determinism. In *CSFW*. IEEE Computer Society, 2006.
16. S. Kiefer, A.S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell. Language equivalence for probabilistic automata. In *CAV '11*, pages 526–540. Springer-Verlag, 2011.
17. S.A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.
18. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF '11*, pages 218–232, 2011.
19. H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *ESORICS*, pages 116–133, 2010.
20. T.M. Ngo, M. Stoelinga, and M. Huisman. Effective verification of confidentiality for multi-threaded programs. Manuscript 201X.
21. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63:243–246, 1997.
22. A.W. Roscoe. CSP and determinism in security modeling. In *IEEE Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society, 1995.
23. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, pages 200–214, 2000.
24. G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *CSFW*, 2003.
25. G. Smith. On the foundations of quantitative information flow. In *FOSSACS '09*, 2009.
26. M.I.A. Stoelinga. *Alea jacta est: verification of probabilistic, real-time and parametric systems*. PhD thesis, University of Nijmegen, the Netherlands, April 2002.
27. T. Terauchi. A type system for observational determinism. In *CSF*, 2008.
28. W.G. Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM Journal on Computing*, 21:216–227, April 1992.
29. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7:231–253, 1999.
30. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 29–43. IEEE, 2003.
31. J. Zhu and M. Srivatsa. Quantifying information leakage in finite order deterministic programs. In *CoRR '10*, 2010.

A Algorithm to take a lasso of a PKS

The following appendices are for reviewing only. This appendix introduces the algorithm that implements Step 4.1.1.

4.1.1: Lasso T of $\mathcal{A}$

```
for all states c ∈ S do Visit[c] := 0;
index := 0;
current := init_state;
for (;;) do
    T[index] := current;      // Implement T as an array
    index := index + 1;
    if Visit[current] = 1 then break;
    Visit[current] := 1;
    current := some state c ∈ succ(A, current);
return(T, position of current in T);
```

In this algorithm, we use an array *Visit* to indicate visited states of $\mathcal{A}$, i.e., $Visit[current] = 1$ indicates that *current* has been visited before. Clearly, this algorithm returns a trace of $\mathcal{A}$. Moreover, it always terminates, because $\mathcal{A}$ is finite and there is a self-loop in every final state.

B Proof of Correctness of Algorithm 4.2

B.1 Loop Invariant of Algorithm 4.2

We first discuss the loop invariant of Algorithm 4.2.

Theorem 8. *Algorithm 4.2 preserves the following loop invariant:*
If continue then $\forall c \in \mathcal{S}$ *such that* $Map[c] = u$*, the trace from init_state to c and the prefix of* T *upto* u *are stuttering equivalent, and if* $\neg$*continue then there exists a trace of* $\mathcal{A}$ *that is not stuttering equivalent to* T*.*

Proof. Clearly, the invariant holds upon first entry of the loop, since initially, *continue* holds, and only *init_state* is mapped to the initial state of T, i.e., u_0.

We show that the invariant is preserved by every iteration of the loop. Assume the invariant holds before the loop body. If *continue* does not hold, then the loop is not executed, and the algorithm ends. The invariant is preserved.

Otherwise, *continue* holds. The invariant before the loop body states that the trace from *init_state* to *current* and the prefix of T upto u are stuttering equivalent. Now consider a successor c of the *current* state. We distinguish the following cases:

Case $c \not\sim_V u$ **and** $c \not\sim_V succ(T, u)$**.** In our algorithm, *potential_map* denotes the candidate mapping of c. It is u if $c \sim_V current$; otherwise, it is $succ(T, u)$. If $c \not\sim_V u$ and $c \not\sim_V succ(T, u)$, then $c \not\sim_V potential\_map$. Thus, *continue* becomes *false*. The invariant is preserved, because any trace that goes from *current* to c is not stuttering equivalent to T.

Case $c \sim_V u$ **or** $c \sim_V succ(T, u)$**.** Thus, $c \sim_V potential\_map$. Now, we consider the following cases:

Case $final(\mathcal{A}, c) \wedge \neg final(T, potential\_map)$**.** Thus, *continue* becomes *false*. The invariant is preserved, because if c is a final state of $\mathcal{A}$, then there must be a trace that stutters in c forever, while T can make a step from *potential_map* to another state with a different valuation due to the fact that T is stuttering-free.

