Secure Information Flow Analysis Using the PRISM Model Checker

Ali A. Noroozi(✉), Khayyam Salehi, Jaber Karimpour, and Ayaz Isazadeh

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

Abstract. Secure information flow checks whether sensitive information leak to public outputs of a program or not. It has been widely used to analyze the security of various programs and protocols and guarantee their confidentiality and robustness.

In this paper, the problem of verifying secure information flow of concurrent probabilistic programs is discussed. Programs are modeled by Markovian processes and secure information flow is specified by observational determinism. Then, two algorithms are proposed to verify observational determinism in the Markovian model. The algorithms employ a trace-based approach to traverse the model and check for satisfiability of observational determinism. The proposed algorithms have been implemented into a tool called PRISM-Leak, which is constructed on the PRISM model checker. An anonymity protocol, the dining cryptographers, is discussed as a case study to show how PRISM-Leak can be used to evaluate the security of programs. The scalability of the tool is demonstrated by comparing it to the state-of-the-art information flow tools.

Keywords: Information security · Secure information flow · Observational determinism · Markovian processes · PRISM-Leak

1 Introduction

Secure information flow is an important mechanism to discover leakages in various programs and protocols [3,28]. Leakages occur when an attacker infers information about *secret* inputs of a program by observing its *public* outputs. In order to detect leakages and prevent insecure information flows, a security property needs to be defined to specify secure behavior of the program and a verification method is used to check whether the property holds or not.

Many security properties have been introduced in the literature, including *observational determinism*, which specifies secure information flow for concurrent programs. Introduced by McLean [17] and Roscoe [26] and improved by Zdancewic and Myers [32] and many others [9,12,14,15,20,21,31], observational determinism requires a concurrent program to produce traces, i.e., sequences of public values, that appear deterministic and thus indistinguishable to the attacker. However, existing definitions of observational determinism are not precise enough and are rather too restrictive or too permissive. An ideal security property should be restrictive enough to reject insecure programs and permissive enough to accept secure programs. Furthermore, most of these definitions are scheduler-independent [9,12,14,15,21,31,32]. Since the security of a concurrent program depends on the choice of a scheduler and might change by modifying the scheduler, observational determinism needs to be defined scheduler-specific [20].

For verifying satisfiability of observational determinism, various methods, including type systems [31,32], logics [9,12,14] and algorithmic verification [15,20,21] have been used. Type systems are often too restrictive and non-automatic. Logic-based methods can be precise, but require a significant amount of manual effort. Algorithmic verification is automatic, but existing methods are not scalable. In fact, there is no automatic and scalable algorithmic verification tool for checking observational determinism.

In this paper, an automatic approach is proposed to specify and algorithmically verify observational determinism for concurrent probabilistic programs using the PRISM model checker. Assume a concurrent program that contains probabilistic modules with shared variables and a probabilistic scheduler that determines the execution order of statements of the modules. The program contains public, secret and possibly neutral variables. The set of public variables is denoted by L. Furthermore, assume an attacker that is able to pick a scheduler, run the program under control of the scheduler and observe the program traces. The attacker does not influence the initial values of the public variables, i.e., the program has no public input.

Considering these assumptions, we model programs using Markovian processes. Observational determinism is defined to be scheduler-specific and more precise. It contains two conditions, OD_1 and OD_2, which a program needs to satisfy both to be observationally deterministic. OD_1 requires prefix and stutter equivalence for traces of each public variable and OD_2 enforces existential stutter equivalence for traces of all public variables. To verify these conditions, two trace-based algorithms are proposed. The proposed approach has been validated by implementing the algorithms in PRISM-Leak [24], which is a tool for evaluating secure information flow of concurrent probabilistic programs. PRISM-Leak has been built upon the PRISM model checker [16] to check the security of PRISM programs. Finally, a case study is discussed and the scalability of the proposed algorithms is compared to the state-of-the-art tools of information flow analysis. The experimental results show that PRISM-Leak has the best performance among the tools that are capable of analyzing the case study.

In summary, the paper contributes to the literature by

- a formal definition of observational determinism on a Markovian program model,
- two algorithms to verify the conditions of observational determinism,
- an automatic and scalable tool to verify observational determinism for concurrent probabilistic programs defined in the PRISM language.

The paper proceeds as follows. Section 2 provides the core background on the Markovian processes, the dining cryptographers protocol and various types of information flow channels considered in this paper. Section 3 discusses the related work and their strengths and weaknesses. Section 4 presents a formal definition of observational determinism and Sect. 5 proposes two verification algorithms. In Sect. 6, the verification algorithms are evaluated and compared to the existing approaches. We conclude the paper in Sect. 7 and discuss some future work.

2 Background

2.1 Markovian Models

Markovian models allow us to define states and transitions containing enough information to extract all traces of a program that are visible to the attacker. *Markov decision processes (MDPs)* are used to model operational semantics of concurrent probabilistic programs. Furthermore, *memoryless probabilistic schedulers*, a simple but important subclass of schedulers, are used to denote schedulers of concurrent programs. When a memoryless probabilistic scheduler is applied to an MDP, a *Markov chain (MC)* is produced, which is the final model used in this paper for specifying observational determinism and verifying it. Here, the notations used throughout the paper are formally defined. For more information on how Markovian models and schedulers work, please see chapter 10 of [1].

Definition 1. *A* ***Markov decision process (MDP)*** *is a tuple* $\mathcal{M} = (S, Act, \mathbf{P}, \zeta, Val_L, V)$ *where* S *is a set of states,* Act *is a set of actions,* $\mathbf{P} : S \times Act \times S \to [0,1]$ *is a transition probability function such that* $\forall s \in S.\ \forall \alpha \in Act.\ \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{0,1\}$*, the function* $\zeta : S \to [0,1]$ *is an initial distribution such that* $\sum_{s \in S} \zeta(s) = 1$*,* Val_L *is the finite set of values of the public variables and* $V : S \to Val_L$ *is a labeling function.*

An MDP $\mathcal{M}$ is called *finite* if S, Act, and Val_L are finite. An action α is *enabled* in state s if and only if $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 1$. Let $Act(s)$ denote the set of enabled actions in s. In our program model, actions represent the program statements.

