Effective Verification of Confidentiality for Multi-Threaded Programs

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. This paper studies how confidentiality properties of multi-threaded programs can be verified efficiently by a combination of newly developed and existing model checking algorithms. In particular, we study the verification of scheduler-specific observational determinism (SSOD), a property that characterizes secure information flow for multi-threaded programs under a given scheduler. Scheduler-specificness allows us to reason about *refinement attacks*, an important and tricky class of attacks that are notorious in practice. SSOD imposes two conditions: (SSOD-1) all individual public variables have to evolve deterministically, expressed by requiring stuttering equivalence between the traces of each individual public variable, and (SSOD-2) the relative order of updates of public variables is coincidental, i.e., there always exists a matching trace. We verify the first condition by reducing it to the question whether all traces of each public variable are stuttering equivalent. To verify the second condition, we show how the condition can be translated, via a series of steps, into a standard strong bisimulation problem. Our verification techniques can be easily adapted to verify other formalizations of similar information flow properties.
We also exploit counter example generation techniques to synthesize attacks for insecure programs that fail either SSOD-1 or SSOD-2, i.e., showing how confidentiality of programs can be broken.

1 Introduction

Finding ways to guarantee confidentiality of applications remains a difficult, but highly important task. Applications such as Internet banking, medical information systems, and authentication systems need to enforce strict protection of private data, e.g., credit card details, medical records etc. In particular, private information should not be derivable from public data[1]. For example, the program `if (h > 0) then l := 0 else l := 1`, where h is a private variable and l is a public variable, leaks secret information, since we can derive the value of h

[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.

from the value of l. If private data is not sufficiently protected, users refuse to use such applications. Using formal means to establish confidentiality is a promising way to gain the trust of users.

Various techniques are capable of modeling and analyzing the confidentiality property. Classical approaches are typically based on type systems (see [34] for an overview): if a program can be typed, it ensures secure information flow. Type systems are efficient, but imprecise, and also reject many innocuous programs.

Therefore, recent work on adopting techniques from model checking [8, 15, 24] is emerging as an alternative approach to gain better precision. An interesting challenge is to extend these results to the multi-threaded case. This is important, since with the development of multiple cores on a chip and massively parallel systems like general-purpose graphic processing units, multi-threading is becoming the standard. However, this is also a challenge, for two reasons. First of all, the formalization of confidentiality for multi-threaded programs is not easy. The outcomes of multi-threaded programs depend on the scheduling policy. Moreover, because of the interactions between threads and the exchange of intermediate results, we should take into account intermediate results in the model of observations [44, 24, 23]. Existing confidentiality properties, such as *noninterference* [18] and *observational determinism* [44, 24] only consider input-output behavior, and ignore the role of schedulers. Thus, they are not suitable to ensure confidentiality for multi-threaded programs. New definitions of confidentiality 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.

The second challenge is, given an appropriate confidentiality property, to develop an efficient and precise verification algorithm for the property. While various, subtly different approaches to formalize multi-threaded confidentiality have been proposed [33, 44, 24, 39, 22], efficient verification techniques for these properties are still lacking. Therefore, this paper develops methods to verify these important information flow properties by combining newly developed and existing model checking algorithms.

Observational Determinism. This paper studies the verification of scheduler-specific observational determinism (SSOD). This is a formalization of the secure information flow requirements for multi-threaded programs. As observed by Roscoe [33], for a multi-threaded program, not to leak information about private data, its public data have to behave deterministically. We formalized this as SSOD. In contrast to other formalizations of observational determinism, SSOD explicitly considers the scheduler that is used to execute the program. Indeed, due to the interactions between threads, data traces of a multi-threaded program depend on the scheduling policy. Therefore, the program's confidentiality is only guaranteed under a particular scheduler. A different scheduler might make the program reveal secret information, as illustrated by the following example.

Example 1.

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

where $\|$ is the parallel operator. Under the uniform scheduler, i.e., a scheduler that chooses threads uniformly and thus all possible interleavings of threads are considered, the secret information cannot be derived, since the traces in the cases $h > 0$ and $h \leq 0$ 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 it is so, the attacker knows that $h > 0$.

Since we assume that an attacker knows the full source code of the program, if he chooses an appropriate scheduler, secret information can be revealed from the limited set of possible traces. This sort of attack is called a *refinement attack* [34, 8], since the choice of the scheduling policy refines the set of possible program traces. Therefore, by formulating a confidentiality requirement that is parametric over the scheduling policy allows to qualify which scheduler (or classes of schedulers) can be used to securely execute a program.

SSOD imposes two requirements on the possible behaviors of a program:

- (SSOD-1) each public variable has to evolve deterministically. This is captured by requiring that, for any two initial states I and I' that are indistinguishable w.r.t. the public variables, *all possible* traces of a *single* public variable starting in I and I' are stuttering equivalent; and
- (SSOD-2) the relative order of updates of public variables is coincidental. This is captured by requiring that for any two initial states I and I' that are indistinguishable w.r.t. the public variables, and 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 uniform scheduler, but is rejected under the scheduler that always executes the leftmost thread first.

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 [23], we compare SSOD with the existing formalizations of confidentiality properties [44, 24, 39], and argue that they are either unsuitable to the multi-threaded context, or more restrictive than SSOD.

Verification. To verify SSOD, we extract a Kripke structure from the execution of a program under the control of a scheduler, in a standard way. To check our first requirement, we reduce it to the problem of verifying whether all traces of each public variable of a certain Kripke structure are stuttering equivalent. Our algorithm to verify all-trace stuttering equivalence is implemented by checking whether there exists a functional bisimulation between 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, since stuttering equivalence is a fundamental concept in the theory of concurrent and distributed systems [6].

Our second requirement reduces to check the stuttering trace equivalence between two Kripke structures. To check this, we first remove stuttering steps, and then determinize two Kripke structures. Next, we check whether these two deterministic and stuttering-free Kripke structure are strongly bisimilar. Our verification is based on the well-known fact that in deterministic and stuttering-free Kripke structures, trace equivalence and strong bimisimulation coincides [16].

Our approach gives a precise verification method for confidentiality. We would like to stress that other formalizations of observational determinism [44, 24, 39] can also be verified by a minor modification of our algorithms.

Another advantage of using model checking techniques to verify information flow properties is that we can synthesize attacks for insecure programs, based on counter example generation techniques. Since the verification algorithm is precise, if it fails, a counter example can be produced, describing a possible attack on the security of the program. This paper describes how the verification algorithms can be instrumented to produce these counter examples. We believe that our idea of applying counter example generation to synthesize attacks for confidentiality property of multi-threaded programs has not previously been mentioned in literature.

Currently, we are implementing our verification techniques in the symbolic model checker LTSmin [10]. The algorithms are implemented, and we are now applying the implementation to case studies.

The formal definition of SSOD, and the comparison with other formalizations of observational determinism in the literature, has been published before in FoVeOOs [23, 22] and has also been presented at SecCo 2011[2]. However, in this earlier version, we used a different verification technique: characterizing SSOD as a temporal logic formula that leads to a model checking problem of very high complexity. This paper proposes more efficient algorithms. Both verification algorithms and the attack synthesis in this paper have not been published before. The algorithm to check SSOD-1 borrows ideas from the one we developed for the verification of scheduler-specific probabilistic observational determinism (SSPOD), a probabilistic version of SSOD [30], but it solves a completely different problem: SSOD-1 checks if all traces are stuttering equivalent, while SSPOD-1 in essence checks that all traces are stuttering equivalent *with probability 1.*

Organization of the paper. The rest of this paper is organized as follows. After the preliminaries in Section 2, Section 3 presents a formal definition of SSOD. Section 4 discusses the algorithms to verify the property. Section 5 discusses the attack synthesis. Finally, Sections 6, and 7 discuss related work, and conclusions.

[2] The 9th International Workshop on Security Issues in Concurrency

2 Preliminaries

2.1 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 $\rho\rho' = \sigma$.

2.2 Kripke Structures

Kripke structures are a standard way to model programs semantics [26]. Basically, Kripke structures are graphs where nodes represent states of the system and edges represent transitions between states. Each state may enable several transitions, modeling different execution orders to be determined by a scheduler. State labels equip each state with the relevant information about that state. For technical convenience, our Kripke structures 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 function (valuation) $V(c) : Var \to Val$ 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 (Kripke structure). *A* 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 $\to \subseteq \mathcal{S} \times \mathcal{S}$*. We assume that* $\to$ *is non-blocking, i.e.,* $\forall c \in \mathcal{S}.\, \exists c' \in \mathcal{S}.\, c \to c'$*.*

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 programs. A program C over a variable set Var can be expressed as a Kripke structure $\mathcal{A}^C$ in a standard way: The states of $\mathcal{A}^C$ 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 $c \to c$, i.e., a self-loop, ensuring that $\mathcal{A}^C$ is non-blocking. In the remainder of this paper, we leave out the superscript C whenever this is clear from the context.

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 $i \in \mathbb{N}$, $c_i \to c_{i+1}$. 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 $c \sim_L c'$, iff $V(c)_{|_L} = V(c')_{|_L}$. Over a trace T, we let $T_{|_l}$ and $T_{|_L}$ denote the projections of T on a low variable l and the set of low variables L, respectively.

