Mathematical and Computational Applications

Article

Bisimulation for Secure Information Flow Analysis of Multi-Threaded Programs

Ali A. Noroozi *, Jaber Karimpour and Ayaz Isazadeh**

Department of Computer Science, University of Tabriz, Tabriz 51666-16471, Iran; karimpour@tabrizu.ac.ir (J.K.); isazadeh@tabrizu.ac.ir (A.I.)

* Correspondence: noroozi@tabrizu.ac.ir

Received: 5 May 2019; Accepted: 3 June 2019; Published: 17 June 2019

Abstract: Preserving the confidentiality of information is a growing concern in software development. Secure information flow is intended to maintain the confidentiality of sensitive information by preventing them from flowing to attackers. This paper discusses how to ensure confidentiality for multi-threaded programs through a property called observational determinism. Operational semantics of multi-threaded programs are modeled using Kripke structures. Observational determinism is formalized in terms of divergence weak low-bisimulation. Bisimulation is an equivalence relation associating executions that simulate each other. The new property is called bisimulation-based observational determinism. Furthermore, a model checking method is proposed to verify the new property and ensure that secure information flow holds in a multi-threaded program. The model checking method successively refines the Kripke model of the program until the quotient of the model with respect to divergence weak low-bisimulation is reached. Then, bisimulation-based observational determinism is checked on the quotient, which is a minimized model of the concrete Kripke model. The time complexity of the proposed method is polynomial in the size of the Kripke model. The proposed approach has been implemented on top of PRISM, a probabilistic model checking tool. Finally, a case study is discussed to show the applicability of the proposed approach.

Keywords: information security; secure information flow; bisimulation; multi-threaded programs

1. Introduction

The increase in number and variety of security attacks on computing systems amplifies the need for improvement in protection mechanisms against the security attacks. Most of these attacks target the confidentiality of sensitive information.

Cryptography, access control and firewalls are common protection mechanisms against the attacks on confidentiality. These mechanisms allow access (read or write) to the secret information for authorized users only. However, once access to secret information is granted for an authorized user, they do not have control over how the information is used. They have no technique to verify and prevent the flow of secret information to unauthorized users. For example, consider an android game with in-app purchase capability. While installing this app, it asks for permission to access the Internet and credit card information (for in-app purchases, like buying additional coins). Once an android user allows these permissions, they have no way to ensure that the credit card information is used in a legal manner and are not sent to an unauthorized server. This is an example of information flows, which potentially result in dangerous types of attacks, such as code injection or sensitive information leakage [1].

To avoid information flows and stop information leakages, *secure information flow* has been introduced. Secure information flow is a security mechanism for verifying and ensuring that secret information in a program does not flow to unauthorized users. It assigns two security levels to program variables: some variables are assigned *public* (low) security level and some are assigned *secret* (high) security level. The attacker, in secure information flow context, is supposed to know the source code of the program and is able to execute the program and observe values of the public variables. Secure information flow seeks to detect information flows from secret variables to public ones [2,3].

For example, consider the program `l:=h`, where `h` is a secret variable and `l` is a public variable. This program has a *direct flow*, since the attacker can infer secret information (`h`) by observing the public variable (`l`). As another example, consider the program `if h>0 then l:=-5 else l:=5`, which has an *indirect flow*. The attacker can infer some secret information (`h` being positive or not) by observing the value of `l`. Verifying confidentiality of multi-threaded programs and ensuring secure information flows is the main motivation of this paper.

To ensure secure information flow in multi-threaded programs, a confidentiality *property* needs to be formalized and a *verification* method is needed to check whether the program satisfies the property or not.

A commonly used confidentiality property to ensure secure information flow for concurrent programs is *Observational determinism* [4–6]. It requires the program to appear deterministic to an attacker capable of observing values of the public variables during the program execution. There are various definitions of observational determinism [6–11]. However, the time complexity of verifying most of these properties is exponential in the size of the state space of the program [10]. Furthermore, they are not restrictive enough in some security contexts to thoroughly avoid information leakages.

In this paper, an automatic approach is proposed to analyze secure information flow for multi-threaded programs. The proposed approach consists of two main parts: (1) a new formalization, *Bisimulation-based Observational Determinism (BOD)*, for specifying secure information flow for multi-threaded programs (Section 4.1) and (2) a polynomial-time algorithm for verifying BOD to ensure that BOD holds in a multi-threaded program (Section 4.2).

Kripke structures are used to model program behavior. Any path in the Kripke structure corresponds to an execution of the program that the Kripke structure is modeling. The sequence of public values of each path denotes a public behavior that is observable by the attacker. We define an equivalence relation, called *divergence weak low-bisimulation*, between paths of the Kripke structure. Divergence weak low-bisimulation requires paths of the program to affect the public variables in the same way. We formalize BOD using divergence weak low-bisimulation and show that a program satisfies BOD, if and only if all paths that have the same initial public value are divergence weak low-bisimilar. Thus, paths that have the same initial public value pass through the same equivalence classes of states but not at the same time, that is, BOD allows some paths to run slower than others. BOD is defined language- and scheduler-independent.

To verify BOD, a sound model checking method is proposed. The method chooses an arbitrary path for all paths having low-equivalent initial states. BOD requires divergence weak low-bisimilarity between each arbitrary path and the corresponding paths in the program. The method computes a divergence weak low-bisimulation quotient of the program, which is an abstract model of the program and consists of equivalence classes under divergence weak low-bisimulation. We show that BOD is satisfied if and only if initial states of each arbitrary path and the corresponding paths in the program are in the same equivalence classes. The time complexity of computing the quotient and hence verifying BOD is polynomial in the size of the program model.

In summary, our contributions include (1) formalizing observational determinism using a bisimulation foundation, (2) a verification algorithm for checking observational determinism.