An MDP with no action and nondeterminism is called a Markov Chain.

Definition 2. *A (discrete-time)* ***Markov chain (MC)*** *is a tuple* $\mathcal{M} = (S, \mathbf{P}, \zeta, Val_L, V)$ *where* $\mathbf{P} : S \times S \to [0,1]$ *is a transition probability function such that* $\forall s \in S.\ \sum_{s' \in S} \mathbf{P}(s, s') = 1$*. The other elements, i.e.,* S*,* ζ*,* Val_L *and* V *are the same as MDP.*

Given a state s, a memoryless probabilistic scheduler returns a probability for each action $\alpha \in Act(s)$. This random choice is independent of what has happened in the history, i.e., which path led to the current state. This is why it is called memoryless. Let $\mathcal{D}(\mathcal{X})$ denote the set of all probability distributions over a set $\mathcal{X}$. Formally,

Definition 3. *Let $\mathcal{M} = (S, Act, \mathbf{P}, \zeta, Val_L, V)$ be an MDP. A* ***memoryless probabilistic scheduler*** *for $\mathcal{M}$ is a function $\delta : S \rightarrow \mathcal{D}(Act)$, such that $\delta(s) \in \mathcal{D}(Act(s))$ for all $s \in S$.*

As all nondeterministic choices in an MDP $\mathcal{M}$ are resolved by a scheduler δ, a Markov chain $\mathcal{M}_\delta$ is induced. Formally,

Definition 4. *Let $\mathcal{M} = (S, Act, \mathbf{P}, \zeta, Val_L, V)$ be an MDP and $\delta : S \rightarrow \mathcal{D}(Act)$ be a memoryless probabilistic scheduler on $\mathcal{M}$. The* ***MC of $\mathcal{M}$ induced by δ*** *is given by*

$$\mathcal{M}_\delta = (S, \mathbf{P}_\delta, \zeta, Val_L, V)$$

where

$$\mathbf{P}_\delta(s, s') = \sum_{\alpha \in Act(s)} \delta(s)(\alpha).\mathbf{P}(s, \alpha, s').$$

In what follows, we fix an MC $\mathcal{M}_\delta^{\mathsf{P}} = (S, \mathbf{P}_\delta, \zeta, Val_L, V)$ which models the executions of the concurrent probabilistic program P under the control of a scheduler δ. A state of $\mathcal{M}_\delta^{\mathsf{P}}$ indicates the current values of variables, together with the current value of the program counter that indicates the next program statement to be executed. The function V *labels* each state with values of the public variables in that state. In fact, a state label is what an attacker observes in that state.

The set of *successors* of s is defined as $Post(s) = \{s' \mid \mathbf{P}_\delta(s, s') > 0\}$. The states s with $\zeta(s) > 0$ are considered as the *initial states*. The set of initial states of $\mathcal{M}_\delta^{\mathsf{P}}$ is denoted by $Init(\mathcal{M}_\delta^{\mathsf{P}})$. To ensure $\mathcal{M}_\delta^{\mathsf{P}}$ is non-blocking, we include a self-loop to each state s that has no successor, i.e., $\mathbf{P}_\delta(s, s) = 1$. Then, a state s is called *final* if $Post(s) = \{s\}$. It is assumed that all final states correspond to the termination of the program.

A **path** (or execution path) π of $\mathcal{M}_\delta^{\mathsf{P}}$ is an infinite state sequence $s_0 s_1 \ldots s_n^\omega$ such that $s_0 \in Init(\mathcal{M}_\delta^{\mathsf{P}})$, $s_i \in Post(s_{i-1})$ for all $0 < i \leq n$, s_n is a final state and ω denotes infinite iteration (self-loop over s_n). The set of paths starting from a state s is denoted by $Paths(s)$. The set of all paths of $\mathcal{M}_\delta^{\mathsf{P}}$ is denoted by $Paths(\mathcal{M}_\delta^{\mathsf{P}})$.

A **trace** of a path $\pi = s_0 s_1 \ldots s_n^\omega$ is defined as $T = trace_{|L}(\pi) = V(s_0)V(s_1)\ldots V(s_n)^\omega$. We refer to n as the length of T, i.e., $length(T) = n$. The labeling function, instead of all public variables, can be restricted to just a single public variable $l \in L$, i.e., $V_{|l} : S \rightarrow Val_l$. Then, the trace of π on l is defined as $T_{|l} = trace_{|l}(\pi) = V_{|l}(s_0)V_{|l}(s_1)\ldots V_{|l}(s_n)^\omega$. Note that $T_{|L} = trace_{|L}(\pi)$. The set of traces starting from a state s is denoted by $Traces(s)$. The set of all trace of $\mathcal{M}_\delta^{\mathsf{P}}$ is denoted by $Traces(\mathcal{M}_\delta^{\mathsf{P}})$.

Two traces T and T' are **stutter equivalent**, denoted $T \triangleq T'$, if they are both of the form $A_0^+ A_1^+ A_2^+ \ldots$ for $A_0, A_1, A_2, \cdots \subseteq Val_L$ where A_i^+ is the

Kleene plus operation on A_i and is defined as $A_i^+ = \{A_i^k \mid k \in \mathbb{N}, k \geq 1\}$. A finite trace T_1 is called a prefix of T, if there exists another infinite trace T_2 such that $T_1 T_2 = T$. Two traces are **prefix and stutter equivalent**, denoted by $\triangleq_p$, if one is stutter equivalent to a prefix of another. For example, the traces $[0, 0, 0, 1, 1^\omega]$ and $[0, 1, 1, 1^\omega]$ are stutter equivalent and the traces $[0, 0, 0, 2, 1^\omega]$ and $[0, 2, 1, 1, 4, 4^\omega]$ are prefix and stutter equivalent.

A state s is **low-equivalent** to another state s', written $s =_L s'$, if $V(s) = V(s')$. Low-equivalence can also be defined on a single public variable $l \in L$: $s =_l s'$ if $V_{|l}(s) = V_{|l}(s')$. This relation corresponds to the observational power of the attacker. When two states are low-equivalent, they are the same to the attacker, even if secret values differ in these states.