2.3 Schedulers

A multi-threaded program executes threads from the set of non-terminated threads, i.e., the live threads. During the execution, a scheduling policy repeatedly decides which thread is picked to proceed next. A scheduler is a function that implements a scheduling policy [35]. 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 δ which determines a set of the possible successor states Q is formally defined as follows,

Definition 2. *A scheduler δ for a Kripke structure $\mathcal{A} = \langle \mathcal{S}, I, Var, Val, V, \rightarrow \rangle$ is a function $\delta : Path^*(\mathcal{A}) \rightarrow 2^{\mathcal{S}}$, such that, for all finite paths $\pi \in Path^*(\mathcal{A})$, if $\delta(\pi) = Q \subseteq \mathcal{S}$ then $last(\pi)$ can take a transition to any $c \in Q$. Given π, we write $c_0 \rightarrow c_1 \rightarrow c_2 \ldots \rightarrow c_n$ if $c_i \in \delta(c_0 \ldots c_{i-1})$ for all $1 \leq i < |\pi|$..*

The effect of a scheduler δ on $\mathcal{A}$ can be described by $\mathcal{A}_\delta$: 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 Kripke structure and let δ be a scheduler for $\mathcal{A}$. The Kripke structure 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 $\pi \rightarrow_\delta \pi c$ iff $c \in \delta(\pi)$, i.e., $\mathcal{A}_\delta$ can transition from a path π to a path πc if δ enables scheduling state c after π.*

2.4 Stuttering-free Kripke Structures and Stuttering Equivalence

Stuttering steps and *stuttering equivalence* [32, 24] are the basic ingredients of our confidentiality properties.

Definition 4 (Stuttering-free Kripke structure). *A* stuttering step *is a transition $c \rightarrow c'$ that leaves the labels unchanged, i.e., $V(c') = V(c)$. A Kripke structure is called* stuttering free *if $c \rightarrow c'$ and $V(c) = V(c')$ imply $c = c'$ and c is a final state, i.e., stuttering steps are only allowed as self-loops in final states.*

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 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, or from the ordering of public data updates. This is captured formally by the definition of *scheduler-specific observational determinism*.

As shown in [44, 23], to be secure, a multi-threaded program must impose an order on the accesses to a single low variable, i.e., the sequence of operations performed at a single low variable must be deterministic. Therefore, SSOD's first condition requires that any two traces of each low variable from any two initial low-equivalent state I_1 and I_2 are stuttering equivalent. This condition ensures that no secret information can be derived from the observation of public data traces. Indeed, when each low variable individually evolves deterministically, the values of low variables are independent of the values of high variables. Notice that requiring only the existence of a single matching low variable trace is not sufficient, as shown in the following example,

Example 2. Consider the following program, where `h` is a Boolean,

$$\texttt{if (h) then } \{\texttt{l} := 0; \texttt{l} := 1\} \parallel \texttt{l} := 0 \texttt{ else } \{\texttt{l} := 0; \texttt{l} := 1\} \parallel \{\texttt{l} := 0; \texttt{l} := 0\};$$

This program leaks information under the uniform scheduler: if h is 1, then l is more likely to contain 1 than 0 in final states. However, there always exists a matching low location trace for l. Therefore, we require instead that traces of each low variable must be deterministic. This deterministic property of traces also avoids *cache attacks*, i.e., attacks that exploit the timing behavior of threads via the cache to derive secret information [44].

It should be noted that a consequence of SSOD-1 is that innocuous programs such as `l:=0 ∥ l:=1` are also rejected, since its set of traces cannot be distinguished from the traces of Example 2.

Notice that the transition relation of the Kripke structure is non-blocking, i.e., there is a self-loop at each final state. Thus, the attacker cannot detect termination. However, the termination leaks are avoided, since SSOD requires stuttering equivalence between traces, instead of stuttering and prefixing equivalence as in [44, 39]. Huisman et al. analyzed this point in detail in [24].

SSOD 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'. The second condition of SSOD requires the existence of a matching public data trace. This existential condition avoids

refinement attacks where an attacker chooses an appropriate scheduler to control the set of possible traces. The second condition also ensures that the relative orders of updates of low variables are coincidental. Thus, no information can be deduced from them.

Formally, SSOD is defined as follows,

Definition 6 (SSOD). *For any two initial low-equivalent states I and I', let $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ denote two Kripke structures corresponding to I and I', respectively. Given a scheduler δ, a program C respects SSOD w.r.t. L and δ, iff*

SSOD-1 $\forall T \in \mathit{Trace}(\mathcal{A}_\delta),\ T' \in \mathit{Trace}(\mathcal{A}'_\delta), l \in L.\ T_{|_l} \sim T'_{|_l}$,

SSOD-2 $\forall T \in \mathit{Trace}(\mathcal{A}_\delta).\ \exists T' \in \mathit{Trace}(\mathcal{A}'_\delta).\ T_{|_L} \sim T'_{|_L}$.

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

4 Algorithmic Verification

This section discusses how we algorithmically verify the two conditions of SSOD. As mentioned above, we use a combination of new and existing algorithms. 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 Kripke structures.

We first simplify SSOD by replacing SSOD-1 with SSOD-1A. Intuitively, SSOD-1A requires that, given a Kripke structure $\mathcal{A}$ that corresponds to any initial state, after projecting on l, all traces are stuttering equivalent,

SSOD-1A $\forall l \in L.\ T, T' \in \mathit{Trace}(\mathcal{A}_\delta).\ T_{|_l} \sim T'_{|_l}$.

The new condition SSOD-1A and SSOD-2 are equivalent to SSOD-1 and SSOD-2.

Theorem 1. *If a program is* scheduler-specific observational deterministic *w.r.t. L and a scheduler δ, then SSOD-1 & SSOD-2 $\Leftrightarrow$ SSOD-1A & SSOD-2.*

Proof. 1. SSOD-1 & SSOD-2 $\Rightarrow$ SSOD-1A
Given any two traces $T1$, $T2 \in \mathit{Trace}(\mathcal{A}_\delta)$, and any $T' \in \mathit{Trace}(\mathcal{A}'_\delta)$. According to SSOD-1, $\forall l \in L$, $T1_{|_l} \sim T'_{|_l} \wedge T2_{|_l} \sim T'_{|_l}$. Therefore, we can conclude that $\forall l \in L.\ T1_{|_l} \sim T2_{|_l}$.

2. SSOD-1A & SSOD-2 $\Rightarrow$ SSOD-1
Given any traces $T \in \mathit{Trace}(\mathcal{A}_\delta)$, $T' \in \mathit{Trace}(\mathcal{A}'_\delta)$. According to SSOD-2, there exists a trace $T'' \in \mathit{Trace}(\mathcal{A}'_\delta)$ such that $T_{|_L} \sim T''_{|_L}$. If $T_{|_L} \sim T''_{|_L}$, then $\forall l \in L.\ T_{|_l} \sim T''_{|_l}$. According to SSOD-1A, $\forall l \in L$, $T'_{|_l} \sim T''_{|_l}$, then $\forall l \in L.\ T_{|_l} \sim T'_{|_l}$. □

Let $\mathcal{A}_{\delta\,|_l}$ and $\mathcal{A}_{\delta\,|_L}$ represent the projections of $\mathcal{A}_\delta$ on the label sets l and L, respectively. Since $\{T_{|_l} \mid T \in \mathit{Trace}(\mathcal{A}_\delta)\} = \mathit{Trace}(\mathcal{A}_{\delta\,|_l})$ and $\{T_{|_L} \mid T \in \mathit{Trace}(\mathcal{A}_\delta)\} = \mathit{Trace}(\mathcal{A}_{\delta\,|_L})$, properties SSOD-1A and SSOD-2 can easily be reformulated over these Kripke structures as follows.

SSOD-1K $\forall l \in L.\ T, T' \in \mathit{Trace}(\mathcal{A}_{\delta\,|_l}).\ T \sim T'$,

SSOD-2K $\forall T \in \mathit{Trace}(\mathcal{A}_{\delta\,|_L}).\ \exists T' \in \mathit{Trace}(\mathcal{A}'_{\delta\,|_L}).\ T \sim T'$.

Thus, SSOD-1K requires that for all $l \in L$, all traces of $\mathcal{A}_{\delta\,|_l}$ are stuttering equivalent, while SSOD-2K requires stuttering trace equivalence between $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$.

4.1 Verification of SSOD-1K

Given a program C, and a scheduler δ, SSOD-1K requires that after projecting $\mathcal{A}_\delta$ on any low variable l, all traces must be stuttering equivalent. 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. (Figures 1, 2 on page 14 provide an elaborate illustration of these steps.)

Algorithm 1: SSOD-1K on l

1: Project $\mathcal{A}_\delta$ on l, yielding $\mathcal{A}_{\delta\,|_l}$.
2: Identify all *divergent* states of $\mathcal{A}_{\delta\,|_l}$. A state c of $\mathcal{A}$ is *divergent* if there exists a trace such that all states following c are equivalent to c.
3: Check whether all traces of $\mathcal{A}_{\delta\,|_l}$ are stuttering equivalent by:
  3.1: Choose a *witness* trace by:
    3.1.1: Take an arbitrary lasso T of $\mathcal{A}_{\delta\,|_l}$.
    3.1.2: Remove stuttering steps and minimize T.
  3.2: Check stuttering trace equivalence between $\mathcal{A}_{\delta\,|_l}$ and T by checking if there exists a functional bisimulation between them.

Notice that the following algorithms can be applied to any Kripke structure, i.e., independent of the scheduling policy. Thus, instead of the notation $\mathcal{A}_\delta$ indicating a specific scheduler, we use a general notation $\mathcal{A}$.

Step 1 is done by labeling every state with the value of l in that state.

Step 2 identifies all divergent states of a Kripke structure $\mathcal{A}$. Before describing the algorithm for Step 2, we first derive two lemmas that follow directly from the definition of a divergent state (given in Step 2 of Algorithm 1).