The remainder of the paper is structured as follows. Section 2 discusses preliminaries and assumptions made throughout the paper. Section 3 discusses previous work. In Section 4, the proposed approach is explained. In this section bisimulation-based confidentiality property is formally defined using low-bisimulation and then the verification is explained in detail. Finally, Section 5 concludes by a discussion on future work.

2. Preliminaries and Assumptions

A security analysis should include a model for *program*, *attacker* and *property* [12]. Program and attacker models along with preliminaries for formalizing the property are explained in this section. The confidentiality property for specifying requirements is defined in Section 4.1.

2.1. Program Model

We model a program as a set of concurrently running threads with a memory shared across the threads. Kripke structures are utilized to model multi-threaded programs and paths are used to model program behavior (executions).

Definition 1 (Kripke structure). *A Kripke structure $\mathcal{K}$ is a tuple $(S, \rightarrow, I, AP, V)$ 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 possible values of the public variables and $V : S \rightarrow 2^{AP}$ is a labeling function.*

$\mathcal{K}$ 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 $Post(s) = \emptyset$. For a Kripke structure modeling a sequential program, terminal states represent the termination of the program.

Definition 2 (Path). *An path fragment $\hat{\pi}$ of $\mathcal{K}$ is a finite state sequence $s_0 s_1 \ldots s_n$ such that $s_i \in Post(s_{i-1})$ for all $0 < i \leq n$ or an infinite state sequence $s_0 s_1 s_2 \ldots$ such that $s_i \in Post(s_{i-1})$ for all $0 < i$. An path of $\mathcal{K}$ is a path fragment that starts in an initial state and is either finite, ending in a terminal state or infinite.*

The first state of the path $\pi = s_0 s_1 s_2 \ldots$ is extracted by $\pi[0]$, that is, $\pi[0] = s_0$. $Paths(s)$ denotes the set of paths starting in s and $Paths(\mathcal{K})$ the set of paths of the initial states of $\mathcal{K}$: $Paths(\mathcal{K}) = \cup_{s \in I} Paths(s)$. Suppose I' is a subset of I; Then, $Paths(I') = \cup_{s \in I'} Paths(s)$.

A *trace* of a path $\pi = s_0 s_1 \ldots$ is defined as $T = trace(\pi) = V(s_0)V(s_1)\ldots$. Two 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 $A_0^+ A_1^+ A_2^+ \ldots$ for $A_0, A_1, A_2, \cdots \subseteq AP$ where A_i^+ is the Kleene plus operation on A_i and is defined as $A_i^+ = \{x_1 x_2 \ldots x_k | k > 0$ and each $x_i = A_i\}$. 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 & stutter equivalent*, if one is stutter-equivalent to a prefix of another.

Two Kripke structure can be combined in a single composite Kripke structure.

Definition 3 (Composite Kripke structure $\mathcal{K}_1 \oplus \mathcal{K}_2$). *For $\mathcal{K}_i = (S_i, \rightarrow_i, I_i, AP, V_i)$, $i = 1, 2$: $\mathcal{K}_1 \oplus \mathcal{K}_2 = (S_1 \uplus S_2, \rightarrow_1 \uplus \rightarrow_2, I_1 \uplus I_2, AP, V)$ where $\uplus$ stands for disjoint union and $V(s) = V_i(s)$ if $s \in S_i$.*

Let us assume $\mathcal{K} = (S, \rightarrow, I, AP, V)$ is a Kripke structure that models executions of a multi-threaded program, with all possible interleavings of the threads. A state of $\mathcal{K}$ indicates the current values of all variables together with the current value of the program counter that indicates the next program statement to be executed. Execution steps of the program are modeled by the transition relation. In case a state has more than one outgoing transition, the next execution step (transition) is chosen in a purely non-deterministic fashion. In consequence, executions result from the resolution of the possible non-determinism in the program. This resolution is performed by a *scheduler*. A scheduler chooses in any state s one of the enabled transitions according to a scheduling policy. Probabilistic choices are not considered, that is, the scheduler is possibilistic. AP is the set of values of the public variables and the function V labels each state with these values. If $\mathcal{K}$ has a terminal state s_n, we include a transition $s_n \rightarrow s_n$, that is, a self-loop, ensuring that the Kripke structure has no terminal state. Therefore, all paths of $\mathcal{K}$ are infinite. It is assumed that the state space of the Kripke structure of the program and the shared memory used by the threads are finite.

As an example of program modeling, consider the following program which consists of two threads

```
l:=0;
l:=1   ||   if l=1 then l:=h                                    (P1)
```

Suppose that `h` is a one-bit secret variable, `l` is a one-bit public variable and `||` is the parallel operator. Two threads are secure if they are executed separately but concurrent execution of the two threads might leak the value of `h` into `l`. The Kripke structure of the program is depicted in Figure 1.

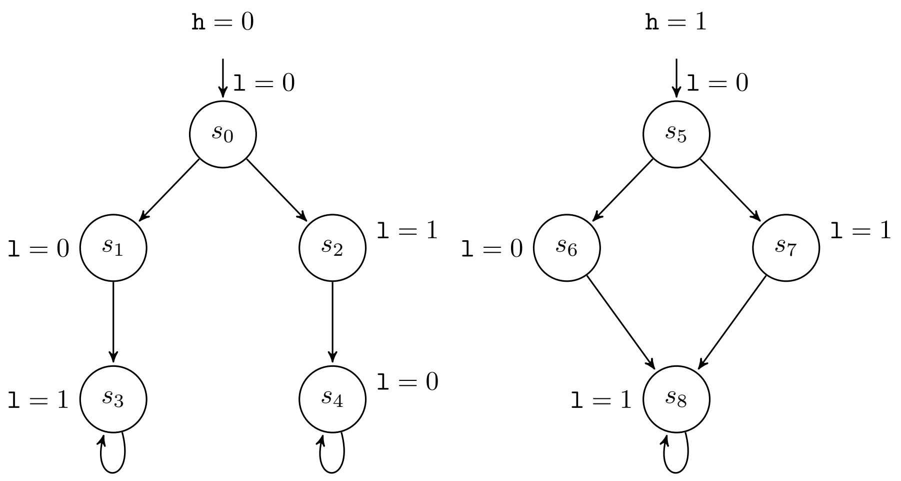

Figure 1. Kripke structure of the program `P1`.

