Verifying Observational Determinism

Jaber Karimpour, Ayaz Isazadeh, Ali Noroozi

Verifying Observational Determinism

Jaber Karimpour, Ayaz Isazadeh, and Ali A. Noroozi

Computer Science, University of Tabriz, Iran
{karimpour,isazadeh,noroozi}@tabrizu.ac.ir

Abstract. This paper proposes an approach to verify information flow security of concurrent programs. It discusses a hyperproperty called observational determinism which aims to ensure secure information flow in concurrent programs, and proves how this hyperproperty can be verified by stutter equivalence checking. More precisely, it defines observational determinism in terms of stutter equivalence of all traces having the same low initial value and shows how stutter trace equivalence can be verified by computing a divergence stutter bisimulation quotient. The approach is illustrated by verifying a small example.

Keywords: Secure information flow. Observational determinism. Verification. Bisimulation

1 Introduction

To perform an effective security analysis of a given program, program model, *security policy* and attacker (observer) model should be defined precisely [1]. In secure information flow analysis, the program model can be represented as a state machine, which produces a set of executions and is considered public knowledge. In this model, program variables are classified into different security levels. A nave classification is to label some variables as L, meaning low security, public information; and other variables as H, meaning high security, private information. The goal of a security policy is to prevent information in H from flowing to L and being leaked [2], [3]. Other classifications of program variables are possible via a lattice of security levels [4]. In this case, the security policy should ensure that information flows only upwards in the lattice.

The security policy is a property that needs to be satisfied by the program model. The attacker is assumed to be able to observe program executions. *Confidentiality policies* are of major concern in security policies. These policies are connected to the ability of an attacker to distinguish two program executions that differ only in their confidential inputs. Noninterference is a confidentiality policy that stipulates an observer able to see only low security data (low observer) learns nothing about high security inputs by watching low security outputs [5]. *Observational determinism* is another confidentiality policy which is a generalized notion of noninterference for concurrent programs. Inspired by earlier work by McLean [6] and Roscoe [7], Zdancevic and Myers [5] proposed observational determinism which requires the program to produce indistinguishable traces to avoid information leaks. Thus, a program satisfying this condition appears deterministic to a low observer who is able to observe low variables and unable to distinguish states which differ only in high variables. As stated by Huisman et al. [8] *"concurrent programs are often used in a context where intermediate states can be observed."* That's why Zdancevic and Myers require determinism on all the states of a trace, instead of final states. Observational determinism is a widely accepted security property for concurrent programs. Observational deterministic programs are immune to *refinement attacks* [5], because observational determinism is preserved under refinement [9].

This paper concentrates on the problem of verifying observational determinism for concurrent programs. We define observational determinism in terms of stutter equivalence on all low variables. Our contributions include (1) a theorem showing that verifying secure information flow can be reduced to equivalence checking in the quotient system and (2) a sound model checking approach for verifying secure information flow in concurrent programs. In fact, our approach is the first that uses quotient space to reduce the state space and check for secure information flow simultaneously. We illustrate the progress made by the verification of a small example program. It is expected that these contributions constitute a significant step towards more widely applicable secure information flow analysis.

The remainder of the paper is organized as follows. In section 2, preliminaries are explained. In section 3, observational determinism is formally defined and section 4 discusses how to verify it. In section 5, some related work is discussed. Finally, Section 6 concludes, and discusses future work.

2 Preliminaries

In this section, at first we introduce the program model considered throughout the paper. Then, some preliminary concepts about bisimulation are discussed. Most of these preliminaries are taken from [10].

Definition 1 (Kripke structure). A *Kripke structure* KS is a tuple $(S, \rightarrow, I, AP, La)$ where S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, AP is the set of atomic propositions, and $La : S \rightarrow 2^{AP}$ is a labeling function. Here, atomic propositions are possible values of the low variables. KS is called *finite* if S and AP are finite. The set of successors of a state s is defined as $Post(s) = \{s' \in S | s \rightarrow s'\}$. A state s is called *terminal* if $Post(s) = \varnothing$. For a Kripke structure modelling a sequential program, terminal states represent the termination of the program.