Lemma 1. *If state c has a stuttering loop or a self-loop, c is divergent.*

Lemma 2. *Assume that c has no self-loop or no stuttering loop. Then, state c is divergent iff c has a divergent equivalent successor.*

Step 2 is implemented by exploring state space of a Kripke structure $\mathcal{A}$ and determining whether each reachable state is divergent or not. The state space of $\mathcal{A}$ is explored in a breadth first search order (BFS). Let $\mathit{Pred}(\mathcal{A}, c)$ and $\mathit{Succ}(\mathcal{A}, c)$ denote the set of all direct predecessors and successors of c, respectively, i.e., $\mathit{Pred}(\mathcal{A}, c) = \{b \in \mathcal{S} |\ b \to c\}$ and $\mathit{Succ}(\mathcal{A}, c) = \{d \in \mathcal{S} | c \to d\}$. Let $c \sim_V c'$ denote that c and c' have the same valuation, i.e., $V(c) = V(c')$.

The algorithm to identify divergent states of $\mathcal{A}$ uses two queues: $\mathcal{Q}$ and *Non_Diver_*$\mathcal{Q}$, and two maps: *Divergent* and *Checked*. The queue $\mathcal{Q}$ stores the set of *frontier* states of the exploration. Initially, *Divergent* indicates the number of stuttering transitions of each state, i.e., *Divergent*$[c] = 3$ indicates that c is equivalent to three of its successors. During the execution, the algorithm changes the values in *Divergent*. When the algorithm terminates, the value of *Divergent*$[c]$ will indicate whether c is divergent or not, i.e., iff *Divergent*$[c] = 0$, c is *non-divergent*. The queue *Non_Diver_*$\mathcal{Q}$ stores non-divergent states, i.e., all reachable state c such that *Divergent*$[c] = 0$. The *Checked* indicates that whether a state has been checked or not, i.e., *true* or *false*.

The algorithm works as follows. States of $\mathcal{A}$ are explored by a BFS (lines 6-12 in Algorithm 2). For each explored state c, the number of its direct stuttering transitions is stored in *Divergent*$[c]$ (line 11). If c has no outgoing stuttering transition, i.e., *Divergent*$[c] = 0$, it is clear that c is non-divergent (line 12).

For any c such that *Divergent*$[c] = 0$, for each predecessor b of c such that c is reached from b by a stuttering transition, the algorithm decreases the value *Divergent*$[b]$ by 1 (lines 13-18). The idea is to remove the number of *non-divergent* equivalent successors out of the value *Divergent*$[b]$ of a state b. Finally, when *Divergent* is stable, the value *Divergent*$[b]$ will indicate whether b is divergent or not, i.e., if b has a divergent equivalent successor, i.e., *Divergent*$[b] \neq 0$, b is divergent (due to Lemma 2).

For example, assume that b is a direct predecessor of a non-divergent c, and b has *only* one stuttering transition, which goes to c, i.e., *Divergent*$[b] = 1$. Since c is non-divergent, according to Lemma 2, b is also non-divergent. Due to the algorithm, the value of *Divergent*$[b]$ is decreased by 1; and thus becomes 0, which indicates that b is non-divergent.

Algorithm 2: Identify Divergent States ($\mathcal{A}$)

```
//  Initialization
1.    for all states c ∈ S do
2.          Checked [c]:=false;
3.          Divergent [c]:=0;
4.    Q := empty_queue(); enqueue(Q, init_state);
5.    Non_Diver_Q := empty_queue();
//  Explore state space by BFS
6.    enqueue(Q, init_state); Checked[init_state] := true;
7.    while !empty(Q) do
8.          current := dequeue(Q);
9.          for all states c ∈ Succ(A, current) and ¬Checked[c] do
10.               enqueue(Q, c); Checked[c] := true;
      //  Record the number of stuttering successors
11.         Divergent[current] := |{c ∈ Succ(A, current) | c ~_V current}|;
12.         if Divergent[current] = 0 then enqueue(Non_Diver_Q, current);
//  Propagate non-divergence backwards
13.   while !empty(Non_Diver_Q) do
14.         current := dequeue(Non_Diver_Q);
```

```
15.          for all states b ∈ Pred(A, current) do
16.               if b ∼_V current and Divergent[b] ≠ 0 then
17.                    Divergent[b] := Divergent[b] − 1;
18.                    if Divergent[b] = 0 then enqueue(Non_Diver_Q, b);
//  Normalize divergence
19.      for all states c ∈ S do
20.           if Divergent[c] ≠ 0 then Divergent[c] := true
21.                               else Divergent[c] := false;
```

Theorem 2. *Algorithm 2 identifies divergent states of a Kripke structure $\mathcal{A}$.*

Proof. Algorithm 2 always terminates, since $\mathcal{A}$ is finite. Two queues $\mathcal{Q}$ and *Non_Diver_*$\mathcal{Q}$ ensure that each state and each edge of $\mathcal{A}$ are processed at most once in each *while* loop. Thus, the time complexity of this algorithm is linear in the size of $\mathcal{A}$

We show that the *second while* loop correctly determines the divergent property of each state. We first discuss its loop invariant *Inv*:

If a state c is divergent, $Divergent[c] \neq 0$.

Initially, the value of $Divergent[c]$ is the number of stuttering successors of c. If c is divergent, according to the definition of divergent state, it must have at least one stuttering successor, i.e., $Divergent[c] \neq 0$. Thus, clearly, *Inv* holds upon the first entry of the loop.

We show that the invariant is preserved by every iteration of the loop. Assume *Inv* holds before the loop body. Consider a divergent state c. Due to the invariant, $Divergent[c] \neq 0$ holds before the execution of the loop.

Suppose that the value of $Divergent[c]$ has been decreased by 1 in the iteration, i.e., c has a stuttering successor d with $Divergent[d] = 0$.

- Case c has a self-loop. In that case, c has at least two stuttering successors: one of them is itself. Therefore, the invariant is preserved after the iteration, since $Divergent[c] \geq 2$ before the iteration.
- Case c has a stuttering loop. States in a stuttering loop always have at least one stuttering successor. Since they are connected in a loop, their *Divergent* values never become 0. Since $Divergent[d] = 0$, d must be outside the stuttering loop. Thus, c has at least two stuttering successors before the iteration. The invariant is preserved.
- If c has no self-loop or no stuttering loop, according to Lemma 2, c must have a divergent stuttering successor e. Since the invariant is true before the iteration, $Divergent[e] \neq 0$. Hence, d and e are not the same state. Thus, c has at least two stuttering successors. The invariant is preserved.

Thus, *Inv* is a loop invariant. Therefore, *Inv* holds after termination: if c is divergent, $Divergent[c] \neq 0$. In other words, any c such that $Divergent[c] = 0$ is non-divergent.

Additionally, we show the following *post-condition*: **if the algorithm terminates, and if c is non-divergent, then** $Divergent[c] = 0$.

If c is non-divergent, all traces starting in c must pass a state that is not equivalent to c. Let $\mathcal{C}$ denote the set of equivalent states that are reachable from c only via stuttering transitions. To define $\mathcal{C}$ formally, we define the *stuttering-closure* $Stut(Q)$ of a subset $Q \subseteq \mathcal{S}$,

$$Q_0 = Q,$$
$$\forall n \geq 0.\, Q_{n+1} = \{c' \in \mathcal{S} \mid \exists c \in Q_n.\, c \rightarrow c' \wedge c \sim_V c'\}.$$

We define $Stut(Q) = \cup_n Q_n$. This is formally defined as an infinite union, but it is actually a finite union; since there are at most a finite number of states in $\mathcal{S}$. Therefore, formally, $\mathcal{C} = Stut(c)$.

There must exist a state $c' \in \mathcal{C}$ such that initially, $Divergent[c'] = 0$, i.e., c' only connects to states outside $\mathcal{C}$. Otherwise, a contradiction occurs: assume that initially, $\forall c' \in \mathcal{C}$, $Divergent[c'] \neq 0$, i.e., all states have at least a stuttering successor. Since the set $\mathcal{C}$ is finite, if all states of $\mathcal{C}$ have stuttering successors, there must exist a stuttering loop inside $\mathcal{C}$. Hence, c is divergent due to Lemma 1; and this is a contradiction, since we assume that c is non-divergent.

Let D_0 denote the set of states $c' \in \mathcal{C}$ such that *initially*, $Divergent[c'] = 0$. Notice that the algorithm changes the *Divergent* value of states. Let $\mathbf{D_0}$ denote the set of states of $\mathcal{C}$ such that *currently*, their *Divergent* values are 0. Initially, $\mathbf{D_0} = D_0$.

Consider the set of direct predecessors of D_0. We claim that there must exist a state c'' in this set of predecessors such that initially, $Divergent[c'']$ is equal to the number of its successors c' with $Divergent[c'] = 0$, i.e, $Divergent[c''] = |\{c' \in Succ(\mathcal{A}, c'') \mid c' \sim_V c'' \wedge Divergent[c'] = 0\}|$. If not, the contradiction occurs by the same argument, i.e., there exists a stuttering loop. Let D_1 denote the set of direct predecessors c'' of D_0 such that *initially*, $Divergent[c''] = 1$. According to the algorithm, the *Divergent* values of states in D_1 are decreased by 1. Thus, after a few iterations, the *Divergent* values of all states in D_1 become 0, i.e., $\mathbf{D_0} = D_0 \cup D_1$.