In this Kripke structure, the state space is $S = \{s_0, s_1, \ldots, s_8\}$. The set of initial states consists of two states, that is, $I = \{s_0, s_5\}$. The set of transitions is $\{s_0 \to s_1, s_1 \to s_3, \ldots, s_7 \to s_8, s_8 \to s_8\}$. The set of possible values of the public variable is $AP = \{0, 1\}$. States s_0, s_1, s_4, s_5 and s_6 are labeled with $\{0\}$ and states s_2, s_3, s_7 and s_8 are labeled with $\{1\}$. The set of paths of the program is $\{s_0 s_1 s_3^\omega, s_0 s_2 s_4^\omega, s_5 s_6 s_8^\omega, s_5 s_7 s_8^\omega\}$, where ω denotes *infinite iteration*.

2.2. Attacker Model

Secure information flow is a security mechanism for establishing program confidentiality. The sufficiency of information flow depends on the attacker model. This model defines the capabilities of the attacker, such as being able to observe program output, read program code or even inject code in the program. These capabilities may lead to *information flow channels* that transfer secret information to the attacker. Examples of such channels are *direct*, *indirect*, *termination behavior*, *probabilistic*, *internally observable timing*, *externally observable timing* [13], *injection* and *power* [14].

Direct flows are caused when the value of a secret variable is directly assigned to a public variable. Indirect flows are caused by the control structure of the program. Termination flows leak information through the termination or non-termination of a program execution [3]. Probabilistic channels signal secret information through the probabilistic behavior of the program. Internally observable timing flows are created when secret information influences the timing behavior of a thread, which, through the scheduler, influences the execution order of assignments to public variables [15]. Externally observable timing flows are caused when secret information influences the timing behavior of the program. Injection flows signal information when an attacker is able to inject her desired code in the program [14]. Power channels leak information via the power consumed to execute an action dependent on secret information, assuming the attacker can measure this consumption [3].

Which of these channels are a concern depends strongly on the specific context and on the power of the attacker, that is, on the observations she is able to make. We suppose that the attacker has the knowledge of the program's source code, is able to choose a scheduling policy, run the program and observe program traces.

We assume a shared memory for the multi-threaded program, where public values are stored and the attacker has read access to it. Stating in terms of Kripke structures, the attacker can only observe state labels. We also assume that the attacker can not access other memory areas, including secret variables (i.e., access control and memory protection works correctly). We aim to detect direct, indirect and internal timing leaks.

2.3. Low-Bisimulation

In this section, low-bisimulation and variations of it are formally defined. These definitions will be used in the next section to formalize observational determinism. Execution (path) indistinguishability is a key requirement for preventing information leaks in concurrent programs [6]. Bisimulation is a widely-used foundation for execution indistinguishability and thus for characterizing security for multi-threaded programs (e.g., References [13,16–18]).

We assume the attacker can observe the state labels and cannot distinguish those states that have the same label but might have different secret values. This observational capability of the attacker is formalized by the following relation.

Definition 4 (Low-equivalent states). *A state s is low-equivalent to another state s', written $s =_L s'$, if $V(s) = V(s')$.*

We believe a program is secure if all executions (paths) having low-equivalent initial states are divergence weak low-bisimilar. Thus, we define observational determinism in terms of divergence weak low-bisimulation. For further clarity on the concept of divergence weak low-bisimulation, we first define weak low-bisimulation. Weak low-bisimulation is defined as a relation between states of a Kripke structure and requires for low-equivalent states that each transition is matched by a (suitable) path fragment.

Definition 5 (Weak low-bisimulation $\approx_L$). *Let $\mathcal{K} = (S, \rightarrow, I, AP, V)$ be a Kripke structure. A weak low-bisimulation for $\mathcal{K}$ is a binary relation R on S such that for all $s_1\ R\ s_2$, the following three conditions hold:*

1. $s_1 =_L s_2$.
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 $n \geq 0$ and $s_1\ R\ u_i$, $i = 1, \dots, n$ and $s_1'\ R\ s_2'$.*
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 $n \geq 0$ and $V_i\ Rs_2$, $i = 1, \dots, n$ and $s_1'\ R\ s_2'$.*

States s_1 and s_2 are weak low-bisimilar, denoted $s_1 \approx_L s_2$, if there exists a weak low-bisimulation R for $\mathcal{K}$ with $s_1\ R\ s_2$.

Weak low-bisimulation is defined as a relation between states within a single Kripke structure. An alternative perspective is to consider low-bisimulation as a relation between two Kripke structures. This enables comparing different Kripke structures. Take Kripke structures $\mathcal{K}_1$ and $\mathcal{K}_2$ and combine them in a single composite Kripke structure $\mathcal{K}_1 \oplus \mathcal{K}_2$ (see Definition 3). Then, $\mathcal{K}_1 \approx_L \mathcal{K}_2$ if and only if, for every initial state s_1 of $\mathcal{K}_1$, there exists a weak low-bisimilar initial state s_2 of $\mathcal{K}_2$, and vice versa.