DAG Structure of Markovian Models. We assume the programs always terminate and states indicate the current values of the variables and the program counter. Furthermore, loops of the program are unfolded. This implies that Markovian models of every program takes the form of a *directed acyclic graph (DAG)*, ignoring later-added self-loops of final states. Initial states of the program are represented as roots of the DAG and final states as leaves. Therefore, there is no loop in the Markovian models (except later-added self-loops) and all path lead to a final state.

2.2 Dining Cryptographers Protocol

We use the *dining cryptographers protocol* [7] as a base to compare precision of various definitions of observational determinism. It is well-known and highly-studied anonymity protocol and thus suitable for comparison purposes.

In the dining cryptographers protocol, n cryptographers are having dinner at a round table. After dinner, they are told that the dinner has been paid by their master or one of the cryptographers. They want to know whether the master has paid the dinner or not, without revealing the identity of the payer cryptographer, if the master did not pay. Hence, each cryptographer tosses an unbiased coin and shows the result only to the right cryptographer. If the two coins that a cryptographer can observe are the same, then she announces 'agree'; otherwise, announces 'disagree'. However, if she is the actual payer, then she announces 'disagree' for the same coins and 'agree' for the different ones. If n is odd, then an even number of 'agree's implies that one of the cryptographers has paid, while an odd number implies that the master has paid. The latter is reverse for an even n.

Assume an external attacker (none of the cryptographers or the master) who tries to find out the payer's identity. The external attacker can observe the announcements of the cryptographers. Two cases are assumed for the secret, i.e., the payer:

1. one of the cryptographers, i.e., $Val_{payer} = \{c_1, \ldots, c_n\}$,
2. the master (m, for short) or one of the cryptographers, i.e., $Val_{payer} = \{m, c_1, \ldots, c_n\}$.

For the first case, where the payer is one of the cryptographers, the protocol is secure and there is no leakage. But for the second case, where the master is also a candidate to be the payer, the protocol is insecure [7]. Therefore, it is expected for a security definition to classify the first case as secure and the second case as insecure.

2.3 Information Flow Channels

Information flow channels are mechanisms that transfer secret information to the attacker. There are various types of channels: *direct*, *indirect*, *possibilistic* [6], *termination behavior* [14], *internally observable timing* [27], *probabilistic* [6], and *externally observable timing* [29].

Direct channels occur when the value of a secret variable is directly assigned to a public variable. Indirect channels occur when the control structure of the program reveals secret information. Possibilistic channels occur in concurrent programs when an interleaving of the components results in a direct or indirect channel. Termination channels reveal secret information through the termination or non-termination of program execution. Internally observable timing channels happen when secret information affects the timing behavior of a module, which, through the scheduler, influences the execution order of updates to public variables. Probabilistic channels leak sensitive information through the probabilistic behavior of the program. Externally observable timing channels occur when sensitive information affect the timing behavior of the program.

3 Related Work

In this section, existing definitions of observational determinism are presented and compared to each other. We formalize all these definitions in our program model $\mathcal{M}_\delta^{\mathtt{P}}$ in order to make the comparison and discussion easier. Since we assumed the attacker does not influence the initial values of the public variables, all the initial states are low-equivalent, i.e., $s_0 =_L s_0'$. A concurrent program P under a scheduler δ satisfies observational determinism, according to

– Zdancewic and Myers [32], if and only if all traces of each public variable are prefix and stutter equivalent, i.e.,

$$\forall T, T' \in Traces(\mathcal{M}_\delta^{\mathtt{P}}), l \in L.\;\; T_{|l} \triangleq_p T'_{|l};$$

– Huisman et al. [14], iff all traces of each public variable are stutter equivalent, i.e.,

$$\forall T, T' \in Traces(\mathcal{M}_\delta^{\mathtt{P}}), l \in L.\;\; T_{|l} \triangleq T'_{|l};$$

– Terauchi [31], iff all traces of all public variables are prefix and stutter equivalent, i.e.,

$$\forall T, T' \in Traces(\mathcal{M}_\delta^{\mathtt{P}}).\;\; T_{|L} \triangleq_p T'_{|L};$$

- Huisman and Blondeel [12], Karimpour et al. [15] and Dabaghchian and Abdollahi [9], iff all traces of all public variables are stutter equivalent, i.e.,

$$\forall T, T' \in Traces(\mathcal{M}_{\delta}^{\mathsf{P}}).\ \ T_{|L} \triangleq T'_{|L};$$

- Ngo et al. [20], iff the following two conditions are satisfied:
SSOD-1: $\forall T, T' \in Traces(\mathcal{M}_{\delta}^{\mathsf{P}}), l \in L.\ \ T_{|l} \triangleq T'_{|l};$
SSOD-2: $\forall s_0, s'_0 \in Init(\mathcal{M}_{\delta}^{\mathsf{P}}), \forall T \in Traces(s_0), \exists T' \in Traces(s'_0).\ \ T_{|L} \triangleq T'_{|L};$
- Noroozi et al. [21], iff all paths of all public variables are divergence weak low-bisimilar, i.e.,

$$\forall \pi, \pi' \in Paths(\mathcal{M}_{\delta}^{\mathsf{P}}).\ \ \pi \approx_L^{div} \pi',$$

where divergence weak low-bisimulation ($\approx_L^{div}$) is an equivalence relation that relates two paths that mutually mimic behavior of each other.

Note that all these definitions, except Ngo et al. [20], are scheduler-independent and consider all possible interleavings of the modules. We redefined them in our scheduler-specific model in order to make the comparison easier.

Zdancewic and Myers define observational determinism in terms of prefix and stutter equivalence of traces of each public variable. This definition correctly accepts the first case of the dining cryptographers protocol. However, it incorrectly accepts the second case too. This shows that the definition is too permissive. On the other hand, requiring stutter equivalence of traces of each public variable, as in Huisman et al. [14], is too restrictive and incorrectly rejects the first case of the protocol. Furthermore, requiring traces to agree on the updates to all public variables, as in [9,12,15,21,31], is too restrictive. For example, the first case of the dining cryptographers protocol is incorrectly rejected by all of these definitions. In our experiments with different programs, we found the definition of Ngo et al. [20] the most precise of all. However, SSOD-1 was not permissive enough. For example, it incorrectly rejected the first case of the dining cryptographers.