Definition 2 (Execution or Path). A finite *path fragment* $\hat{\pi}$ of KS is a finite state sequence $s_0 s_1 \dots s_n$ such that $s_i \in Post(s_{i-1})$ for all $0 < i \leq n$. An *infinite* path fragment π is an infinite state sequence $s_0 s_1 s_2 \dots$ such that $s_i \in Post(s_{i-1})$ for all $0 < i$. A *maximal* path fragment is either a finite path fragment that ends in a terminal state, or an infinite path fragment. A path fragment is called *initial* if it starts in an initial state, i.e., if $s_0 \in I$. A *path* of KS is an initial, maximal path fragment. $Paths(s)$ denotes the set of paths starting in s. All paths of a Kripke structure with no terminal state are infinite.

Definition 3 (Trace). The *trace* of a path $\pi = s_0 s_1 \dots$ is defined as $T = trace(\pi) = La(s_0)La(s_1)\dots$. Thus, the trace of a path is the sequence of sets of atomic propositions that are valid in the states of the path. $T[0]$ extracts the first atomic proposition of the trace, i.e., $T[0] = La(s_0)$. Let $Traces(s)$ denote the set of traces of s, and $Traces(KS)$ the set of traces of the initial states of KS: $Traces(s) = trace(Paths(s))$ and $Traces(KS) = \cup_{s \in I} Traces(s)$.

Definition 4 (Combination of Kripke structures $KS_1 \oplus KS_2$). For $KS_i = (S_i, \rightarrow_i, I_i, AP, La_i)$, $i = 1, 2$: $KS_1 \oplus KS_2 = (S_1 \uplus S_2, \rightarrow_1 \cup \rightarrow_2, I_1 \cup I_2, AP, La)$ where $\uplus$ stands for disjoint union and $La(s) = La_i(s)$ if $s \in S_i$.

Definition 5 (Stutter equivalence). Traces T_1 and T_2 over 2^{AP} are *stutter equivalent*, denoted $T_1 \triangleq T_2$, if they are both of the form $A_0^+ A_1^+ A_2^+ \dots$ for $A_0, A_1, A_2, \dots \subseteq AP$ where A_i^+ is the *Kleene plus* operation on A_i and is defined as $A_i^+ = \{x_1 x_2 \dots x_k | k > 0 \text{ and each } x_i = A_i\}$. Kripke structures KS_i over AP, $i = 1, 2$, are *stutter trace equivalent*, denoted $KS_1 \triangleq KS_2$, if $KS_1 \trianglelefteq KS_2$ and $KS_2 \trianglelefteq KS_1$, where $\trianglelefteq$ is defined by:

$$KS_1 \trianglelefteq KS_2 \text{ iff } \forall T_1 \in Traces(KS_1)(\exists T_2 \in Traces(KS_2).\ T_1 \triangleq T_2)$$

Definition 6 (Stutter bisimulation). A *stutter bisimulation* for KS is a binary relation R on S such that for all $(s_1, s_2) \in R$, the following three conditions hold: (1) $La(s_1) = La(s_2)$. (2) If $s_1' \in Post(s_1)$ with $(s_1', s_2) \notin R$, then there exists a finite path fragment $s_2 u_1 \dots u_n s_2'$ with $0 \le n$ and $(s_1, u_i) \in R$, $i = 1, \dots, n$ and $(s_1', s_2') \in R$. (3) If $s_2' \in Post(s_2)$ with $(s_1, s_2') \notin R$, then there exists a finite path fragment $s_1 v_1 \dots v_n s_1'$ with $0 \le n$ and $(v_i, s_2) \in R$, $i = 1, \dots, n$ and $(s_1', s_2') \in R$.

Definition 7 (Divergence stutter bisimulation). Let R be an equivalence relation on S. A state $s \in S$ is *R-divergent* if there exists an infinite path fragment $\pi = s s_1 s_2 \dots \in Paths(s)$ such that $(s, s_j) \in R$ for all $0 < j$. Stated in words, a state s is R-divergent if there is an infinite path starting in s that only visits states in $[s]_R$. $[s]_R$ is the equivalence class of s under the equivalence relation R. R is *divergence-sensitive* if for any $(s_1, s_2) \in R$: if s_1 is R-divergent, then s_2 is R-divergent. States s_1, s_2 are *divergent stutter bisimilar*, denoted $s_1 \approx^{div} s_2$, if there exists a divergence sensitive stutter bisimulation R such that $(s_1, s_2) \in R$.