Two Kripke structures can be weak low-bisimilar (initial states can be weak low-bisimilar) but may not produce indistinguishable paths. This is caused by *stutter paths*, that is, paths that stay forever in an equivalence class without performing any visible step. This behavior is called *divergent*. We would like to adapt weak low-bisimulation such that states may only be related if they both exhibit divergent paths or none of them has a divergent path. This yields a variant of weak low-bisimulation called divergence weak low-bisimulation.

Definition 6 (**Divergence weak low-bisimulation $\approx_L^{div}$** [19]). *Let R be an equivalence relation on S. A state $s \in S$ is R-divergent if there exists an infinite path fragment $\pi = ss_1s_2 \ldots \in Paths(s)$ such that $s\ R\ s_j$ for all $j > 0$. 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 $s_1\ R\ s_2$: if s_1 is R-divergent, then s_2 is R-divergent. States s_1, s_2 are divergence weak low-bisimilar, denoted $s_1 \approx_L^{div} s_2$, if there exists a divergence sensitive weak low-bisimulation R such that $s_1\ R\ s_2$.*

We defined $\approx_L^{div}$ between two states. It can also be defined between two paths.

Definition 7 (**Divergence weak low-bisimilar paths** [19]). *For infinite path fragments $\pi_i = s_{0,i}s_{1,i}s_{2,i} \ldots$, $i = 1, 2$ in $\mathcal{K}$, π_1 is divergence weak low-bisimilar to π_2, denoted $\pi_1 \approx_L^{div} \pi_2$ if and only if there exists an infinite sequence of indexes $0 = j_0 < j_1 < j_2 < \ldots$ and $0 = k_0 < k_1 < k_2 < \ldots$ with:*

$$s_{j,1} \approx_L^{div} s_{k,2} \textit{ for all } j_{r-1} \leq j < j_r \textit{ and } k_{r-1} \leq k < k_r \textit{ with } r = 1, 2, \ldots$$

The following lemma asserts that divergence stutter low-bisimilar states produce paths that are divergence stutter low-bisimilar. It also asserts that initial states of divergence stutter low-bisimilar paths are themselves divergence stutter low-bisimilar. The fact that $\approx_L^{div}$ can be lifted from states to paths and vice versa will used in proving the correctness of the verification algorithm in Section 4.2.

Lemma 1. *Divergence weak low-bisimilar states have divergence weak low-bisimilar paths and vice versa:*

$$s_1 \approx_L^{div} s_2 \textit{ iff } \forall \pi_1 \in Paths(s_1)\ (\exists \pi_2 \in Paths(s_2).\ \pi_1 \approx_L^{div} \pi_2)$$

Proof. See Baier and Katoen [19], page 550. □

Divergence weak low-bisimulation is an equivalence and partitions the set of states of a Kripke structure into equivalence classes. The result is called the quotient Kripke structure. For a state set S and an equivalence R, $[s]_R = \{s' \in S \mid (s, s') \in R\}$ defines the equivalence class of the state $s \in S$ under R. The quotient Kripke structure with respect to $\approx_L^{div}$ is defined as follows.

Definition 8 (**Divergence weak low-bisimulation quotient $\mathcal{K}/\approx_L^{div}$**). *The divergence weak low-bisimulation quotient of a Kripke structure $\mathcal{K}$ is defined by $\mathcal{K}/\approx_L^{div} = (S/\approx_L^{div}, \rightarrow', I', AP, V')$, where $S/\approx_L^{div} = \{[s]_{\approx_L^{div}} \mid s \in S\}$, $V'([s]_{\approx_L^{div}}) = V(s)$, $I' = \{[s]_{\approx_L^{div}} \mid s \in I\}$ and $\rightarrow'$ is defined by*

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

3. Related Work

In the literature, there are various definitions of observational determinism. These definitions formalize determinism in terms of various forms of stutter equivalence of the program executions, such as prefix and stutter equivalence on traces of each public variable [6], stutter equivalence on traces of each public variable [7], prefix and stutter equivalence on traces of all public variables together [8], stutter equivalence on traces of all public variables [9,20], scheduler-specific stutter equivalence on traces of each public variable and also traces of all public variables together [10] and equivalence to public operations [11].

For most of these definitions, model checking algorithms have been presented to verify them [7,9,10]. Huisman and Blondeel [9] specify observational determinism in modal μ-calculus. They use Concurrency Workbench (CWB) as the model checking tool. They encode the self-composed program and modal μ-calculus formulation of observational determinism in CWB's specification language. Dabaghchian and Abdollahi [20] specify observational determinism in Linear Temporal Logic. They encode the self-composed program and LTL formula in the PROMELA specification language, using the SPIN model checker. Ngo et al. [10] present two bisimulation-based algorithms to verify observational determinism. These model checking methods have exponential time complexity, while BOD requires polynomial time complexity. BOD is a little more restrictive than these definitions, as it uses divergence weak low-bisimulation (Divergence weak low-bisimulation implies stutter equivalence but the reverse does not necessarily hold (See Baier and Katoen [19], page 549).). This restriction is not bad, as most applications require strict confidentiality properties in order to avoid leakage of sensitive information.

Giffhorn and Snelting [21] define observational determinism in terms of low-equivalence on public operations and present a program analysis algorithm for verifying it. The algorithm models concurrent programs as *dependence graphs*. In a program dependence graph, nodes represent program statements and edges represent data and control dependencies. Giffhorn and Snelting's definition of observational determinism is based on traces which consist of read/write operations and memory values and are enriched with control and data dependencies. This is different from our definition of BOD and many other definitions of observational determinism, in which traces are defined using public variable values.