Observational determinism has also been defined using traces of operations that read or write on public variables, instead of traces of public values. Well-known examples of these definitions are LSOD [10] and its improvements, RLSOD [10] and iRLSOD [6]. These definitions have been implemented in a tool, named JOANA [11], which uses program dependence graphs to model JAVA programs and verify them. JOANA does not explicitly classify variables into public or secret. However, it offers the ability to classify program statements into low (public) or high (secret). Thus, a variable might contain a public value at one point of the program, but a secret value at another point. This allows JOANA to detect intermediate leakages if a statement in intermediate steps is labeled as low. The use of program dependence graphs makes JOANA a scalable tool but reduces its precision. LSOD [10] and its relaxed forms, RLSOD [10] and iRLSOD [6] incorrectly produced security violations for many examples we tried, including the first case of the dining cryptographers protocol. JOANA only works with dependencies and does not take into account concrete values of variables or explicit probability distributions, as in the dining cryptographers protocol. This is an inherent limitation for all analyses that are based on program dependence graphs.

There are probabilistic versions of observational determinism for concurrent programs, such as probabilistic noninterference [19,23,29,30]. These properties match transition probabilities, in addition to the traces. This is a rather strong condition which can detect probabilistic channels but is too restrictive for most cases and programs.

To verify observational determinism, Zdancewic and Myers [32] and Terauchi [31] use type systems, which are widely used to verify secure information flow. However, they are not extensible [2]. They can be defined compositional, but at the cost of either being too restrictive or losing automatic analysis [6].

Huisman et al. [14], Huisman and Blondeel [12], Huisman and Ngo [13] and Dabaghchian and Abdollahi [9] use logic-based methods to specify observational determinism and verify it. These methods build a self-composed model [2] of the program. Then, observational determinism is specified using a program logic, such as CTL*, modal μ-calculus, LTL or CTL. Out of these methods, Huisman and Blondeel [12] and Dabaghchian and Abdollahi [9] have verified the specified property using the model checking tools Concurrency Workbench and SPIN, respectively. In contrast to type systems and program dependence graphs, logical-based methods can specify arbitrarily precise definitions for observational determinism. Most of these methods are compositional. However, they are often non-automatic and require a significant amount of manual effort.

Ngo et al. [20], Karimpour et al. [15] and Noroozi et al. [21] use algorithmic verification methods. These methods mostly model the program as a state transitions system and specify the property using states, paths, and traces of the transition system. Ngo et al. [20] model programs using Kripke structures and use a trace-based method to verify SSOD-1. In order to verify SSOD-2, they determinize the Kripke model and compute a bisimulation quotient of the determinezed model. The time complexity of verifying SSOD-1 is linear in the size of the model, whereas verifying SSOD-2 is exponential. Karimpour et al. [15] and Noroozi et al. [21] compute a weak bisimulation quotient of the model and then verify observational determinism. Algorithmic verification methods make it possible to specify secure information flow with arbitrary precision. They are fully-automatic but generally less scalable, in comparison with type systems and program dependence graphs.

A closely-related filed to secure information flow is quantitative information flow, in which information theory is used to measure the amount of information leakage of a program. If the leakage is computed to be 0, then the program is secure. Many methods and tools are available to compute the information leakage of various programs, including LeakWatch [8], QUAIL [5], HyLeak [4] and PRISM-Leak [24]. LeakWatch [8] estimates leakage of Java programs, including multi-threaded programs and taking into account the intermediate leakages. QUAIL [5] precisely computes leakage at final states of sequential programs, written in the QUAIL imperative language. HyLeak [4] is an extension of QUAIL and combines estimation and precise methods in order to improve the scalability. Finally, PRISM-Leak [24] contains a quantitative package that uses a trace-based method [22] to precisely compute leakage of concurrent probabilistic programs, written in the PRISM language [16]. In quantifying information leakage, PRISM-Leak takes into account the intermediate leakages occurred in the intermediate steps of the program executions.

4 Specifying Observational Determinism

In this section, observational determinism is defined in the Markovian model $\mathcal{M}_\delta^{\mathtt{P}}$. The definition should be able to detect direct, indirect, possibilistic, internally observable timing and termination channels. In order to detect external timing or probabilistic channels, the security property should be strengthened further. For example, to detect external timing channels, the property should require *equivalence* of traces. This makes the property too restrictive, which is the exact opposite of this paper's goal. Therefore, external timing and probabilistic channels are not considered in this paper.

As Ngo et al. [20] discuss, a concurrent program might be secure with a scheduler and insecure with another one and thus defining observational determinism to be scheduler-independent makes the definition imprecise. Therefore, we define observational determinism to be scheduler-specific. Another benefit of a scheduler-specific property is that it is able to find those schedulers that the program is insecure under control of them. This makes the property immune to refinement attacks, in which the attacker selects a scheduler in order to limit the set of possible traces of the program and infer secret information from these traces.

Observational determinism requires a concurrent program to be deterministic to the attacker and produce indistinguishable traces. It demands that low-equivalent inputs produce low-equivalent traces and thus changes in the secret inputs do not change the public behavior. Inspired by Ngo et al. [20], we define observational determinism scheduler-specific and require existential stutter equivalence for traces of all public variables. However, we relax the requirement of stutter equivalence for traces of each public variable to prefix and stutter equivalence in order to improve precision and thus reduce the number of false alarms.

Definition 5. *Let $\mathcal{M}_\delta^{P}$ be a Markov chain, modeling executions of a concurrent probabilistic program $\mathtt{P}$ under the control of a scheduler δ. Formally, $\mathtt{P}$ satisfies* observational determinism*, iff OD_1 and OD_2 hold:*

$$OD_1\colon \forall T, T' \in Traces(\mathcal{M}_\delta^{P}), l \in L.\;\; T_{|l} \triangleq_p T'_{|l},$$
$$OD_2\colon \forall s_0, s'_0 \in Init(\mathcal{M}_\delta^{P}), \forall T \in Traces(s_0), \exists T' \in Traces(s'_0).\;\; T_{|L} \triangleq T'_{|L}.$$

OD_1 requires that traces agree on the updates to each public variable. This requirement enforces deterministic observable behavior and thus the secret data do not affect the public variables.