Therefore, by a similar argument, the *Divergent* values of other states, e.g., the predecessors of D_0 with the *Divergent* value 2, also become 0; and so on. The set $\mathbf{D_0}$ gradually grows, and at the termination, $\mathbf{D_0} = \mathcal{C}$. Therefore, when the algorithm terminates, if c is non-divergent, $Divergent[c] = 0$. This is also equivalent that if $Divergent[c] \neq 0$, c is divergent. □

Step 3: Step 3.1.1 is implemented via a classical cycle-detection algorithm based on depth-first search. The initial state of a lasso is also the initial state of the Kripke structure. The algorithm essentially proceeds by picking arbitrary next steps, and terminates when it hits a state that was picked before.

Algorithm 3: Lasso T of $\mathcal{A}$

for all states $c \in \mathcal{S}$ **do** $Visit[c] := false$;
$index := 0$;
$current := init\_state$;
for $(;;)$ **do**

$T[index] := current$; `// Implement` T `as an array`
$index := index + 1$;
if $Visit[current] = true$ **then break**;
$Visit[current] := true$;
$current :=$ some state $c \in Succ(\mathcal{A}, current)$;
return(T, position of $current$ in T);

In this algorithm, we use a map *Visit* to indicate visited states of $\mathcal{A}$, i.e., $Visit[current] = true$ 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 at every final state.

Step 3.1.2 is done via the standard strong bisimulation reduction [9]. 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*. Except the final state, all states of the witness trace are set to be non-divergent.

Step 3.2 checks stuttering trace equivalence between a Kripke structure $\mathcal{A}$ 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 $\mathcal{A}$'s initial state to u_0, i.e., $Map[init\_state] = u_0$ (line 4 in Algorithm 4). Each iteration of the algorithm examines the successors of the state stored in the variable *current* (lines 6-8). Assume that $Map[current]$ is u, consider a successor $c \in Succ(\mathcal{A}, current)$. The *potential_map* of c is u if $current \to c$ is a stuttering transition; otherwise, it is $Succ(T, u)$ (line 9). The algorithm returns *false*, i.e., $continue = false$, if (i) c and *potential_map* have different valuations, (ii) c and *potential_map* have different divergent values, or (iii) c has been checked before, but its mapped state is not *potential_map* (line 10, 11, and 13).

If none of these cases occurs and c was not checked before, c is added to $\mathcal{Q}$, and mapped to *potential_map* (line 12). 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 up to u are stuttering equivalent.

Let $final(\mathcal{A}, c)$ 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 queue $\mathcal{Q}$ of *frontier* states. The termination of the following algorithm follows from the termination of BFS over a finite $\mathcal{A}$.

Algorithm 4: All-Trace Stuttering Equivalence ($\mathcal{A}$, T)

1. **for** all states $c \in \mathcal{S}$ **do** $Map[c] := \perp$;
2. $continue := true$;
3. $\mathcal{Q} := empty\_queue()$; $enqueue(\mathcal{Q}, init\_state)$;
4. $Map[init\_state] := u_0$; `//` u_0 `is` T_0
5. **while** $!empty(\mathcal{Q}) \wedge continue$ **do**
6. $current := dequeue(\mathcal{Q})$;

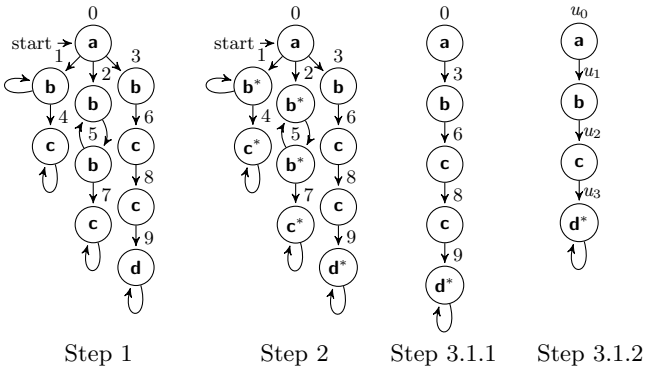

Fig. 1: Step 1 - Step 3.1 of Algorithm 1

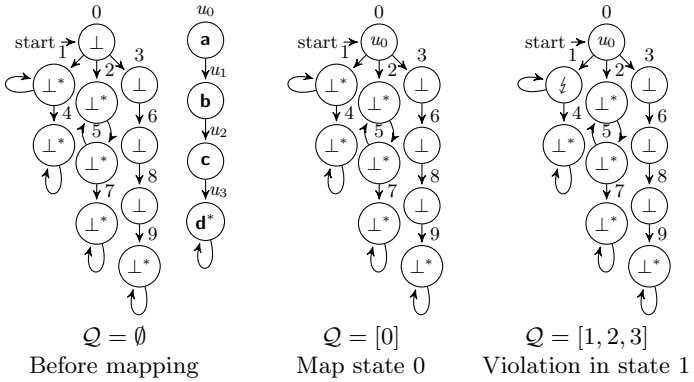

Fig. 2: Step 3.2 of Algorithm 1 (i.e., Algorithm 4)

7. $u := Map[current]$;
8. **for** all states $c \in Succ(\mathcal{A}, current)$ **do**
9. $potential\_map := (c \sim_V current)\ ?\ u : Succ(T, u)$;
10. **case** $c \not\sim_V potential\_map \rightarrow continue := false$;
11. $\parallel Divergent[c] \neq Divergent[potential\_map] \rightarrow continue := false$;
12. $\parallel Map[c] = \bot \rightarrow enqueue(\mathcal{Q}, c);\ \ Map[c] := potential\_map$;
13. $\parallel Map[c] \neq potential\_map \rightarrow continue := false$;
14. **return** *continue*;

Example 3. Figure 1 illustrates Step 1 - Step 3.1 on a Kripke structure $\mathcal{A}$ consisting of 10 states, numbered from 0 to 9. Step 1 shows a projection of $\mathcal{A}$ on a low variable l where 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 identifies divergent states by *. Step 3.1 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 3.2. Initially, all states of $\mathcal{A}$ are mapped to a special symbol $\perp$ that indicates unchecked states. To keep states readable, we skip the valuation. Next, state 0 is enqueued, and mapped to u_0. In the next step, 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. State 1 is divergent while *potential_map* is not, thus, *continue* = *false*. SSOD-1K fails, since there exists a trace that stutters in state 1 forever, and thus, $\mathcal{A}$ and T are not stuttering trace equivalent. The algorithm terminates.

As a first step towards proving correctness, we prove that Algorithm 4 ensures the following loop invariant.

Theorem 3. *Algorithm 4 preserves the following loop invariant Inv:*
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 *up to* 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, *Inv* holds upon the 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 that *Stm* 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 up to u are stuttering equivalent. Now consider a successor c of *current*. We distinguish the following cases:

Case $c \not\sim_V u$ **and** $c \not\sim_V Succ(T, u)$**.** Let *potential_map* denote the mapping candidate 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, since 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** $Divergent[c] \neq Divergent[potential\_map]$**.** If c is a divergent state of $\mathcal{A}$, then there must be a trace that stutters in c forever, while T can evolve from *potential_map* to a state with a different valuation (or vice versa). Thus, these two traces are not stuttering equivalent. Hence, *continue* becomes *false*; and the invariant is preserved.
- **Case** $Divergent[c] = Divergent[potential\_map]$**.**
  - **Case** c **is unchecked.** Thus, $Map[c] = \perp$. State c is added to $\mathcal{Q}$, 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 up to *potential_map* are stuttering equivalent. Hence, the invariant is preserved.
  - **Case** c **is checked before.** Thus, $Map[c] \neq \perp$.

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, since 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. □

Theorem 4. *Algorithm 4 returns true iff there exists a bisimulation between* $\mathcal{A}$ *and* T*.*

Proof. If Algorithm 4 returns *false*, it follows directly from the invariant that no functional bisimulation exists. If it returns *true*, due to the loop invariant, 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 the algorithm maps c to *potential_map*. Since the algorithm is divergent-sensitive, and in T, the only divergent state is the final state, thus *potential_map* is also the final state of T. Therefore, $T1$ and T are stuttering equivalent.

Case $T1$ **ends with a non-stuttering loop that starts and ends in state** c. Thus, state c is investigated twice, and in the second visit, its corresponding mapped state (of T) must be the same as its mapped state in the first visit; otherwise, the algorithm returned *false*. Hence, the c's mapped state is also the start and end of a loop that terminates T. Thus, $T1$ and T are stuttering equivalent. □

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 is based on BFS, thus its time complexity is $O(n + m)$, where m is the number of transitions of $\mathcal{A}$. The time complexity of Step 3.1 to find a witness trace is $O(m)$. The core of Step 3.2 is also BFS, 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 SSOD-1K (for all $l \in L$) is $|L|\ O(n + m)$. If we put restrictions on the initial inputs, i.e., the number of initial states is finite, the verification of SSOD-1K is feasible in practice.

4.2 Verification of SSOD-2K

SSOD-2K requires that, given two Kripke structures $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ that model the executions of a program C from any two initial low-equivalent states I and I', if we project them on the set of low variables L, $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$ are stuttering trace equivalent.

To verify SSOD-2, our algorithm first transforms the Kripke structures into two equivalent ones, without *stuttering steps*, and then *determinizes* them[3]. It is well-known that, for deterministic and stuttering-free Kripke structures, trace equivalence and strong bisimilarity coincide [16]. Therefore, we verify SSOD-2K by combining several existing algorithms.

Algorithm 5: SSOD-2K

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 Kripke structures ${\mathcal{A}^{sf}_{\delta}}_{|_L}$ and ${\mathcal{A}'^{sf}_{\delta}}_{|_L}$.

3: Re-establish self-loops for final states of ${\mathcal{A}^{sf}_{\delta}}_{|_L}$ and ${\mathcal{A}'^{sf}_{\delta}}_{|_L}$.