Another widely-used scheduler-independent property is *strong security*, introduced by Sabelfeld and Sands [13]. They define a partial equivalence relation, called strong low-bisimulation, that relates two multi-threaded programs, with the same number of threads, only if they execute in lock-step and affect the public variables in the same manner. Then, a multi-threaded program satisfies strong security if it is related to itself. Strong security is too strong and requires step-by-step indistinguishability, which implies, many intuitively secure programs are rejected. Compared to strong security, BOD is much more permissive on harmless programs.

Mantel and Sudbrock [16] propose *Flexible Scheduler-Independent (FSI) Security* for multi-threaded programs. They partition threads of the program into high threads that definitely do not modify public variables and public threads that potentially modify public variables. An order-preserving bijection, called low matching, is defined to map the positions of public threads in two multi-threaded programs. Then, they define a partial equivalence relation, called low-bisimulation modulo low matching, which relates two multi-threaded programs, with the same number of threads, if each step of a public thread in one execution is matched by a step of the matching public thread in the other execution. Thus, a multi-threaded program is FSI-secure if it is related to itself. FSI-security is less restrictive than strong security but is language-dependent. Note that BOD is language-independent.

Type systems have been widely used for verifying secure information flow properties [8,13,16,18,22–24]. They are language-dependent, in the sense that a simple language is considered and the confidentiality property is defined using the semantics of this language. Type systems are useful because they support automated, compositional verification. However, they are not extensible, as they are language-dependent and for each change in the program language or the security property, a new type system needs to be defined and proven sound [25]. On the other hand, security requirements are subject to dynamic changes. Accordingly, we use algorithmic verification and model checking, instead of type systems to verify secure information flow.

Another verification approach for information flow properties is to extend temporal logics and introduce new logics to specify these properties. HyperLTL [26] is a recently introduced temporal logic, which is an extension of Linear-time Temporal Logic (LTL) [27]. Runtime verification of information flow properties, including observational determinism, using HyperLTL is discussed in [28,29]. Verifying BOD using HyperLTL would be an interesting future work.

4. The Proposed Approach

The proposed approach for verifying observational determinism consists of two main parts: (1) a new formalization, Bisimulation-based Observational Determinism (BOD), for specifying secure information and (2) an algorithm for verifying BOD. BOD is discussed in Section 4.1 and the verification is explained in Section 4.2.

4.1. Bisimulation-Based Observational Determinism

Observational determinism needs to ensure that as secret inputs change, public behavior of the program remains unchanged. Thus, it requires paths with low-equivalent initial states to be indistinguishable [6]. We characterize indistinguishability of paths by divergence weak low-bisimulation.

Definition 9 (**BOD**)**.** *A multi-threaded program MT satisfies BOD with respect to all public variables, if and only if*

$$\forall\, \pi, \pi' \in Paths(\mathcal{K}).\;\; \pi[0] =_L \pi'[0] \Longrightarrow \pi \approx_L^{div} \pi'$$

where $\mathcal{K}$ denotes the Kripke structure of the program MT, $=_L$ is state low-equivalence relation, and $\approx_L^{div}$ is divergence weak low-bisimulation relation. BOD is acronym for Bisimulation-based Observational determinism.

Stated in words, BOD requires two initial states that have the same public data but different secret data, to produce paths that visit the same sequence of $\approx_L^{div}$-equivalence classes and affect the public variables in the same sequence. This implies weak stepwise indistinguishability, by which direct, indirect and internally observable timing flows are detected. Termination behavior flows are eliminated by adding self-loops to terminal states. As discussed earlier, paths of the program result from the resolution of the *possible* non-determinism in the program and the probability of a path is not considered. This research hence does not consider probabilistic flows. In order to handle externally observable timing flows, the confidentiality property must be too restrictive, ensuring that the execution time does not depend on secret inputs. Requiring a high degree of restriction causes rejection of many intuitively secure programs and harms the precision of the property. That is why BOD does not impose a high degree of restriction. A confidentiality property should be as restrictive as possible in the sense that it does not accept leaky programs and should be permissive enough, such that it does not reject intuitively secure programs.

As an example of information flow analysis using BOD, consider the following program

```
l:=0; h:=l+3; l:=l+1                                                    (P2)
```

The program is intuitively secure because `h` is not read. The Kripke structure of the program is depicted in Figure 2.

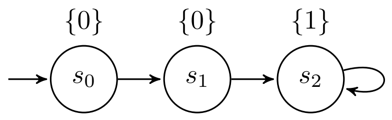

Figure 2. Kripke structure of the program `P2`.

It has just one path $s_0 s_1 s_2^{\omega}$. Thus, `P2` satisfies BOD. As another example, consider the following secure program in which `l` is not updated

```
if l>0 then h:=h+1
       else h:=0                                                        (P3)
```

This program is clearly BOD-secure, because for each value of `l`, the Kripke structure has one state with a self-loop.

For insecure examples, consider the program `P1` (Figure 1). Two threads are secure if they are executed separately. However, concurrent execution of the two threads might leak the value of `h` into `l`. The paths of this program are not divergence weak low-bisimilar and hence not BOD-secure. As another example, consider the following program

```
l:=0; while h>0 do {l++; h--}                                           (P4)
```

This program is insecure and leaks the value of `h` into `l`. The Kripke structure of the program is shown in Figure 3.

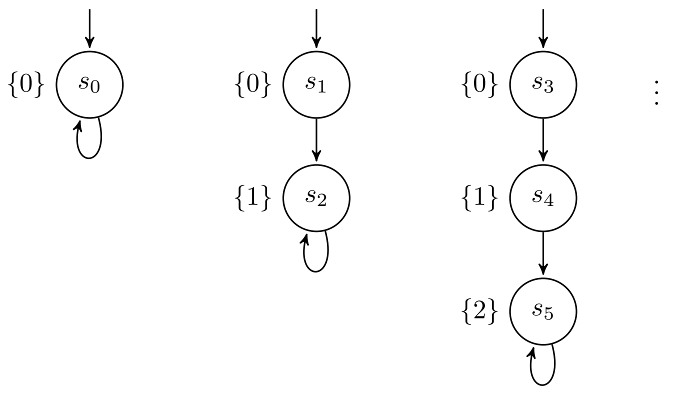

Figure 3. Kripke structure of the program `P4`.