Definition 8 (Divergent stutter bisimilar paths). For infinite path fragments $\pi_i = s_{0,i} s_{1,i} s_{2,i} \dots$, $i = 1, 2$ in KS, π_1 is *divergent stutter bisimilar* to π_2, denoted $\pi_1 \approx^{div} \pi_2$ if and only if there exists an infinite sequence of indices $0 = j_0 < j_1 < j_2 < \dots$ and $0 = k_0 < k_1 < k_2 < \dots$ with:

$$s_{j,1} \approx^{div} s_{k,2} \text{ for all } j_{r-1} \le j < j_r \text{ and } k_{r-1} \le k < k_r \text{ with } r = 1, 2, \dots$$

The following lemma follows directly from the definition of $\approx^{div}$ on paths and $\triangleq$ on paths.

Lemma 1. *For all infinite paths π_1 and π_2, we have $\pi_1 \approx^{div} \pi_2$ implies $\pi_1 \triangleq \pi_2$.*

Lemma 2. *Divergent stutter bisimilar states have divergent stutter bisimilar paths:*

$$s_1 \approx^{div} s_2 \text{ implies } \forall \pi_1 \in Paths(s_1)\ (\exists \pi_2 \in Paths(s_2).\ \pi_1 \approx^{div} \pi_2)$$

Proof: see [10], page 549.

Definition 9 (Divergence stutter bisimulation quotient $KS/\approx^{div}$). The *quotient* of a Kripke structure KS is defined by $KS/\approx^{div} = (S/\approx^{div}, \rightarrow', I', AP, La')$, where $S/\approx^{div} = \{[s]_{\approx^{div}} | s \in S\}$, $La'([s]_{\approx^{div}}) = La(s)$, and $\rightarrow'$ is defined by

$$\frac{s \rightarrow s' \ \wedge \ s \not\approx^{div} s'}{[s]_{\approx^{div}} \rightarrow' [s']_{\approx^{div}}} \quad \text{and} \quad \frac{s \ \ is \ \ \approx^{div} -divergent}{[s]_{\approx^{div}} \rightarrow' [s]_{\approx^{div}}}$$

Theorem 1. *For any Kripke structure KS, we have $KS \approx^{div} KS/\approx^{div}$.*

Proof: Follows from the fact that $R = \{(s, [s]_{\approx^{div}}) | s \in S\}$ is a divergence stutter bisimulation for $(KS, KS/\approx^{div})$. ■

3 Observational Determinism

A concurrent program is secure if it appears deterministic to a low observer and produces indistinguishable executions. Zdancevic and Myers [5] call this observational determinism and define it as:

$$\forall\ T, T' \in Traces(P).\ \ T[0] =_L T'[0] \Longrightarrow T \approx_L T'$$

where KS is a model of the program (e.g., a Kripke structure, modelling the program executions). Indistinguishability to a low observer is expressed as state equivalence relation $=_L$ and trace equivalence relation $\approx_L$. Zdancevic and Myers define trace equivalence as prefix and stutter equivalence of the sequence of states in each trace. However, Huisman et al. [8] argue that allowing prefixing causes information flows. That's why they propose stutter equivalence instead of prefix and stutter equivalence. For example, consider the following program:

```
        l:=0; while(h>0) then {l++}                  (P1)
```

where h is a high variable and l is a low veriable. The set of traces of this program is $\{<0>, <0,1>, <0,1,2>, \dots\}$. These traces are prefix and stutter equivalent, hence considered secure by the definition of Zdancevic and Myers; But, the attacker can easily get the value of h by observing the traces. Huisman et al. [8] require stutter equivalence of traces of each low variable, but as Terauchi [11] shows, this kind of definition is not as restrictive as possible and accepts leaky programs. Thus, Terauchi requires prefix stutter equivalence of all traces w.r.t. all low variables.