4: Determinize ${\mathcal{A}^{sf}_{\delta}}_{|_L}$ and ${\mathcal{A}'^{sf}_{\delta}}_{|_L}$, yielding deterministic stuttering-free $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$.

5: Combine $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$, yielding ${\mathcal{R}^{+}_{\delta}}_{|_L}$, and then compute all bisimula--rity equivalence classes of ${\mathcal{R}^{+}_{\delta}}_{|_L}$.

6: Check if I and I' are in the same bisimularity equivalence class.

Step 1 is done by labeling every state of a Kripke structure with the set of low values L in that state. To remove the stuttering steps in **Step 2**, we compute the stuttering closure of each state, using the standard all-pair shortest path algorithm, and then collapse these components into a single state. To ensure that the transition relation remains non-blocking, **Step 3** re-establishes self-loops for final states. Notice that $\mathcal{A}$ and $\mathcal{A}^{sf}$ are stuttering trace equivalent, since traces of $\mathcal{A}^{sf}$ are traces of $\mathcal{A}$, but all stuttering steps in traces have been removed.

The determinization of a Kripke structure in **Step 4** is obtained via the well-known subset construction. Due to the property of determinization of finite automata, $\mathcal{A}^{sf}$ and $\mathcal{R}$ are trace equivalent. Notice that the determinization is based on *low events*, i.e., operations of changing the values of low variables. Thus, a state of $\mathcal{R}$ is a group of states in the original Kripke structure $\mathcal{A}$ that are reached via the same low operation. Therefore, states of $\mathcal{R}$ have the same label with their inside component states.

Step 5: Computing bisimularity equivalence classes. For deterministic stuttering-free Kripke structures, trace equivalence and strong bisimilarity coincide. To verify strong bisimilarity of $\mathcal{R}$ and $\mathcal{R}'$, we first take the union of state spaces of the two Kripke structures, denoted $\mathcal{R}^+$, and use the classic algorithm for computing bisimilarity by Paige and Tarjan [31]. This is a standard algorithm, and readers can refer to [31] for a detailed description of the algorithm. However, since this step strongly relates to the next section where we synthesize attacks for insecure programs, we represent briefly the algorithm's main idea.

The algorithm for computing bisimularity equivalence classes exploits the well-known partition-refinement technique [31]. The main idea of this technique

[3] We refer to the determinism concept in automata theory [21].

is to partition the state space into disjoint blocks of states, and to repeatedly refine this partition: whenever we find that states of a block are not equivalent, we split the block into separate blocks. If the partition is stable, i.e., it is not necessary to refine it more, we terminate.

A *partition* $\mathcal{P}$ of S is a collection $\{Q_i\}_{i\in\mathcal{I}}$ of nonempty subsets of S such that $\bigcup_i Q_{i\in\mathcal{I}} = S$ and for any $i' \neq i : Q_{i'} \cap Q_i = \emptyset$. The elements of a partition are called *blocks*. Given a partition $\mathcal{P}$, we say that another partition $\mathcal{P}'$ *refines* $\mathcal{P}$, if any block of $\mathcal{P}'$ is included in a block of $\mathcal{P}$. We define $\mathcal{P}$-equivalence as follows: $c \sim_{\mathcal{P}} c' \Leftrightarrow \exists Q \in \mathcal{P}.\, c \in Q \wedge c' \in Q$, i.e., intuitively, c and c' are in the same block.

The initial partition $\mathcal{P}_0$ is constructed by categorizing states with the same valuation into blocks, i.e., $c \sim_{\mathcal{P}_0} c' \Leftrightarrow V(c) = V(c')$. In the refinement step, we split a block Q into two disjoint subblocks, one collects all states being able to reach another block Q', while the other collects all states that cannot reach Q'. In this case, we call Q' a *splitter* of Q. Partition $\mathcal{P}$ is *stable* w.r.t. a block Q' if there is no block $Q \in \mathcal{P}$ such that Q' is a splitter of Q. $\mathcal{P}$ is *stable* if it is stable w.r.t. all its blocks.

Step 6: Inclusion check. Finally, we check if two initial states I and I' are in the same bisimularity equivalence class. This will indirectly answer whether $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$ are bisimilar, i.e., they are bisimilar if two initial states fall into the same block, otherwise they are not.

Overall complexity. The stuttering closures in Step 2 can be computed in $O(n^3)$ using the all-pair shortest path algorithm. However, improved algorithms for doing so run in $O(n^{2.376})$ [14].

The algorithm by Paige and Tarjan computes the partition corresponding to strong bisimilarity in $O(m \cdot \log n)$ [31]. Thus, the complexity of verifying SSOD-2K is dominated by the determinization in Step 4, which is exponential in the number of states. However, the worse case complexity is often reached in only the extreme cases. Therefore, we believe that the verification of SSOD-2K is feasible in practice.

5 Attack Synthesis for SSOD

Counter example generation is a powerful technique in model checking, with many applications, such as diagnosis, scheduler synthesis, and debugging [13, 11, 20, 5]. This section presents counter example generation techniques for attack synthesis. That is, if for a given scheduler, a program does not satisfy the confidentiality requirements, we generate program traces that reveal the reason why the confidentiality is broken.

5.1 Attack Synthesis for SSOD-1K

We propose to extend Algorithm 4 that checks SSOD-1K for counter example generation, i.e., Algorithm 6 returns a trace that is not stuttering equivalent to

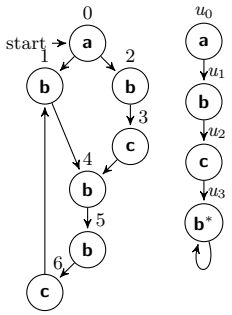

Fig. 3: Abnormal state

the witness trace when the Kripke structure does not satisfy SSOD-1K. These two algorithms are similar. However, when a state that violates SSOD-1K is found, Algorithm 6 does not terminate, as Algorithm 4 does, but continues until it finds an unmatched trace. The unmatched trace is returned via Algorithm 8: **Trace Return** given below.

State c is denoted as an *unmatched* state if it is not equivalent to its corresponding state *potential_map*. State c is also *unmatched* if c is equivalent to *potential_map*, but c is divergent while *potential_map* is not. Whenever an *unmatched* state is found, SSOD-1K is not satisfied. A counter example trace is the trace from the initial state to the unmatched state. Since the search is based on BFS, the trace is the shortest path from the initial state to the unmatched state (line 11 in Algorithm 6).

Notice that if c is equivalent to *potential_map*, and c is *not divergent* but *potential_map* is, it is clear that $\mathcal{A}$ does not satisfy SSOD-1K. However, the trace from I up to c is not a counter example, since it is still stuttering equivalent to the witness trace T. Hence, in this case, state c is treated as a normal state, i.e., if c is unchecked, it is still assigned the label *potential_map*, and the algorithm continues until a real counter example is found (lines 13-15).

Consider a situation that the check hits a state c that has been checked before, , if $c \sim_V$ *potential_map* but $Map[c] \neq$ *potential_map*, we consider c is an *abnormal checked* state, denoted *abnormal* (lines 16-18). It means that there exist at least two traces that both lead to *abnormal*; and *abnormal* corresponds to two different states of T. Notice that one of them hits *abnormal* via *current*, thus we denote *current* as *pre_abnormal*.

Figure 3 explains this situation. According to the algorithm, state 1 and state 2 are both mapped to u_1. Next, state 4 is also mapped to u_1, since it follows state 1 via a stuttering step. In this check, we store *parent*[state 4] = state 1. Next, the algorithm examines the successor of state 2, i.e., state 3, and maps state 3 to u_2; since this is a non-stuttering transition, and state 3 is equivalent to u_2.

When the algorithm examines the successor of state 3, which is state 4, the *potential_map* is u_3; since this is a non-stuttering transition. However, state 4 has been explored before, and its current assigned label is u_1. Since the current label of state 4 is different from *potential_map*, but state 4 is still equivalent to *potential_map*, then state 4 is an *abnormal checked* state. Notice that if state 4 and *potential_map* are *not* equivalent, state 4 is an unmatched state. State 3 is the *pre_abnormal* state, since in this check, state 4 is its successor. However, *parent*[state 4] still indicates that state 1 is the parent of state 4, i.e., referring to the step of checking the successor of state 1.

When an *abnormal* is found, SSOD-1K is not satisfied. However, neither of the two traces from *init_state* up to *abnormal*, e.g., **abb** and **abcb** in Figure 3, is a complete counter example. Actually, one of them is a prefix of a real counter example. Therefore, the function **Trace Return** returns two traces: P_1 that is from *init_state* to *parent*[*abnormal*], e.g., $P_1 = \mathbf{ab}$, and P_2 that is from *init_state* to *pre_abnormal*, e.g., $P_2 = \mathbf{abc}$. The current BFS check is terminated. The new check starts from *abnormal* (given by Algorithm 9), in which P_1 is extended to a lasso. In this new check, if an unmatched state is found, we are done; otherwise, a complete lasso $T1$ is obtained, i.e., $T1 = P_1 + P_3$. In Figure 3, $T1 = \mathbf{a}(\mathbf{bbbc})^\omega$. If $T1$ is *not* stuttering equivalent to the witness trace T, it is a counter example; otherwise, the counter example is the lasso $T2$ that is an extension of $P_2 + P_3$. The reason is that three traces T, $T1$, and $T2$ cannot be all stuttering equivalent to each other. In Figure 3, $T1$ is the counter example.

If no counter example is found after all reachable states from the initial state have been explored, SSOD-1K is satisfied.