The set of paths is $\{s_0^\omega, s_1 s_2^\omega, s_3 s_4 s_5^\omega, \ldots\}$. These paths are not divergence weak low-bisimilar and thus BOD correctly recognizes the program as insecure. For an example of externally observable timing flow, consider the following program

```
l:=0; if h>0 then l:=1
             else {sleep 100; l:=1}                          (P5)
```

where `sleep 100` means 100 consecutive skip commands. The Kripke structure of the program is depicted in Figure 4.

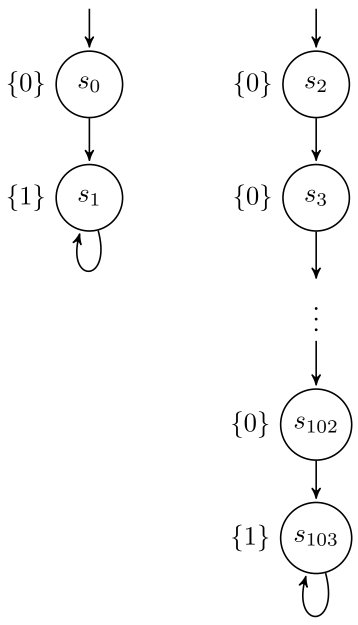

Figure 4. Kripke structure of the program `P5`.

Paths produced are $s_0 s_1^\omega$ and $s_2 s_3 \ldots s_{102} s_{103}^\omega$, which are divergence weak low-bisimilar. Therefore, BOD (incorrectly) classifies the program as secure, while the program has external timing leak. As discussed earlier, BOD (and all other definitions of observational determinism) do not try to detect external timing leaks and avoid them.

For an example of internally observable timing flow, consider the following insecure program

```
l:=0
l:=2   ||   (if h=1 then sleep 100); l:=1                    (P6)
```

A part of the Kripke structure is depicted in Figure 5.

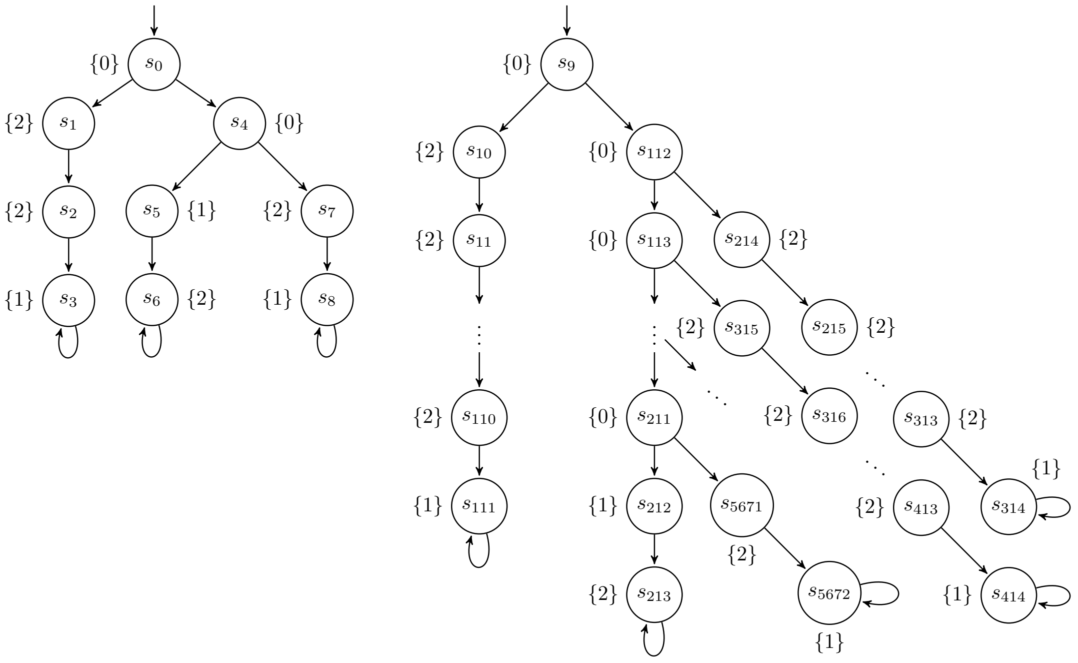

Figure 5. Kripke structure of the program P6.

Due to the fact that, for example, two paths $s_0s_1s_2s_3^\omega$ and $s_0s_4s_5s_6^\omega$ are not divergence weak bisimilar, BOD correctly recognizes the program as insecure.

4.2. Verifying BOD

Here we discuss the proposed model checking algorithm for BOD verification. First, an arbitrary path π_i is extracted for each set of low-equivalent initial states and $\approx_L^{div}$ is checked between the arbitrary path π_i and each path starting in these low-equivalent initial states.

A main step of the verification algorithm is to construct the quotient structure of the program w.r.t $\approx_L^{div}$. Divergence weak low-bisimulation quotient is an abstraction of the Kripke structure and by considering it, enormous state-space reductions may be obtained [19]. We use an abstraction refinement technique to compute the quotient structure.

4.2.1. The Algorithm

The main steps of the verification algorithm are outlined in Algorithm 1.