Consequently, we define observational determinism in terms of stutter equivalence on all low variables:

$$\forall\, T, T' \in \mathit{Traces}(P).\;\; T[0] =_L T'[0] \Longrightarrow T \triangleq_L T'$$

where $\triangleq_L$ is stutter trace equivalence. For example, consider the following insecure program that can reveal the value of h:

```
l₁:=0; l₂:=0;
l₁:=1 || if(l₁=1) then l₂:=h                    (P2)
```

where $||$ is the parallel operator. If the right program is executed first, the corresponding trace of low variables would be: $<(0,0),(0,0),(1,0)>$. Each ordered pair (l_1,l_2) shows the values of low variables in each state of the program. If the left program is executed first, the following traces are produced: $<(0,0),(1,0),(1,h)>$. As you can see, these traces are not stutter equivalent, so the program is insecure. As another example, consider the following secure program:

```
l₁:=0; l₂:=0;
l₁:=2 || if(l₁=1) then l₂:=h                    (P3)
```

If the right program is executed first, the corresponding trace would be: $<(0,0),(0,0),(2,0)>$, but if the left program is executed first, the following trace is produced: $<(0,0),(2,0),(2,0)>$. These two traces are stutter equivalent, hence the program is secure. Thus, this paper defines trace indistinguishability in observational determinism as stutter equivalence.

4 Verification of Observational Determinism

Let us assume $KS = (S, \rightarrow, I, AP, La)$ is a Kripke structure that models the behavior of the concurrent execution of the processes (or threads) of a concurrent program. AP is the set of the values of low variables and the function La labels each state with these values. It is assumed that the state space of KS is finite. If KS has a terminal state s_n, we include a transition $s_n \rightarrow s_n$, i.e., a self-loop, ensuring that the Kripke structure has no terminal state. Therefore, all traces of KS are infinite.

The main steps of the verification algorithm are outlined in Algorithm 1. The input of this algorithm is a finite Kripke structure KS modeling the program, and the output is *yes* or *no*. The first step is to partition the set I of initial states into sets of low equivalent initial states called initial state clusters $ISC_1, \ldots, ISC_m$, and define $ISC = \{ISC_1, \ldots, ISC_m\}$. The second step is to extract an arbitrary trace T_i from KS for each initial state cluster and build a Kripke structure KST_i from the path in KS corresponding to trace T_i. As the next step, we combine Kripke structures KST_i ($i = 1, \ldots, |ISC|$), forming a single Kripke structure $KST = (S_{KST}, \rightarrow_{KST}, I_{KST}, AP_{KST}, La_{KST})$, where

$KST = KST_1 \oplus KST_2 \oplus \ldots KST_{|ISC|}$. The following theorem reduces the problem of verifying observational determinism to checking divergence stutter bisimulation of KS and KST.

```
input : finite Kripke structure $KS$ modeling the program
output: yes if the program satisfies observational determinism;
        Otherwise, no

1  Partition the set $I$ of initial states into initial state clusters;
2  Take an arbitrary trace $T_i$ of $KS$ for each initial state cluster;
3  Construct Kripke structure $KST_i$ from the path in $KS$ corresponding to
   trace $T_i$;
4  Construct Kripke structure $KST = KST_1 \oplus KST_2 \cdots \oplus KST_{|ISC|}$;
5  Compute divergence stutter bisimulation quotient of $KS \oplus KST$;
6  for each initial state cluster $ISC_i$ do
7      for each pair of initial states $s_0$ and $s'_0$ in $ISC_i$ do
8          if $[s_0]_{\approx^{div}_{KS\oplus KST}} \neq [s'_0]_{\approx^{div}_{KS\oplus KST}}$ then
9              return no;
10         else
11     end
12     for an arbitrary state $s_0$ in $ISC_i$ and its correspondent initial state
       $st_0$ in $KST$ do
13         if $[s_0]_{\approx^{div}_{KS\oplus KST}} \neq [st_0]_{\approx^{div}_{KS\oplus KST}}$ then
14             return no;
15         else
16     end
17 end
18 return yes;
```

Algorithm 1: Verification of observational determinism

Theorem 2. *The problem of the verification of observational determinism is reduced to the following problem:*

$$\forall C \in (S \uplus S_{KST})/\approx^{div}_{KS\oplus KST},\ \forall s_0, s'_0 \in ISC_i,\ 1 \leq i \leq |ISC|.$$
$$s_0 \in C \Leftrightarrow s'_0 \in C \quad \text{and} \quad ISC_i \cap C \neq \phi \Leftrightarrow I_{KST} \cap C \neq \phi$$

where $\uplus$ stands for disjoint union and $(S \uplus S_{KST})/\approx^{div}_{KS\oplus KST}$ denotes the quotient space with respect to $\approx^{div}_{KS\oplus KST}$, i.e., the set of all divergence stutter bisimulation equivalence classes in $S \uplus S_{KST}$.