OD_2 requires that there always exists a matching trace of all public variables for any possible initial states. This requirement results in the independence of the relative ordering of updates to the public variables from the secret values.

Both OD_1 and OD_2 correctly recognize the first case of the dining cryptographers protocol as secure. OD_1 labels the second case secure, but OD_2 labels it insecure. Therefore, our definition of observational determinism correctly recognizes the first case as secure and the second case as insecure.

For an example of an indirect channel, consider the following program, from Huisman et al. [14]

```
P1 ≡ while h>0 do
         l1:=l1+1;
         h:=h-1
        od;
        l2:=1
```

where `h` is a secret variable and `l1` and `l2` are public variables, initially set to 0. The program is insecure, because the final value of `l1` contains the initial value of `h`. It produces the following traces of all public variables

$$\begin{aligned}
\mathtt{h} \le 0 \;\; &: \; [(0,0),(0,1)^\omega],\\
\mathtt{h} == 1 : &\; [(0,0),(1,0),(1,1)^\omega],\\
\mathtt{h} == 2 : &\; [(0,0),(1,0),(2,0),(2,1)^\omega],\\
\mathtt{h} == 3 : &\; [(0,0),(1,0),(2,0),(3,0),(3,1)^\omega],\\
&\vdots
\end{aligned}$$

If we consider the public variables separately, the traces $[0^\omega]$, $[0,1^\omega]$, $[0,1,2^\omega]$, $[0,1,2,3^\omega]$, ... for `l1` and the traces $[0^+,1^\omega]$ for `l2`. Obviously, the separate traces are stutter and prefix equivalent and thus OD_1 holds for this program. However, OD_2 does not hold, because for the trace $[(0,0),(0,1)^\omega]$, for example, there does not exist a stutter equivalent trace of other initial states. Therefore, our definition of observational determinism detects the termination channel of this program.

Allowing prefixing in OD_1 makes it vulnerable to termination channels [14]. However, these channels are detected by OD_2, which requires existential stutter equivalence. For example, consider the following program, which contains a termination channel

```
P2 ≡ if h>0 then l:=1 else S1 fi
```

where

```
S1 ≡ while true do skip od
```

where `l` is a public variable, with the initial value of 0. The attacker can infer truth value of `h>0` from termination of the program. `P2` has the following traces

$$\begin{aligned}
\mathtt{h} > 0 \; &: \; [0,1^\omega],\\
\mathtt{h} \le 0 \; &: \; [0^\omega].
\end{aligned}$$

OD_2 does not hold and thus observational determinism detects this channel.

As an example of a possibilistic channel, consider the following program, from Ngo [18]

```
P3 ≡ [S2 || S3]
```

where

```
S2 ≡ if l=1 then l:=h else skip fi
S3 ≡ l:=1
```

and `l` is initially set to 0 and `||` is the parallel operator with shared variables. The modules `S2` and `S3` are secure if they are run separately. However, concurrent execution of the modules under a uniform scheduler might reveal the whole value of `h`. `P3` under a uniform scheduler has the following traces

$$\texttt{S2}\text{ is executed first: } [0, 0, 1^\omega],$$
$$\texttt{S3}\text{ is executed first: } [0, 1, \texttt{h}^\omega].$$

OD_2 does not hold and thus our definition correctly labels this program as insecure. This example also demonstrates the importance of defining observational determinism scheduler-specific. If `P3` is executed by a scheduler that always picks `S2` first, then the program would be secure. However, it is insecure for a uniform scheduler. Therefore, the security of a concurrent program should be discussed in the context of a given scheduler.

For internally observable timing channels, consider the following program, from Russo et al. [27]

```
P4 ≡ [S4 || S5]
```

where

```
S4 ≡ if h ≥ 0 then skip; skip
               else skip fi;
     l:=1
S5 ≡ skip; skip; l:=0
```

and `l` has the initial value of 0. Under a one-step round-robin scheduler that picks `S5` for the first step, the following traces are produced

$$\texttt{h} \geq 0 \;:\; [0, 0, 0, 0, 0, 0, 1^\omega],$$
$$\texttt{h} < 0 \;:\; [0, 0, 0, 0, 1, 0^\omega].$$

The truth value of $\texttt{h} \geq 0$ is leaked into `l`. OD_2 does not hold and hence observational determinism detects this channel.

5 Verifying Observational Determinism

In this section, two algorithms are proposed for verifying the conditions OD_1 and OD_2. The algorithms take $\mathcal{M}_\delta^{\mathtt{P}}$ as input and return true or false for the satisfaction of the conditions. Both algorithms incorporate a path exploration and trace analysis approach to traverse $\mathcal{M}_\delta^{\mathtt{P}}$ and check the required conditions.

5.1 Verifying OD_1

OD_1 requires that all traces of $\mathcal{M}_\delta^{\mathsf{P}}$ be prefix and stutter equivalent. To verify this, a depth-first exhaustive path exploration of $\mathcal{M}_\delta^{\mathsf{P}}$ is performed and prefix and stutter equivalence is checked between the traces. Once a violation is detected, the algorithm stops the exploration and returns *false*; otherwise, it continues until the exploration is complete and returns *true*.