Algorithm 1 Verification of BOD

```
Input: finite Kripke structure $\mathcal{K}$ modeling the program
Output: true if the program satisfies BOD; Otherwise, false
Partition $I$ into initial state blocks $ISC_0,\ldots,ISC_m$;
Define $ISC = \{ISC_0, \ldots, ISC_m\}$;
Take an arbitrary path $\pi_i \in Paths(ISC_i)$ for each $ISC_i$ $(i = 0, \ldots, |ISC| - 1)$;
Construct Kripke structure $\mathcal{KP}_i$ from $\pi_i$;
Construct Kripke structure $\mathcal{KP}_{min} = \mathcal{KP}_0 \oplus \mathcal{KP}_1 \ldots \oplus \mathcal{KP}_{|ISC|-1}$;
Construct Kripke structure $\mathcal{K}' = \mathcal{K} \oplus \mathcal{KP}_{min}$;
Compute divergence weak low-bisimulation quotient $\mathcal{K}' / \approx_L^{div}$;
for each initial state block $ISC_i$ and the corresponding path $\pi_i$ do
    $class_i = [\pi_i[0]]_{\approx_L^{div}}$;   /* equivalence class of $\pi_i[0]$ w.r.t. $\approx_L^{div}$ */
    if $ISC_i \not\subseteq class_i$ then
        return false
    end if
end for
return true
```

The input of the algorithm is a finite Kripke structure $\mathcal{K} = (S, \rightarrow, I, AP, V)$ modeling the program and the output is *true* or *false*. I is partitioned into sets of low-equivalent initial states called *initial state blocks* $ISC_0,\ldots,ISC_m$ and $ISC = \{ISC_0, \ldots, ISC_m\}$ is defined. An arbitrary path $\pi_i \in Paths(ISC_i)$ is extracted from $\mathcal{K}$ for each ISC_i and a Kripke structure $\mathcal{KP}_i$ is built from π_i. Kripke structures $\mathcal{KP}_i (i = 0, \ldots, |ISC| - 1)$ are combined, forming a single Kripke structure $\mathcal{KP}_{min} = (S_{\mathcal{KP}_{min}}, \rightarrow_{\mathcal{KP}_{min}}, I_{\mathcal{KP}_{min}}, AP_{\mathcal{KP}_{min}}, V_{\mathcal{KP}_{min}})$, where $\mathcal{KP}_{min} = \mathcal{KP}_0 \oplus \mathcal{KP}_1 \oplus \ldots \oplus \mathcal{KP}_{|ISC|-1}$. Kripke structure $\mathcal{K}' = (S', \rightarrow', I', AP, V')$ is also constructed, where $\mathcal{K}' = \mathcal{K} \oplus \mathcal{KP}_{min}$ and quotient of $\mathcal{K}'$ w.r.t. $\approx_L^{div}$ is computed. Then, $\mathcal{K}$ satisfies BOD iff

$$\forall i \in \{0, \ldots, |ISC| - 1\}. \quad ISC_i \subseteq [\pi_i[0]]_{\approx_L^{div}}$$

where $[\pi_i[0]]_{\approx_L^{div}}$ denotes the equivalence class of $\pi_i[0]$ in $\mathcal{K}' / \approx_L^{div}$. Stated in words, $\mathcal{K}$ satisfies BOD if after computing the divergence weak low-bisimulation quotient of $\mathcal{K}' = \mathcal{K} \oplus \mathcal{KP}_{min}$, all initial states of an initial state block and the initial state of the corresponding arbitrary path belong to the same equivalence class.

Now we discuss the main steps of Algorithm 1.

Taking an arbitrary path. To take an arbitrary path from $\mathcal{K}$, a depth-first search is done. The search starts from an initial state and stops at a terminal state (a state with a self-loop).

Computing the divergence weak low-bisimulation quotient. Before explaining how to compute the quotient with respect to $\approx_L^{div}$, some definitions are provided.

Definition 10 (Stutter cycle). *A stutter cycle is a cycle $s_0 s_1 \ldots s_n$ in $\mathcal{K}$ such that $s_0 =_L s_i$ for $i = 1, \ldots, n$.*

Definition 11 (Divergence-sensitive Expansion $\overline{\mathcal{K}}$). *The divergence-sensitive expansion of finite Kripke structure $\mathcal{K} = (S, \rightarrow, I, AP, V)$ is $\overline{\mathcal{K}} = (S \cup \{s_{div}\}, \rightarrow, I, AP \cup \{div\}, \overline{V})$, where $s_{div} \notin S$, $\rightarrow$ extends the transition relation of $\mathcal{K}$ by the transitions $s_{div} \rightarrow s_{div}$ and $s \rightarrow s_{div}$ for every state $s \in S$ on a stutter cycle in $\mathcal{K}$ and $\overline{V}(s) = V(s)$ if $s \in S$ and $\overline{V}(s_{div}) = \{div\}$.*

In order to compute the quotient with respect to $\approx_L^{div}$, $\mathcal{K}'$ is transformed into divergence-sensitive expansion $\overline{\mathcal{K}'}$ such that the equivalence classes under $\approx_L$ in $\overline{\mathcal{K}'}$ coincide with the equivalence classes under $\approx_L^{div}$ in $\mathcal{K}'$. To construct divergence-sensitive expansion, all states on a stutter cycle must be determined, as a transition from each of these states to s_{div} will be added. This is done by finding the strongly connected components (SCCs) in $\mathcal{K}'$ that only contain stutter steps (Transition $s \rightarrow s'$ is a stutter step if $s =_L s'$.). The latter can be carried out using a depth-first search algorithm. Then, computation of the quotient with respect to $\approx_L^{div}$ is reduced to the problem of computing the quotient with respect to $\approx_L$. The algorithm proposed by Groote and Vaandrager [30] is used to compute the quotient of $\overline{\mathcal{K}'}$ with respect to $\approx_L$. The algorithm starts by partitioning the state space based on the low-equivalence relation. Thus, the first partition contains blocks of low-equivalent states. Then, each block is refined based on the set of reachable states from the block. Refinement continues until no refinement is possible and the blocks are stable. For more details of the algorithm please refer to Groote and Vaandrager [30] and Baier and Katoen [19]. Finally, $\mathcal{K}' / \approx_L^{div}$ is obtained from $\overline{\mathcal{K}'} / \approx_L$ by replacing the transitions $s \rightarrow s_{div}$ in $\overline{\mathcal{K}'}$ with self-loops $[s]_{\approx_L^{div}} \rightarrow [s]_{\approx_L^{div}}$ and removing the state s_{div}.