Algorithm 6: Attack Synthesis for SSOD-1K ($\mathcal{A}$, T)

```
1.   for all states c ∈ S do Map[c] := ⊥;
2.   continue := true;
3.   Q := empty_queue(); enqueue(Q, init_state);
4.   parent[init_state] := root;      // to indicate the start of traces
5.   Map[init_state] := u_0;      // u_0 is T_0
6.   while !empty(Q) ∧ continue do
7.        current := dequeue(Q);
8.        u := Map[current];
9.        for all states c ∈ Succ(A, current) do
10.            potential_map := (c ∼_V current) ? u : Succ(T, u);
11.            Unmatched-State Check (c, potential_map, current)
12.            case
13.                ‖ Map[c] = ⊥ → enqueue(Q, c);
14.                               parent[c] := current;
15.                               Map[c] := potential_map;
16.                ‖ Map[c] ≠ potential_map →
17.                        continue := false;
18.                        Abnormal-State Check (current, c);
19.  return continue;
```

The function **Unmatched-State Check** $(c, potential\_map, current)$ returns a counter example if c is an unmatched state.

Algorithm 7: Unmatched-State Check $(c,\ potential\_map, current)$

if $c \not\sim_V potential\_map$ **or**
$Divergent[c] = true \wedge Divergent[potential\_map] = false$ **then**
$parent[c] := current$;
$continue := false$;
Counter Example = **Trace Return**$(init\_state, c)$;

The function **Trace Return** (a, b) returns a finite trace from state a to state b. Notice that the stack *trace* stores the trace backwards, i.e., a is the last element that enters the stack.

Algorithm 8: Trace Return (a, b)

$trace := empty\_stack()$; // $trace$ is a stack
PUSH $(trace, b)$; // Add an item to the stack
$parent\_value := parent[b]$;
while $parent\_value \neq parent[a]$ **do**
PUSH$(trace, parent_value)$;
$parent\_value := parent[parent\_value]$;
while !$empty(trace)$ **do**
return POP $(trace)$; // Remove an item from the top of the stack

---

The function **Abnormal-State Check** $(pre\_abnormal, abnormal)$ returns a counter example when an *abnormal* state is found.

Algorithm 9: Abnormal-State Check $(pre\_abnormal, abnormal)$

Let P_1 = **Trace Return**$(init\_state, parent[abnormal])$;
Let P_2 = **Trace Return**$(init\_state, pre\_abnormal)$;
// Except states on P_1, erase the labels and parents of other states
for all states $c \in \mathcal{S} \wedge c \notin P_1$ **do**
$Map[c] := \perp$;
$parent[c] := \perp$;
// The new check starts from *abnormal*
$current := abnormal$;
$u := Map[abnormal]$;
$c :=$ some state $\in Succ(\mathcal{A}, current)$;
// Extend P_1 to a lasso
while $Map[c] = \perp$ **do**
$potential\_map := (c \sim_V current)\ ?\ u : Succ(T, u)$;
// if an unmatched state is found, we are done
Unmatched-State Check $(c, potential\_map, current)$;
$parent[c] := current$;
$Map[c] := potential\_map$;
$current := c$;

$u :=$ *potential_map*
$c :=$ some state $\in Succ(\mathcal{A}, current)$
// Return T_1 if it is not stuttering equivalent to T
if $Map[c] \neq$ *potential_map* **then**
Counter Example = P_1 + **Trace Return**$(abnormal, c)$;
// Return T_2 if T_1 is stuttering equivalent to T
if $Map[c] =$ *potential_map* **then**
if $c \in P_1$ **then**
Counter Example = P_2 + **Trace Return**$(abnormal, c)$ +
Trace Return$(c, abnormal)$;
else Counter Example = P_2 + **Trace Return**$(abnormal, c)$;

Notice that in case $c \in P_1$, P_2 + **Trace Return**$(abnormal, c)$ is not a complete lasso (see Figure 3 for a visual example), a counter example should be P_2 + **Trace Return**$(abnormal, c)$ + **Trace Return**$(c, abnormal)$.

Theorem 5. *In case a Kripke structure $\mathcal{A}$ does not satisfy SSOD-1K, Algorithm 6 returns a trace that is not stuttering equivalent to the witness trace T.*

Proof. Algorithm 6 is a variant of Algorithm 4 whose correctness has been proved. Here we show that the algorithm produces a correct counter example.

Case $c \not\sim_V$ *potential_map***.** It is clear that the trace from *init_state* to c is an counter example.

Case $Divergent[c] =$ *true* **and** $Divergent[$*potential_map*$] =$ *false***.** Since c is a divergent state, there exists a trace that goes from *init_state* to c, and then stutters in c forever. State *potential_map* is not divergent, thus, it is not the final state of T. Since T is stuttering-free, it can evolve to a state that is not equivalent to *potential_map*. Therefore, the trace from *init_state* to c is an counter example.

Otherwise,. Let $P_1 =$ **Trace Return** $(init\_state, parent[abnormal])$, and $P_2 =$ **Trace Return** $(init\_state, pre\_abnormal)$. Let $T1$ denote a lasso that is an extension of P_1 from *abnormal*, i.e., $T1 = P_1 + P_3$, where P_3 is the extension. While extending P_1, if an unmatched state is found, we are done. Otherwise, when the lasso T_1 is complete, i.e., when $Map[c] \neq \perp$, one of the two following scenarios occurs.

Case $Map[c] \neq$ *potential_map***.** $T1$ is not stuttering equivalent to T, since c corresponds to two different states of T.

Case $Map[c] =$ *potential_map***.** It is clear that T_1 is stuttering equivalent to a prefix of T, i.e., the prefix up to *potential_map*. We show that $T1$ is actually stuttering equivalent to the whole T. Given that c has been checked before, then c is the start and end of a loop that terminates the lasso T_1.

Case c **is not divergent.** Since c is mapped to *potential_map* twice, *potential_map* is also the start and end of a loop that terminates T. Thus $T1 \sim T$.

Case c is a divergent state. Thus, *potential_map* is also divergent.
Therefore, *potential_map* is the final state of T; then $T1 \sim T$.

Let T_2 denote an extension of $P_2 + P_3$ to a lasso. Since T is deterministic stuttering-free, and *abnormal* is mapped to two different states of T, $T1$ and $T2$ cannot be both stuttering equivalent to T. Since $T1 \sim T$, T_2 is the counter example. □

5.2 Attack Synthesis for SSOD-2K

Given two deterministic stuttering-free Kripke structures $\mathcal{R}$ and $\mathcal{R}'$, if $\mathcal{R}$ and $\mathcal{R}'$ do not satisfy SSOD-2K, Algorithm 10 outputs a trace of $\mathcal{R}$ that does not match with any trace of $\mathcal{R}'$, or vice versa.

It is clear that for $\mathcal{R}$ and $\mathcal{R}'$, any two strongly bisimilar states must be in the same block. We denote a couple of states of $\mathcal{R}$ and $\mathcal{R}'$ to be *valid* if they have the same label, but they are not in the same block of the stable partition $\mathcal{P}$ given by the algorithm that computes bisimularity equivalence classes in Section 4.2. Given a *valid* couple (c, c') ($c \in \mathcal{S}_{\mathcal{R}}$, $c' \in \mathcal{S}_{\mathcal{R}'}$, where $\mathcal{S}_{\mathcal{R}}$, $\mathcal{S}_{\mathcal{R}'}$ are state sets of $\mathcal{R}$ and $\mathcal{R}'$, respectively), similarly, two successors of (c, c') are also *valid* if they have the same label, but they are not strongly bisimilar. Therefore, from an initial state, a counter example trace must pass states in a sequence of valid couples until it hits a state such that the other trace (from the other initial state) cannot find a successor to form a *valid* couple. The shortest counter example trace is found based on BFS.

Given two valid states $c \in \mathcal{S}_{\mathcal{R}}$ and $c' \in \mathcal{S}_{\mathcal{R}'}$. Let **SameBlock**$(c, c')$ be a predicate to check whether c and c' are in the same block of the given partition, i.e., **SameBlock** (c, c') {**return** $(\exists Q \in \mathcal{P}.\, c \in Q \wedge c' \in Q)$}. We formally define **Valid** (c, c') {**return** $(c \in \mathcal{S}_{\mathcal{R}} \wedge c' \in \mathcal{S}_{\mathcal{R}'} \wedge V(c) = V(c') \wedge \neg$**SameBlock** (c, c')}, and the set *Valid_Succ*(c, c') of valid successors of (c, c') as follows,

$$\mathit{Valid\_Succ}(c, c') = \{(d, d') \mid c \in \mathcal{S}_{\mathcal{R}}, c' \in \mathcal{S}_{\mathcal{R}'}.\, c \to d, c' \to d' \wedge \textbf{Valid } (d, d')\}.$$

We also define a predicate **Unmatched Succ** on a valid couple (c, c') to (1) check whether either c or c' can take a transition to a state d such that none of the other state's successors is equivalent to d, and to (2) return d, if d exists. Formally,

Unmatched Succ (c, c') {
  if ($c \to d \wedge \nexists d' \in \mathcal{S}_{\mathcal{R}'}.\ c' \to d' \wedge V(c') = V(c)$) **then return** (d)
  if ($c' \to d' \wedge \nexists d \in \mathcal{S}_{\mathcal{R}}.\ c \to d \wedge V(c) = V(c')$) **then return** (d') }

Therefore, given the stable partition $\mathcal{P}$, the following algorithm explores the state spaces and returns the first state given by **Unmatched Succ**.