The detailed steps are outlined in Algorithm 1. For each $l \in L$, the algorithm uses a *witness* stutter-free trace for checking whether prefix and stutter equivalence between the traces holds or not. The witness might be changed to another stutter-free trace running the algorithm. First, an empty string is considered as a witness for each l (lines 1 and 2). Then, $\mathcal{M}_\delta^{\mathsf{P}}$ is explored by a depth-first recursive function, i.e., explorePathsOD1(). In order to explore all reachable states, the function is called for each initial state (lines 4 and 5). It starts from a state and traverses all successors of that state (lines 25–26) until a final state is reached (line 11), which shows that a path has been found. When the algorithm finds a path, for each public variable l (line 12) it performs the following steps to check prefix and stutter equivalence (lines 12–23). It extracts a trace (line 13), removes stutter data from it (line 14) and picks the corresponding witness (line 15). If the witness is longer than the trace, then a prefixing test is done: if the trace is not a prefix of the witness, then a violation of OD_1 has been found and the algorithm returns *false* (lines 16–18). If the witness is shorter than the trace, then the second prefixing test is done: if the witness is not a prefix of the trace, then a violation of OD_1 has been found and the algorithm returns *false*; otherwise (the witness is a prefix of the trace), the trace is longer than the witness and it should be the witness for l (lines 19–23). This process continues until a violation is found, for which *false* is returned; or all paths are explored without finding a violation and *true* is returned.

Time Complexity. The number of possible paths of a DAG can be exponential in the number of its states. This implies that the core of Algorithm 1, i.e., finding all the possible paths using depth-first exploration takes time $O(2^n)$ in the worst case, where n is in the number of states of $\mathcal{M}_\delta^{\mathsf{P}}$. Lines 13–23 for extracting the trace of a path, removing stutter steps and checking prefixing takes $O(n)$ in the worst case. These lines repeat for all $l \in L$ and take time $O(n * |L|)$. Therefore, the worst-case time complexity of Algorithm 1 is exponential in the number of states of $\mathcal{M}_\delta^{\mathsf{P}}$. Note that if the program is insecure, Algorithm 1 does not traverse all the paths and stops as soon as a violation is found. Another point worthy of note is that in most of our experiments, programs had a linear number of paths and a few public variables and hence the total time complexity was linear in the size of $\mathcal{M}_\delta^{\mathsf{P}}$.

Algorithm 1. Verifying OD_1

```
Input: finite MC $\mathcal{M}_\delta^{\mathtt{P}}$
Output: true if the program satisfies $OD_1$; otherwise, false

    // Consider an empty string as a witness for each public variable
 1: for $l$ in $L$ do
 2:     Let $witnesses[l]$ be an empty string;
 3: Let $\pi$ be an empty list of states for storing a path;
 4: for $s_0$ in $Init(\mathcal{M}_\delta^{\mathtt{P}})$ do
 5:     result = explorePathsOD1($s_0$, $\pi$, $witnesses$);
 6:     if not result then
 7:         return false;
 8: return true;

 9: function explorePathsOD1($s$, $\pi$, $witnesses$)
10:     $\pi$.add($s$); // add state s to the current path from the initial state
11:     if $s$ is a final state then // found a path stored in $\pi$
12:         for $l$ in $L$ do
13:             $T_{|l} = trace_{|l}(\pi)$;
14:             Remove stutter data from $T_{|l}$, yielding stutter-free trace $T_{|l}^{sf}$;
15:             $T_w = witnesses[l]$;
16:             if $length(T_{|l}^{sf}) \leq length(T_w)$ then
17:                 if $T_{|l}^{sf}$ is not prefix of $T_w$ then
18:                     return false;
19:             else
20:                 if $T_w$ is not prefix of $T_{|l}^{sf}$ then
21:                     return false;
22:                 else
23:                     $witnesses[l] = T_{|l}^{sf}$;
24:     else
25:         for $s'$ in $Post(s)$ do
26:             result = explorePathsOD1($s'$, $\pi$, $witnesses$);
27:             if not result then
28:                 return false;
29:     $\pi$.pop(); // done exploring from s, so remove it from $\pi$
30:     return true;
```

5.2 Verifying OD_2

OD_2 requires that, given two initial states s_0 and s_0' of $\mathcal{M}_\delta^{\mathtt{P}}$, for each trace of s_0 there exists a stutter equivalent trace of s_0'. This condition can be interpreted as requiring the initial states to have the same set of stutter-free traces:

$$OD_2 : \forall s_0, s_0' \in Init(\mathcal{M}_\delta^{\mathtt{P}}).\;\; Traces_{sf}(s_0) = Traces_{sf}(s_0').$$

where $Traces_{sf}(s_0)$ denotes the set of stutter-free traces of s_0. To verify this, a depth-first exhaustive path exploration of $\mathcal{M}_\delta^{\mathtt{P}}$ is performed to store all the

Algorithm 2. Verifying OD_2

```
    Input: finite MC $\mathcal{M}_\delta^{\mathtt{P}}$
    Output: true if the program satisfies $OD_2$; otherwise, false

 1: Let $\pi$ be an empty list of states for storing a path;
 2: for $s_0$ in $Init(\mathcal{M}_\delta^{\mathtt{P}})$ do
        // Consider an empty set of stutter-free traces for each initial state
 3:     Let $allTraces[s_0]$ be an empty set;
 4:     explorePathsOD2($s_0$, $\pi$, $allTraces$);
 5: for each pair of initial states $(s_0, s_0')$ do
 6:     if $allTraces[s_0]$ != $allTraces[s_0']$ then
 7:         return false;
 8: return true;

 9: function explorePathsOD2($s$, $\pi$, $allTraces$)
10:     $\pi$.add($s$); // add state s to the current path from the initial state
11:     if $s$ is a final state then // found a path stored in $\pi$
12:         $T_{|L} = trace_{|L}(\pi)$;
13:         Remove stutter data from $T_{|L}$, yielding stutter-free $T_{|L}^{sf}$;
14:         $s_0 = \pi[0]$; // initial state of $\pi$
15:         $allTraces[s_0].add(T_{|L}^{sf})$;
16:     else
17:         for $s'$ in $Post(s)$ do
18:             explorePathsOD2($s'$, $\pi$, $allTraces$);
19:     $\pi$.pop(); // done exploring from s, so remove it from $\pi$
20:     return ;
```

stutter-free traces of each initial state in a set and then the equivalence of the sets is checked.

Algorithm 2 shows the detailed steps. It initiates an empty set for each initial state (lines 2–3). Each set will contain stutter-free traces of the corresponding initial state. The set of traces are extracted by the function explorePathsOD2(), which recursively explores all states of $\mathcal{M}_\delta^{\mathtt{P}}$. When a final state is reached (line 11), the trace of the path from the initial state to the final state is extracted (line 12), stutter removed (line 13) and stored in the corresponding set (lines 14–15). After extracting the set of stutter-free traces of all the initial states, the equivalence of the sets is checked (lines 5–7). If they are all equivalent, then the algorithm returns *true*; otherwise, *false* is returned.