Proof: To prove that KS is observational deterministic, one should prove that for every initial state of KS, all traces starting in that state are stutter equivalent. Thus, for all traces starting from an initial state s_0 of KS, there should be a stutter equivalent trace of KST starting from an initial state st_0 of KST. By Lemma 1, stutter equivalence of traces reduces to divergence stutter bisimulation. From Lemma 2, it follows that each initial state of KS should be divergent stutter bisimilar to an initial state in KST, and vice versa. Thus, KS

and KST should be divergent stutter bisimilar. Then, $KS \approx^{div} KST$ if and only if

$$\forall C \in (S \uplus S_{KST})/ \approx^{div}_{KS \oplus KST} . \quad I \neq \phi \Longleftrightarrow I_{KST} \neq \phi$$

where $(S \uplus S_{KST})/ \approx^{div}_{KS \oplus KST}$ denotes the quotient space with respect to $\approx^{div}_{KS \oplus KST}$. Considering that some initial states may be low equivalent and consequently they form initial state clusters, it is sufficient to take an arbitrary trace for only each initial state cluster. Of course, all states of an initial state cluster should have stutter equivalent traces and thus should be divergent stutter bisimilar. As a result, KS is observational deterministic if and only if

$$\forall C \in (S \uplus S_{KST})/ \approx^{div}_{KS \oplus KST},\ \forall s_0, s'_0 \in ISC_i,\ 1 \leq i \leq |ISC|.$$
$$s_0 \in C \Leftrightarrow s'_0 \in C \quad \text{and} \quad ISC_i \cap C \neq \phi \Leftrightarrow I_{KST} \cap C \neq \phi \qquad \blacksquare$$

The input finite Kripke structure KS has no terminal states. Hence, all traces are infinite and form a cycle in KS. To take an arbitrary trace from KS, we can use *cycle detection* algorithms of graphs. In order to detect cycle, a modified depth first search called *colored DFS* may be used. In colored DFS, all states are initially marked white. When a state is encountered, it is marked grey, and when its successors are completely visited, it is marked black. If a grey state is ever encountered, then there is a cycle and sequence of states pushed in the stack of the DFS so far forms a path.

It remains to explain how to compute divergence stutter bisimulation quotient of $KS \oplus KST$. The algorithm to compute the quotient relies on a partition refinement technique, where the finite state space $S \uplus S_{KST}$ is partitioned in blocks. Starting from a straightforward initial partition, where all states with the same label (low variable values) form a partition, the algorithm successively refines these partitions until a stable partition is reached. A partition is stable if it only contains divergent stutter bisimilar states and no refinement is possible on it. Further details can be found, e.g. in [12], [10].

Complexity. The time complexity of finding an arbitrary trace is $O(M)$, where M denotes the number of transitions of KS. Thus, the time complexity of constructing KST is $O(|I|.M)$. The quotient space of $KS \oplus KST$ under $\approx^{div}$ can be computed in time $O((|S| + M) + |S|.(|AP| + M))$ under the assumption that $M|S|$ [10]. Thus, the costs of verifying observational determinism for concurrent programs are dominated by the costs of computing the quotient space under $\approx^{div}$, which is polynomial-time.

Using the quotient space to verify observational determinism has two advantages: (1) Instead of analyzing the concrete model of the program, a minimized abstract model is analyzed. Provided the quotiening preserves stutter equivalence, the analysis of the minimized model suffices to decide the satisfaction of observational determinism in the program. (2) The proposed approach can easily be adapted to verify programs with infinitely many states, as there are efficient algorithms for computing the quotient space of infinite state programs [13].