Algorithm 10: Invalid-State Search ($\mathcal{P}$)

$\mathcal{Q}_{\mathcal{R}}[0]$:= initial state of $\mathcal{R}$;
$\mathit{parent}_{\mathcal{R}}[\mathcal{Q}_{\mathcal{R}}[0]] := \mathit{root}$;
$\mathcal{Q}_{\mathcal{R}'}[0]$:= initial state of $\mathcal{R}'$;

$parent_{\mathcal{R}'}[\mathcal{Q}_{\mathcal{R}'}[0]] := root$;
$i := 0$; // i: position of the latest items in both $\mathcal{Q}_{\mathcal{R}}$ and $\mathcal{Q}_{\mathcal{R}'}$
while (true) **do**
  Unmatched Succ $(\mathcal{Q}_{\mathcal{R}}[i], \mathcal{Q}_{\mathcal{R}'}[i])$;
  for each $(c, c') \in Valid\_Succ(\mathcal{Q}_{\mathcal{R}'}[i], \mathcal{Q}_{\mathcal{R}'}[i])$ **do**
    $\mathcal{Q}_{\mathcal{R}}[i+1] := c$;
    $parent_{\mathcal{R}}[\mathcal{Q}_{\mathcal{R}}[i+1]] := \mathcal{Q}_{\mathcal{R}}[i]$;
    $\mathcal{Q}_{\mathcal{R}'}[i+1] := c'$;
    $parent_{\mathcal{R}'}[\mathcal{Q}_{\mathcal{R}'}[i+1]] := \mathcal{Q}_{\mathcal{R}'}[i]$;
  $i := i + 1$;

Then **Parent Return** outputs the trace from the initial state up to the state returned by the algorithm.

Theorem 6. *In case $\mathcal{R}$ and $\mathcal{R}'$ do not satisfy SSOD-2K, Algorithm 10 returns a trace of $\mathcal{R}$ that does not match with any trace of $\mathcal{R}'$, or vice versa.*

Proof. Two queues $\mathcal{Q}_{\mathcal{R}}$ and $\mathcal{Q}_{\mathcal{R}'}$ store couples of valid successors reachable from two initial states. When the first unmatched state is found, the trace from the initial state to this state is returned. Due to BFS, this trace is one of the shortest paths from the initial state to it.

Notice that in case no state is returned by **Unmatched Succ** $(\mathcal{Q}_{\mathcal{R}}[i], \mathcal{Q}_{\mathcal{R}'}[i])$, for each successor of $\mathcal{Q}_{\mathcal{R}}[i]$, there must exist a successor of $\mathcal{Q}_{\mathcal{R}'}[i]$ such that these two states have the same label, or vice versa. If none of these couples is valid, i.e., two states of any couple are bisimilar, then $\mathcal{Q}_{\mathcal{R}}[i]$ and $\mathcal{Q}_{\mathcal{R}'}[i]$ are also bisimilar. This is a contradiction, since $\mathcal{Q}_{\mathcal{R}}[i]$ and $\mathcal{Q}_{\mathcal{R}'}[i]$ are a valid couple. Thus, the set $Valid\_Succ(\mathcal{Q}_{\mathcal{R}}[i], \mathcal{Q}_{\mathcal{R}'}[i])$ is not empty; and the algorithm continues until an unmatched state is found. □

Now, we have to derive the original traces of $\mathcal{A}$ that correspond to the trace of $\mathcal{R}$ given by Algorithm 10; since these original traces are the real counter examples.

Derive the original traces of $\mathcal{A}$ from a trace of $\mathcal{R}$. Suppose that $\mathcal{R}$ is the corresponding deterministic Kripke structure of the stuttering-free $\mathcal{A}^{sf}$. We define a relation Re between a transition of $\mathcal{A}^{sf}$ and a transition of $\mathcal{R}$, i.e., $Re \subseteq \rightarrow_{\mathcal{A}^{sf}} \times \rightarrow_{\mathcal{R}}$ as follows. Let trn denote a transition, $source(trn)$ and $dest(trn)$ the source and the destination state of trn.

$$\forall trn \in \rightarrow_{\mathcal{A}^{sf}}, trn' \in \rightarrow_{\mathcal{R}} .\ trn\ Re\ trn' \Leftrightarrow V(source(trn)) = V(source(trn')) \wedge V(dest(trn)) = V(dest(trn')).$$

Given a trace T of $\mathcal{R}$, the following algorithm derives a set $Trace(T) = \{T'\}$ of $\mathcal{A}^{sf}$ corresponding to T. Let N denote the number of transitions in T. We rewrite T as an array of N transitions, i.e., $T = T[0], T[1], \ldots, T[N-1]$, where

$T[i] = (T_i, T_{i+1})$. We implement an array *States* of size $N+1$, where each element $States[i]$ is a set of states of $\mathcal{A}^{sf}$, i.e., $States[i] \subseteq \mathcal{S}_{\mathcal{A}^{sf}}$.

Algorithm 11: Trace Map (T)

$States[0]$:= initial state of $\mathcal{A}^{sf}$;
for $i := 1$ to N **do**
  $States[i] := \{c' \in \mathcal{S}_{\mathcal{A}^{sf}} \mid \exists c \in States[i-1].\ c \rightarrow_{\mathcal{A}^{sf}} c' \wedge (c, c')\ Re\ T[i-1]\}$;
for each $c \in States[N]$ **do**
  $current := c$;
  for $i := N - 1$ to 0 **do**
    Take $c' \in States[i] \wedge (c', current)\ Re\ T[i]$;
    $T'[i] := (c', current)$;
    $current := c'$;

Traces of $\mathcal{A}$ can be derived easily, since we assumed that states are numbered and the transition relation between states is accessible.

6 Related Work

The idea of observational determinism originates from the notion of noninterference, which only considers the leakages in the final states of program traces. We refer to [34, 24] for a more detailed description of noninterference, its verification, and a discussion why it is not appropriate for multi-threaded programs.

Roscoe [33] was the first to state the importance of determinism to ensure secure information flow of multi-threaded programs. Intuitively, observational determinism expresses that a multi-threaded program is secure when its *publicly observable traces* are independent of its confidential data. In the literature, many formal definitions have been proposed [44, 24, 39], but none of these earlier definitions captures exactly this intuition.

The first formal definition of observational determinism was proposed by Zdancewic and Myers [44]. It states that a program is observationally deterministic iff given any two initial low-equivalent states, any two traces of each low variable are equivalent up to stuttering and prefixing. Zdancewic and Myers only consider traces of each single low variable. They also argue that prefixing is a sufficiently strong relation, as this only causes external termination leaks of one bit of information [44]. In 2006, Huisman, Worah and Sunesen showed that allowing termination leaks might reveal more than one bit of information. Thus, they strengthened the definition of observational determinism by requiring that traces of each low variable must be stuttering equivalent [24]. In 2008, Terauchi showed that an attacker might derive secret information by observing the relative order of low variable updates [39]. Therefore, he proposed another variant of observational determinism, requiring that all traces should be equivalent up to stuttering and prefixing w.r.t. all low variables. The main difference between this definition and the two previous ones is that instead of dealing with each low variable separately, this one considers all of them together. However, Terauchi's

definition still accepts programs that reveal secret information. Moreover, it rejects too many innocuous programs, since it requires the complete set of low variables to evolve in a deterministic way [23].

Besides, these definitions of observational determinism claim that they are scheduler-independent. However, in [23], we show that this claim is not correct. SSOD is the only one to consider the effect of schedulers on confidentiality. In [23], we also discuss the properties of SSOD, and claim that SSOD approximates the intuitive notion of security more precisely than the earlier definitions of observational determinism, which either accept insecure programs, or are overly restrictive. In [23], we also propose a definition of scheduler-independent observational determinism. We show that given the uniform scheduler, for any two initial low-equivalent states I and I', if *all possible* traces starting in I and I' are stuttering equivalent w.r.t. all low variables, this program is secure under any scheduling policy.

Mantel et al. [28] 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.

SSOD is a *possibilistic* secure information flow property: it only considers the nondeterminism that is possible in an execution, but it does not consider the probability that an execution will happen. When a scheduler's behavior is *probabilistic*, some threads might be executed more often than others, which opens up the possibility of a probabilistic attack. To prevent information leakage under probabilistic attacks, several notions of probabilistic noninterference have been proposed, e.g., by Volpano et al., Sabelfeld and Sands, and Smith [42, 35, 37]. However, in [30], we show that these definitions have limitations. We introduce the notion of *scheduler-specific probabilistic observational determinism* (SSPOD), together with an algorithmic technique to verify it. 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 low-equivalent states I and I', 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. This definition extends SSOD, and makes it usable in a larger context.

Sabelfeld and Sands's definition of probabilistic noninterference also takes into account the role of schedulers on confidentiality [35]. This definition is based on a *probabilistic low-bisimulation*, which requires that given any two initial low-equivalent states, 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 too 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 our definitions, both SSOD and SSPOD, accept, such as `if` $(\mathtt{h} > 0)$ `then` $\{\mathtt{l1} := 3; \mathtt{l1} := 3; \mathtt{l2} := 4\}$ `else` $\{\mathtt{l1} := 3; \mathtt{l2} := 4\}$.

To overcome the restriction on timing, Smith proposes to use a *weak* probabilistic bisimulation [37]. 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 probabilistic condition of bisimulation is more restrictive than SSPOD.

Notice that 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 [23].

Palamidessi et al., Chen et al., Smith, and Zhu et al. [2–4, 12, 38, 45], and also we [29] 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 this work.

To verify confidentiality, Zdancewic and Myers, Sabelfeld and Sands, Terauchi, Smith, and Kobayashi [44, 35, 39, 37, 25] use type systems. As discussed in [23], type systems are not suited to verify existential properties, as the one in SSOD and 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., $\mathtt{h} := \mathtt{l}; \mathtt{l} := \mathtt{h}$.