Time Complexity. The core of Algorithm 2 is a depth-first exploration to find all paths of $\mathcal{M}_\delta^{\mathtt{P}}$, which takes time $O(2^n)$ in the worst case. The final check for the equivalence of the sets of traces takes worst-case complexity of $O(t^2)$, where t is the number of initial states of $\mathcal{M}_\delta^{\mathtt{P}}$. Therefore, the time complexity of Algorithm 2 is dominated by a depth-first exploration of paths, which is exponential in the size of $\mathcal{M}_\delta^{\mathtt{P}}$. As discussed in the complexity of Algorithm 1, real-world programs in our experiments had linear time complexity and both algorithms showed a high performance in practice.

6 Experimental Evaluation

In this section, the proposed algorithms are compared to other state-of-the-art tools for information flow analysis. The algorithms have been integrated into PRISM-Leak [24], which is a tool to evaluate secure information flow of concurrent probabilistic programs, written in the PRISM language [16]. PRISM-Leak contains two packages, a qualitative package that checks observational determinism using the algorithms of this paper and a quantitative package which measures various types of information leakage using a trace-based algorithm [22].

PRISM-Leak is based on the PRISM model checker [16]. PRISM is a formal modeling and analysis tool for probabilistic and concurrent programs. It has been widely used in many application domains, including security protocols, distributed algorithms, and many others. It uses the PRISM language to describe programs and build Markovian models of them. It builds the models using binary decision diagrams and multi-terminal binary decision diagrams. PRISM-Leak accesses these data structures to create an explicit list of reachable states and a sparse matrix containing the transitions. It then traverses the model based on Algorithms 1 and 2 to check observational determinism. The source codes and binary package of PRISM-Leak are available for download at [24].

The dining cryptographers protocol is used as a comparative case study. As discussed in the related work, other definitions of observational determinism [9,12,14,15,20,31,32] are imprecise for the dining cryptographers protocol. JOANA, a scalable information flow tool, is also imprecise. The only remaining choice for runtime comparison is the quantitative tools that were introduced in the related work: LeakWatch [8], QUAIL [5], HyLeak [4] and PRISM-Leak [24]. These tools compute a leakage of 0 for the first case of the protocol and a leakage greater than 0 for the second case.

We compare the runtime of the proposed algorithms and the quantitative tools in Table 1 for the first case of the dining cryptographers protocol, where the attacker is external and the master is not a candidate of being the payer. Two columns of the table are allocated for PRISM-Leak: the first column, i.e., quantitative method, is for the quantitative package and the second column, i.e., observational determinism, is for the qualitative package which contains the algorithms of this paper. Since QUAIL and HyLeak did not support concurrency, we considered a sequential version of the dining cryptographers, which is available at [24]. Table 2 compares the runtime of the tools for the second case, where the attacker is external and the payer is the master or one of the cryptographers. These run times have been obtained on a laptop with an Intel Core i7-2640M CPU @ 2.80 GHz $\times$ 2 and 8 GB RAM.

As demonstrated by the results in both Tables 1 and 2, the proposed algorithms are faster and more scalable than LeakWatch, QUAIL, and HyLeak and comparable to the quantitative method of PRISM-Leak. In Table 1, the quantitative method of PRISM-Leak is faster than the proposed algorithms, but

Table 1. Runtime comparison of the proposed algorithms to other tools for the first case of the dining cryptographers protocol. Runtime is in seconds and timeout is set to five minutes.

n	LeakWatch [8]	QUAIL [5]	HyLeak [4]	PRISM-Leak [24]	
				Quantitative method [22]	Observational determinism
7	2	1.8	30.5	0.6	**0.7**
8	3.7	3.1	39.7	0.8	**1.2**
9	7.5	6.3	55	1.3	**1.9**
10	15	12.6	72.2	2.9	**3.9**
11	32.2	26.5	97	7.3	**9.6**
12	72.4	62.1	135.4	18.7	**25.2**
13	150.7	151.6	249.3	49.9	**66.7**
14	Timeout	Timeout	Timeout	145.7	**192.4**

in Table 2, algorithms of this paper perform a little better. Note that cores of both qualitative and quantitative packages of PRISM-Leak are the same. They both rely on PRISM to construct the Markov model and use a sparse matrix to access the model transitions. However, the quantitative method traverses the model once to compute the leakage, but the qualitative package traverses it twice (first for OD_1 and second for OD_2). In Table 2, observational determinism does not hold and as soon as the qualitative package discovers that, stops the traversal. This is why the qualitative package outperforms the quantitative method in Table 2.

Table 2. Runtime comparison of the proposed approach to other tools for the second case of the dining cryptographers protocol. Runtime is in seconds and timeout is set to five minutes.

n	LeakWatch [8]	QUAIL [5]	HyLeak [4]	PRISM-Leak [24]	
				Quantitative method [22]	Observational determinism
7	3.1	2.4	30.8	0.6	**0.6**
8	6	4.5	41.7	1	**0.9**
9	12.3	9.7	57	1.5	**1.4**
10	28.2	17.5	75.3	3.5	**3.3**
11	60.5	35	99.3	7.7	**7.4**
12	122.1	78.5	144	20.4	**20.5**
13	Timeout	156.2	277.1	60.5	**58.8**
14	Timeout	Timeout	Timeout	215	**211.8**

Runtime efficiency of PRISM-Leak mainly depends on the size of the Markovian model built by the PRISM model checker and the model size can easily get large. There are a few heuristics that can be performed when writing PRISM descriptions in order to avoid large models. For example, closely-related variables should be defined near each other; or variables that have a relationship with most of the other variables should be defined first in the PRISM descriptions. For more information on these heuristics, please see [25].