Case $\neg(final(\mathcal{A}, c) \wedge \neg final(T, potential\_map))$**.**

Case c **is unchecked.** Thus, $Map[c] = \bot$. State c is added to Q, and becomes a frontier state. Moreover, it is mapped to *potential_map*. It is easy to see that the trace from *init_state* to c and the prefix of T upto *potential_map* are stuttering equivalent. Hence, the invariant is preserved.

Case c **is checked before.** Thus, $Map[c] \neq \bot$.

Case $Map[c] = potential\_map$**.** State c has been explored before; the algorithm does not explore it further. Since *continue* and *Map* are not updated, the invariant is preserved.

Case $Map[c] \neq potential\_map$**.** Thus, *continue* becomes *false*. The invariant is preserved, because there exist two traces that both lead to c and in these two traces, c is mapped to two different states of T; thus, one of these two trace is not stuttering equivalent to T. □

B.2 Proof of Theorem 1

Proof. The algorithm preserves the following loop invariant: If *continue*, then $\forall c \in \mathcal{S}$ such that $Map[c] = u$, the trace from *init_state* to c in $\mathcal{A}$ and the prefix of T upto u are stuttering equivalent, and if $\neg$*continue* then there exists a trace of $\mathcal{A}$ that is not stuttering equivalent to T (see Appendix B.1).

If the algorithm returns *false*, it follows directly from the invariant that no functional bisimulation exists. If it returns *true*, we can conclude that for any trace of $\mathcal{A}$, e.g., $T1$, there exists a prefix of T that is stuttering equivalent to $T1$. We show that $T1$ is actually stuttering equivalent to the whole T.

Case $T1$ **ends with a final state** c**.** Assume that c is mapped to *potential_map*. Since the algorithm is termination-sensitive, *potential_map* is also the final state of T. Thus, $T1$ and T are stuttering equivalent.

Case $T1$ **ends with a non-stuttering loop that starts and ends in** c**.** Assume that c is mapped to *potential_map*. State c is investigated twice, and during the second visit, its mapped state is also *potential_map*; otherwise, the algorithm returned *false*. Hence, *potential_map* is also the start and end of a loop that terminates T. Thus, $T1$ and T are stuttering equivalent. □

C Proof of Theorem 2

Proof. Let T be a trace of $\mathcal{A}_\delta$ and let $T_{|_l}$ denote the projection of T on l. First, notice that $\{T_{|_l} \mid T \in Trace(\mathcal{A}_\delta)\} = Trace(\mathcal{A}_{\delta\,|_l})$. Let $X \subseteq Trace(\mathcal{A}_\delta)$ such that

X is closed under stuttering equivalence w.r.t. l. Clearly, also $X \subseteq \mathit{Trace}(\mathcal{A}_{\delta\,|_l})$, and X is closed under stuttering equivalence. Moreover, the probability space of $\mathcal{A}_\delta$ is preserved by projection on l. Thus, if for any l, $\mathbf{P}_{\delta,l}[X] = 1$ or $\mathbf{P}_{\delta,l}[X] = 0$, then $\mathbf{P}_\delta[X] = 1$ or $\mathbf{P}_\delta[X] = 0$, respectively. Thus, SSPOD-1 holds. □

D Proof of Theorem 4

Proof. Let $\mathcal{L}$ be a non-final stuttering loop of $\mathcal{A}$. Since $\mathcal{L}$ is non-final, it contains at least a state with an outgoing transition that leads to a non-stuttering transition. From the definition of fair traces, any T that is *trapped* in $\mathcal{L}$ forever is unfair. Hence, removing all stuttering loops, and re-establishing the self-loops for final states, preserve the set of fair traces of $\mathcal{A}$. □

E Proof of Theorem 6

Proof. Step 1 labels states of a PKS by the set of values of L. This preserves the probability space of the PKS. Let $X \subseteq \mathit{Trace}(\mathcal{A}_\delta)$ such that X is closed under stuttering equivalence w.r.t. L. Clearly, also $X \subseteq \mathit{Trace}(\mathcal{A}_{\delta\,|_L})$, and X is closed under stuttering equivalence. Moreover, $\mathbf{P}_{\delta,L}[X] = \mathbf{P}_\delta[X]$. Thus, SSPOD-2 holds. □