Example. Consider the following program from [14]:

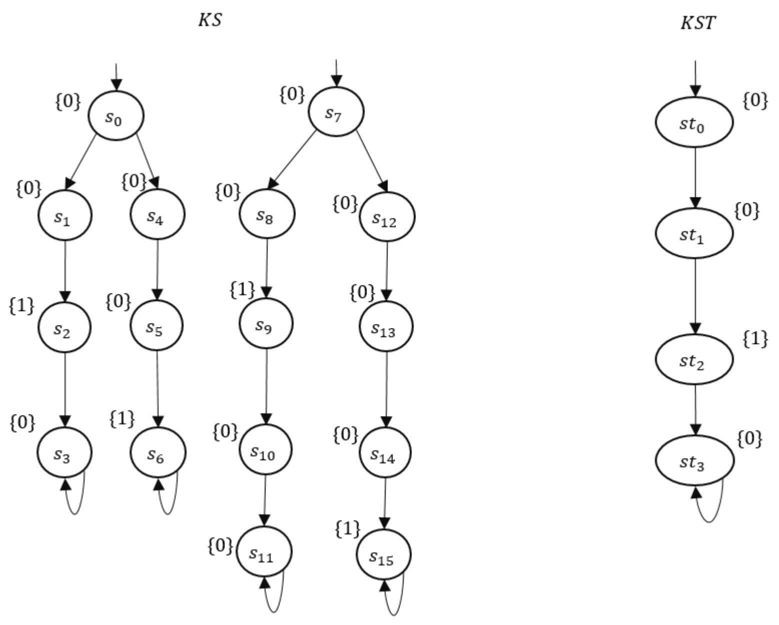

Fig. 1. The Kripke structure KS of P5 and the Kripke structure KST of an arbitrary trace of KS

```
l:=0;
if(h) then {l:=0; l:=1} || l:=0
      else {l:=0; l:=1} || {l:=0; l:=0}     (P4)
```

where h is a boolean and high variable and l is a low variable. The Kripke structure KS of the program P4 and the Kripke structure KST of an arbitrary trace of KS are shown in Figure 1. The equivalence classes of $KS \oplus KST$ under $\approx^{div}$ are $[s_0]_{div} = \{s_0, s_7\}$, $[s_1]_{div} = \{s_1, s_8, st_0, st_1\}$, $[s_2]_{div} = \{s_2, s_9, st_2\}$, $[s_3]_{div} = \{s_3, s_{10}, s_{11}, st_3\}$, $[s_4]_{div} = \{s_4, s_5, s_{12}, s_{13}, s_{14}\}$ and $[s_6]_{div} = \{s_6, s_{15}\}$. Therefore, the divergence stutter bisimulation quotient of it is computed as depicted in Figure 2. Since the initial state s_0 of KS and the initial state st_0 of KST are not in the same equivalence class, the program is labelled as insecure.

5 Related Work

Zdancevic and Myers [5] define observational determinism in terms of prefix and stutter equivalence each low variable. Huisman et al. [8] show that allowing prefixing permits some leaks, so they define observational determinism in terms of stutter equivalence on each low variable. Terauchi [11] shows that independent consideration of low variables is not correct and information flows may occur. Hence, he proposes prefix and stutter equivalence on all low variables. Huisman

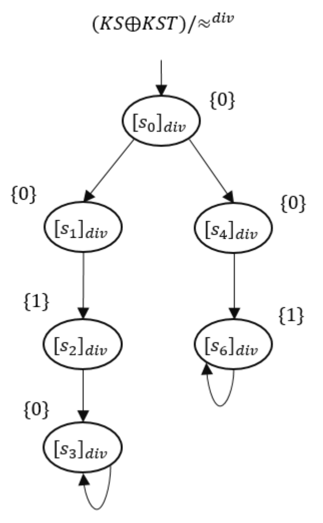

Fig. 2. The divergence stutter bisimulation quotient of $KS \oplus KST$ for `P5`

and Blondeel [15] define observational determinism in terms of stutter equivalence on all low variables. Ngo et al. [16] defines observational determinism with two conditions: condition 1 requires stutter equivalence on each low variable and condition 2 requires stutter equivalence on all low variables. Ngo et al. argue that a concurrent program satisfying both of these conditions is secure.