7 Conclusion

An algorithmic verification approach was proposed to check secure information flow of concurrent probabilistic programs. Observational determinism was defined as a specification of secure information flow. Comparisons to existing definitions of observational determinism demonstrated that the proposed definition is more precise. Furthermore, two algorithms were proposed to verify observational determinism. The proposed algorithms have been integrated into the PRISM-Leak tool. Experimental evaluations showed promising scalability results for PRISM-Leak.

As future work, we aim to develop symbolic algorithms based on binary decision diagrams to verify observational determinism. This can further improve the scalability of PRISM-Leak. We also aim to use PRISM-Leak to evaluate secure information flow of case studies from newer application domains.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE Workshop on Computer Security Foundations, CSFW 2004, pp. 100–114. IEEE Computer Society (2004)
3. Biondi, F., Enescu, M.A., Heuser, A., Legay, A., Meel, K.S., Quilbeuf, J.: Scalable approximation of quantitative information flow in programs. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 10747, pp. 71–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_4
4. Biondi, F., Kawamoto, Y., Legay, A., Traonouez, L.-M.: HyLeak: hybrid analysis tool for information leakage. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 156–163. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_11
5. Biondi, F., Legay, A., Quilbeuf, J.: Comparative analysis of leakage tools on scalable case studies. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 263–281. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_17
6. Bischof, S., Breitner, J., Graf, J., Hecker, M., Mohr, M., Snelting, G.: Low-deterministic security for low-nondeterministic programs. J. Comput. Secur. **3**, 335–366 (2018)
7. Chaum, D.: The dining cryptographers problem: unconditional sender and recipient untraceability. J. Cryptol. **1**(1), 65–75 (1988)

8. Chothia, T., Kawamoto, Y., Novakovic, C.: LeakWatch: estimating information leakage from Java programs. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8713, pp. 219–236. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_13
9. Dabaghchian, M., Abdollahi Azgomi, M.: Model checking the observational determinism security property using promela and spin. Form. Asp. Comput. **27**(5–6), 789–804 (2015)
10. Giffhorn, D., Snelting, G.: A new algorithm for low-deterministic security. Int. J. Inf. Secur. **14**(3), 263–287 (2015)
11. Graf, J., Hecker, M., Mohr, M., Snelting, G.: Tool demonstration: JOANA. In: Piessens, F., Viganò, L. (eds.) POST 2016. LNCS, vol. 9635, pp. 89–93. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49635-0_5
12. Huisman, M., Blondeel, H.-C.: Model-checking secure information flow for multi-threaded programs. In: Mödersheim, S., Palamidessi, C. (eds.) TOSCA 2011. LNCS, vol. 6993, pp. 148–165. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27375-9_9
13. Huisman, M., Ngo, T.M.: Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 178–195. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31762-0_12
14. Huisman, M., Worah, P., Sunesen, K.: A temporal logic characterisation of observational determinism. In: Proceedings of the 19th IEEE Workshop on Computer Security Foundations, CSFW 2006. IEEE Computer Society (2006)
15. Karimpour, J., Isazadeh, A., Noroozi, A.A.: Verifying observational determinism. In: Federrath, H., Gollmann, D. (eds.) 30th IFIP International Information Security Conference (SEC). ICT Systems Security and Privacy Protection, Hamburg, Germany, Part 1: Privacy, vol. AICT-455, pp. 82–93, May 2015
16. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72522-0_6
17. McLean, J.: Proving noninterference and functional correctness using traces. J. Comput. Secur. **1**(1), 37–57 (1992)
18. Ngo, T.M.: Qualitative and quantitative information flow analysis for multi-thread programs. Ph.D. thesis, University of Twente (2014)
19. Minh Ngo, T., Stoelinga, M., Huisman, M.: Confidentiality for probabilistic multi-threaded programs and its verification. In: Jürjens, J., Livshits, B., Scandariato, R. (eds.) ESSoS 2013. LNCS, vol. 7781, pp. 107–122. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36563-8_8
20. Ngo, T.M., Stoelinga, M., Huisman, M.: Effective verification of confidentiality for multi-threaded programs. J. Comput. Secur. **22**(2), 269–300 (2014)
21. Noroozi, A.A., Karimpour, J., Isazadeh, A.: Bisimulation for secure information flow analysis of multi-threaded programs. Math. Comput. Appl. **24**(2), 64 (2019). https://doi.org/10.3390/mca24020064
22. Noroozi, A.A., Karimpour, J., Isazadeh, A.: Information leakage of multi-threaded programs. Comput. Electr. Eng. **78**, 400–419 (2019). https://doi.org/10.1016/j.compeleceng.2019.07.018. http://www.sciencedirect.com/science/article/science/article/pii/S0045790618331549
23. Noroozi, A.A., Karimpour, J., Isazadeh, A., Lotfi, S.: Verifying weak probabilistic noninterference. Int. J. Adv. Comput. Sci. Appl. **8**(10) (2017). https://doi.org/10.14569/IJACSA.2017.081026

24. Noroozi, A.A., Salehi, K., Karimpour, J., Isazadeh, A.: Prism-leak - a tool for computing information leakage of concurrent probabilistic programs (2018). https://github.com/alianoroozi/PRISM-Leak
25. Parker, D.: Implementation of symbolic model checking for probabilistic systems. Ph.D. thesis, University of Birmingham (2002)
26. Roscoe, A.W.: CSP and determinism in security modelling. In: IEEE Symposium on Security and Privacy, pp. 114–127. IEEE Computer Society (1995)
27. Russo, A., Hughes, J., Naumann, D., Sabelfeld, A.: Closing internal timing channels by transformation. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 120–135. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77505-8_10
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
29. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings 13th IEEE Computer Security Foundations Workshop, CSFW-13, pp. 200–214, July 2000
30. Smith, G.: Probabilistic noninterference through weak probabilistic bisimulation. In: Proceedings of the 16th IEEE Workshop on Computer Security Foundations, CSFW 2003, pp. 3–13. IEEE Computer Society (2003)
31. Terauchi, T.: A type system for observational determinism. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, pp. 287–300. IEEE Computer Society (2008)
32. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: 2003 Proceedings of the 16th IEEE Computer Security Foundations Workshop, pp. 29–43, June 2003. https://doi.org/10.1109/CSFW.2003.1212703