Recently, dynamic monitoring is emerging as an approach to gain better precision than type systems [27, 36, 43, 19]. This approach follows the precise control flow of a program, and thus, calculation of control dependences can be more accurate. However, this approach is often runtime overhead.

Huisman et al. use a different approach [24] based on self-composition [7, 15], and in particular, on the temporal logic characterization of non-interference by Barthe et al. Developing a temporal logic characterization allows to use a standard model checker to verify the information flow property. Huisman et al. characterize observational determinism in CTL*, using a special *non-standard* synchronous composition operator, and also in the polyadic modal μ-calculus (a variation of the modal μ-calculus) [24]. In an attempt to make the result more generally applicable, Huisman and Ngo [23, 22] characterize stuttering equivalence as a conjunction of an LTL and a CTL formula. However, the result is still a complicated formula, where states have to maintain a queue of the *difference in changes* between two threads, and actually using a model checker to verify this is not realistic. Therefore, in this paper, instead of giving a temporal logic characterization, we propose to use algorithmic techniques. This has an additional advantage that we can use the negative results of the algorithms to generate attacks. It should be also stressed that our algorithmic verification techniques could easily be adapted to verify other formalizations of observational

determinism, including our scheduler-independent definition in [23], since they are all based on stuttering equivalence.

Giffhorn et al. [17] propose a verification method based on program dependence graphs to check observational determinism. Program dependence graph models information flow through a program where nodes are program statements, and edges are data dependences or control dependences. Their definition of observational determinism is similar to our definition of scheduler-independent observational determinism [23]. However, in their approach, there is no global security classification of variables, i.e., a variable at one program point may contain a low value, but at another point a high value. Thus, their trace definition is based on low operations, i.e., read or write on a low variable, instead of low values as in the traditional approaches. This work also allows prefixing if traces are finite. This verification method has a rather high time complexity, i.e., $O(n^3)$, while our algorithm to check all-trace stuttering equivalence has a linear time complexity in the size of the graph modeling the program.

Alur et al. [1] enrich the traditional tree model with labeled edges that capture observational indistinguishability between nodes. This enriched model is expressive enough to specify information flow properties in temporal logics. In later work, Černý and Alur [41] then consider a weaker class of properties, so-called conditional confidentiality properties, and develop a sound automated analysis for this, based on a combination of under- and over-approximations. The practical impact of this work is illustrated by applying it on Java2ME. However, their properties are for sequential programs, i.e., they do not consider multiple threads. Finally, Van der Meyden and Zhang [40] develop algorithmic verification techniques on state-based models for a number of different *noninterference* notions, and characterize the computational complexity of the associated verification problems.

7 Conclusion

Summary. This paper discusses the efficient verification of scheduler-specific observational determinism. This formalization captures the intuitive idea of observational determinism more precisely than other formalizations in the literature, i.e., a program is accepted by SSOD, no secret information can be derived from the publicly observable traces and the relative order of updates of low variables. The verification uses a combination of new and existing algorithms. The new algorithm solves a standard problem, i.e., checking all-trace stuttering equivalence and stuttering trace equivalence of Kripke structures, which makes them applicable also in a broader context. The advantage of using model checking algorithms is that they can generate counter examples when the verification fails. We extend our algorithms for this purpose, i.e., presenting counter examples to synthesize information leaking attacks.

Future work. The implementation of our verification algorithms has been done in the symbolic model checker LTSmin [10]. Currently, we are applying the tool to case studies.

As future work, we would like to understand if the verification algorithms can be further optimized for particular classes of schedulers, e.g., for all round robin schedulers. We also would like to run the attack synthesis for scheduler, i.e., find a set of schedulers that might break the confidentiality of a given program.

Finally, we believe that the same approach of adapting existing model checking algorithms will also be appropriate to efficiently and precisely verify other security properties, such as integrity and availability.

Acknowledgment: Our work is supported by the Netherlands Organization for Scientific Research under grant 612.067.802 (SLALOM) and grant Dn 63-257 (ROCKS). In addition, the authors would like to thank Stefan Blom for many fruitful discussions and the anonymous reviewers for useful feedback of an earlier version of this paper.

References

1. R. Alur, P. Černy, and S. Chaudhuri. Model checking on trees with path equivalences. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 664–678. Springer-Verlag, 2007.
2. M. S. Alvim, M. E. Andrés, K. Chatzikokolakis, and C. Palamidessi. On the relation between differential privacy and quantitative information flow. In *Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II*, ICALP'11, pages 60–76. Springer-Verlag, 2011.
3. M.S. Alvim, M.E. Andrés, K. Chatzikokolakis, and C. Palamidessi. Foundations of security analysis and design vi. chapter Quantitative information flow and applications to differential privacy, pages 211–230. Springer-Verlag, 2011.
4. M. E. Andrés, C. Palamidessi, P. van Rossum, and A. Sokolova. Information hiding in probabilistic concurrent systems. *Theor. Comput. Sci.*, 412(28):3072–3089, 2011.
5. M.E. Andrés, P. D'Argenio, and P. Rossum. Significant diagnostic counterexamples in probabilistic model checking. In *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'08, pages 129–148. Springer-Verlag, 2009.
6. C. Baier and J.P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
7. G. Barthe, P.R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW'04, pages 100–. IEEE Computer Society, 2004.
8. G. Barthe and L.P. Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE'04, pages 13–22. ACM, 2004.
9. S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *Int. J. Softw. Tools Technol. Transf.*, 7:74–86, 2005.
10. S. Blom, J. van de Pol, and M. Weber. LTSmin: distributed and symbolic reachability. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 354–359. Springer-Verlag, 2010.
11. M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. *Int. J. Softw. Tools Technol. Transf.*, 9:429–445, 2007.

12. H. Chen and P. Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS'07, pages 31–40. ACM, 2007.
13. E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC'95, pages 427–432. ACM, 1995.
14. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC'87, pages 1–6. ACM, 1987.
15. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05, pages 193–209. Springer-Verlag, 2005.
16. W. Du and Y. Deng. A quasi-local algorithm for checking bisimilarity. In *Proceedings of the 2011 IEEE International Conference on Computer Science and Automation Engineering*, volume 2 of *CSAE'11*, pages 1–5, 2011.
17. D. Giffhorn and G. Snelting. Probabilistic noninterference based on program dependence graphs. Technical report, Karlsruhe Institute of Technology, 2012.
18. J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
19. G. Le Guernic. Precise dynamic verification of confidentiality. In *Proceedings of the 5th International Verification Workshop*, 2008.
20. T. Han, J.P. Katoen, and D. Berteun. Counterexample generation in probabilistic model checking. *IEEE Trans. Softw. Eng.*, pages 241–257, March 2009.
21. J.E. Hopcroft and J.D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1990.
22. M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. Technical Report TR-CTIT-11-22, CTIT, University of Twente, Netherlands, 2011.
23. M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In *Proceedings of the 2011 international conference on Formal Verification of Object-Oriented Software*, FoVeOOS'11, pages 178–195. Springer-Verlag, 2012.
24. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of oservational determinism. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, CSFW'06, pages 3–. IEEE Computer Society, 2006.
25. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4):291–347.
26. S.A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.
27. G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF'07, pages 218–232. IEEE Computer Society, 2007.
28. H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 116–133. Springer-Verlag, 2010.
29. T.M. Ngo and M. Huisman. Quantitative security analysis for multi-threaded programs. *CoRR*, abs/1306.2693, 2013.

30. T.M. Ngo, M. Stoelinga, and M. Huisman. Confidentiality for probabilistic multi-threaded programs and its verification. In *Proceedings of the 5th international conference on Engineering Secure Software and Systems*, ESSoS'13, pages 107–122. Springer-Verlag, 2013.
31. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.
32. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. In *Inf. Processing Letters*, volume 63, pages 243–246, 1997.
33. A.W. Roscoe. CSP and determinism in security modeling. In *IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 1995.
34. A. Sabelfeld and A. Myers. Language-based information flow security. In *IEEE Journal on Selected Areas in Communications*, volume 21, pages 5–19, 2003.
35. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE workshop on Computer Security Foundations*, CSFW'00, pages 200–. IEEE Computer Society, 2000.
36. P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF'07, pages 203–217. IEEE Computer Society, 2007.
37. G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings of the 16th IEEE workshop on Computer Security Foundations*, CSFW'03. IEEE Computer Society, 2000.
38. G. Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'09, pages 288–302. Springer-Verlag, 2009.
39. T. Terauchi. A type system for observational determinism. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF'08, pages 287–300. IEEE Computer Society, 2008.
40. R. van der Meyden and C. Zhang. Algorithmic verification of noninterference properties. *Electron. Notes Theor. Comput. Sci.*, 168:61–75, 2007.
41. P. Černý and R. Alur. Automated analysis of java methods for confidentiality. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV'09, pages 173–187. Springer-Verlag, 2009.
42. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Comput. Secur.*, 7:231–253, 1999.
43. S. Yoshihama, T. Yoshizawa, Y. Watanabe, M. Kudoh, and K. Oyanagi. Dynamic information flow control architecture for web applications. In *Proceedings of the 12th European conference on Research in Computer Security*, ESORICS'07, pages 267–282. Springer-Verlag, 2007.
44. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Proceedings of 16th IEEE Computer Security Foundations Workshop*, CSFW'03, pages 29–43. IEEE Computer Society, 2000.
45. J. Zhu and M. Srivatsa. Quantifying information leakage in finite order deterministic programs. *CoRR*, abs/1009.3951, 2010.