A common approach to verify information flow properties is the use of type systems. A type system is a collection of type inference rules for assigning types to variables of a programming language and making judgments about programs [17]. With a type system developed for secure information flow, a program can be type checked for secure-flow violations [18]. But type systems are not extensible, as for each variation of confidentiality policy and programming language to be verified, a new type system should be redefined and proven sound. For more details about disadvantages of type systems, see [19].

Accordingly, logic-based verification and model checking has been advocated. But as security policies are not trace properties [6], [21], standard logic-based verification and model checking methods can't be utilized and need to be modified. Trace properties, e.g. safety or liveness properties [10], are properties of individual traces; But most of security policies, such as secure information flow, are properties of sets of traces, called hyperproperties [21]. For instance, observational determinism is a hyperproperty because a trace is allowed if it is indistinguishable from all other traces having the same low vales. Various attempts for logical specification of information flow properties has been made. One promising one is self-composition which composes a program with a copy of it, with all variables renamed. Then, the problem of verifying secure information flow is reduced to a trace property for the composed program. Huisman and Blondeel [15] use this idea to model check observational determinism for multi-threaded programs. They specify observational determinism in μ-calculus. A disadvantage with these kinds of methods that exploit logical verification and self-composition is the common problem of state space explosion [22], [23].

The concept of stutter equivalence in the definition of observational determinism brings weak bisimulation to mind. Ngo et al. [16] use bisimulation to verify observational determinism in multi-threaded programs. They make a copy of the program for each initial state of the program and check bisimilarity of each pair of the programs after removing stutter steps and determinizing them. But the cost of determinizing a program is exponential in the size of the program. This method suffers from state space explosion problem too, as it makes a copy of the program for each initial state of it.

Another line of research for verifying hyperproperties is to extend temporal logics and introduce new logics to specify these properties. Many attempts have been made, including HyperLTL, HyperCTL* [9], HyperCTL [24], SecLTL [25], etc. Clarkson et al. [9] specify observational determinism and many other security properties in HyperLTL and provide model checking techniques for verifying HyperLTL and HyperCTL*. Finkbeiner et al. [24] introduce HyperCTL, which extends CTL* with path variables. They reduce the model checking problem for HyperCTL to the satisfiability problem for QPTL to obtain a model checking algorithm for HyperCTL. Dimitrova et al. [25] add a new modal operator, the hide operator, to LTL and name the resulting logic SecLTL. They propose an automata-theoretic technique for model checking SecLTL.

A similar research to our work is Mantel's unwinding possibilistic security properties [26]. In this work, he proposes a modular framework in which most security properties can be composed from a set of basic security predicates (BSPs); he also presents unwinding conditions for most BSPs. These unwinding conditions can be seen as simulation relations on system states. Intuitively, unwinding conditions require that each high transition is simulated in such a way that a low observer cannot infer whether such high transition has been performed or not. Thus the low observation of the process is not influenced in any way by its high behavior. In 2011, D'Souza et al. [27] propose an automata-theoretic technique for model checking Mantel's BSPs. The proposed model checking approach is based on deciding set inclusion on regular languages.

6 Conclusion

This paper discusses a bisimulation-based approach for model checking observational determinism in concurrent programs. Concretely, we extract some trace(s) of the program and check stutter trace equivalence between the trace(s) and the program. This is done by computing a bisimulation quotient. The time complexity of the verification is polynomial. The advantage of our proposed approach is that the analysis is done on a minimized abstract model of the program. Hence, the state explosion problem may be avoided. Another advantage is that the approach can be easily adapted for infinite state programs.

As future work, we plan to implement the proposed approach. We will also study whether bisimulation-based modular minimization algorithms are appropriate for verifying observational determinism. We also aim to modify our algorithm and use compositional verification techniques to benefit from modular structure of the concurrent program.

References

1. Balliu, M.: Logics for information flow security: from specification to verification. (2014)
2. Smith, G.: Principles of secure information flow analysis. In: Malware Detection. Springer US, pp. 291-307 (2007)
3. Sabelfeld, A., Myers A. C.: Language-based information-flow security. In: Selected Areas in Communications, IEEE Journal on, vol. 21, no. 1, pp. 5-19 (2003)
4. Denning, D.: A lattice model of secure information flow. In: Communications of the ACM, vol. 19, no. 5, pp. 236-243 (1976)
5. Zdancewic, S., Myers, A. C.: Observational determinism for concurrent program security. In: Computer Security Foundations Workshop, Proceedings. 16th IEEE (2003)
6. McLean, J.: Proving noninterference and functional correctness using traces. In: Journal of Computer security, vol. 1, no. 1, pp. 37-57 (1992).
7. Roscoe, A. W.: CSP and determinism in security modelling. In: Security and Privacy, Pro-ceedings., IEEE Symposium on (1995)
8. Huisman, M., Worah, P., Sunesen, K.: A temporal logic characterisation of observational de-terminism. In: Computer Security Foundations Workshop, 19th IEEE (2006)
9. Clarkson, M. R., Finkbeiner, B., Koleini, M., Micinski, K. K., Rabe, M. N., Snchez, C.: Temporal Logics for Hyperproperties. In: Principles of Security and Trust, Springer Berlin Heidelberg, pp. 265-284 (2014)
10. Baier, C., Katoen, J. P.: Principles of model checking. Vol. 26202649. Cambridge: MIT press (2008)
11. Terauchi, T.: A type System for observational determinism. In: Computer Security Founda-tions, pp. 287-300 (2008)
12. Groote, J. F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stutter-ing equivalence. Springer Berlin Heidelberg, pp. 626-638 (1990)
13. Chutinan, A., Krogh, B. H.: Verification of infinite-state dynamic systems using approxi-mate quotient transition systems. In: Automatic Control, IEEE Transactions on, vol. 46, no. 9, pp. 1401-1410 (2001)
14. Ngo, T. M.: Qualitative and quantitative information flow analysis for multi-thread pro-grams. University of Twente (2014)
15. Huisman, M., Blondeel, H. C.: Model-checking secure information flow for multi-threaded programs. In: Theory of Security and Applications, Springer Berlin Heidelberg, pp. 148-165 (2012)
16. Ngo, T. M., Stoelinga, M., Huisman, M.: Effective verification of confidentiality for multi-threaded programs. In: Journal of Computer Security, vol. 22, no. 2, pp. 269-300 (2014)

17. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. In: Jour-nal of computer security, vol. 4, no. 2, pp. 167-187 (1996)
18. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of pro-gramming languages. ACM, pp. 355-364 (1998)
19. Barthe, G., D'argenio, P. R., Rezk, T.: Secure information flow by self-composition. In: Mathematical Structures in Computer Science, vol. 21, no. 06, pp. 1207-1252 (2011)
20. McLean, J.: A general theory of composition for trace sets closed under selective interleav-ing functions. In: Research in Security and Privacy, Proceedings, IEEE Computer Society Symposium on, pp. 79-93 (1994)
21. Clarkson, M. R., Schneider, F. B.: Hyperproperties. In: Journal of Computer Security, vol. 18, no. 6, pp. 1157-1210 (2010)
22. Clarke, E. M.: The birth of model checking. In: 25 Years of Model Checking. Springer Ber-lin Heidelberg, pp. 1-26 (2008)
23. Emerson, E. A.: The beginning of model checking: A personal perspective. In: 25 Years of Model Checking. Springer Berlin Heidelberg, pp. 27-45 (2008)
24. Finkbeiner, B., Rabe, M. N., Snchez, C.: A temporal logic for hyperproperties. In: arXiv preprint arXiv:1306.6657 (2013)
25. Dimitrova, R., Finkbeiner, B., Kovcs, M., Rabe, M. N., Seidl, H.: Model checking infor-mation flow in reactive systems. In: Verification, Model Checking, and Abstract Interpreta-tion, Springer Berlin Heidelberg, pp. 169-185 (2012)
26. Mantel, H.: Unwinding possibilistic security properties. In: Computer Security-ESORICS 2000. Springer Berlin Heidelberg, pp. 238-254 (2000)
27. D'Souza, D., Holla, R., Raghavendra, K. R., Sprick, B.: Model-checking trace-based infor-mation flow properties. In: Journal of Computer Security, vol. 19, no. 1, pp. 101-138 (2011)