4.2.2. Correctness of the Algorithm

Correctness of Algorithm 1 is proven by the following theorem. Algorithm 1 is correct when it returns true if and only if the input Kripke structure satisfies BOD, otherwise returns false.

Theorem 1. *A Kripke structure $\mathcal{K}$ satisfies BOD iff*

$$\forall i \in \{0, \ldots, |ISC| - 1\}. \quad ISC_i \subseteq [\pi_i[0]]_{\approx_L^{div}}$$

Proof. Given $\mathcal{K} = (S, \rightarrow, I, AP, V)$, BOD requires that all path starting in low-equivalent initial states are divergence weak low-bisimilar. To prove BOD, for each ISC_i ($i = 0, \ldots, |ISC| - 1$) an arbitrary path $\pi_i \in Paths(ISC_i)$ is extracted and divergence weak low-bisimilarity between π_i and each path $\sigma \in Paths(ISC_i)$ is checked. π_i is divergence weak low-bisimilar to σ, if and only if $\pi_i[0]$ is divergence weak low-bisimilar to $\sigma[0]$. The latter fact is developed as a result of Lemma 1. Then, $\pi_i[0]$ is divergence weak low-bisimilar to each $\sigma[0]$, if and only if

$$\forall C \in S' / \approx_L^{div} . \quad \pi_i[0] \in C \Longleftrightarrow \sigma[0] \in C$$

where $S' / \approx_L^{div}$ is the quotient of S' w.r.t $\approx_L^{div}$, which is an equivalence class. Note that all states of an initial state cluster should have divergence weak low-bisimilar paths and hence belong to the same equivalence class. The above condition ensures the latter too.

Thus, BOD requires that all initial states of an initial state block ISC_i and the initial state of the corresponding arbitrary path π_i belong to the same block in $\mathcal{K}' / \approx_L^{div}$. Then, $\mathcal{K}$ satisfies BOD, iff

$$\forall i \in \{0, \ldots, |ISC| - 1\}. \quad ISC_i \subseteq [\pi_i[0]]_{\approx_L^{div}}$$

where $[\pi_i[0]]_{\approx_L^{div}}$ denotes the equivalence class of $\pi_i[0]$ in $\mathcal{K}' / \approx_L^{div}$. □

4.2.3. Complexity of the Algorithm

Assume t is the number of transitions of $\mathcal{K}$. The time complexity of taking an arbitrary path is $O(t + |S|)$. Thus, the time complexity of constructing $\mathcal{KP}_{min}$ is $O(\ |I| * (t + |S|)\ )$. The time complexity of determining SCCs in $\mathcal{K}'$ is $O(t' + |S'|)$, where t' denotes the number of transitions of $\mathcal{K}'$. The quotient space of $\mathcal{K}'$ under $\approx_L^{div}$ can be computed in time $O(\ (|S'| + t') + |S'| * (|AP| + t')\ )$ under the assumption that $t' \geq |S'|$. Thus, the costs of verifying BOD are dominated by the costs of finding the SCCs and computing the quotient space under $\approx_L^{div}$, which are both polynomial-time.

4.2.4. Implementation and Case Study

We have implemented the proposed approach upon the *PRISM model checker* [31]. PRISM is a tool for modeling and analyzing concurrent and probabilistic systems [31]. It includes a language, called the *PRISM language*, which is a state-based language for specifying systems and programs. The PRISM tool uses binary decision diagrams to build a state model of a PRISM program and store states and transitions of the model. A PRISM program can contain a set of modules which run in parallel. The overall model of the program contains all possible transitions and interleavings of the modules. A PRISM program can also contain global variables, which can be accessed and modified by all modules of the program.

We use PRISM modules to represent threads of a multi-threaded program. Global variables represent the shared memory of the multi-threaded program. Then, the state model (Kripke structure) is built as the parallel composition of the modules containing all possible interleavings. We changed parser of PRISM and added two reserved keywords `observable` and `secret` to indicate public (observable) and secret variables.

For evaluation, we take a case study (described below) and specify it in the PRISM language. We run PRISM to build the Kripke structure of the program. We extract the set of reachable states and create a sparse matrix containing the transitions. We then traverse the Kripke, take an arbitrary path for each initial state cluster and add the path to the Kripke with new state numbers. The divergence weak low-bisimulation quotient of the new model is computed and BOD is checked.

As a case study, consider the following program, which is borrowed from [23,32]. The program—which we call the SmithVolpano program—consists of three threads.

```
Thread α:
  while mask != 0 do
    while trigger0 = 0 do; /* busy waiting */
    result := result | mask; /* bitwise ‘or’ */
    trigger0 := 0;
    maintrigger := maintrigger + 1;
    if maintrigger = 1 then trigger1 := 1

Thread β:
  while mask != 0 do
    while trigger1 = 0 do; /* busy waiting */
    result := result & ~mask; /* bitwise ‘and’ with the complement of mask */
    trigger1 := 0;
    maintrigger := maintrigger + 1;
    if maintrigger = 1 then trigger0 := 1

Thread γ:
  while mask != 0 do
    maintrigger := 0;
    if (PIN & mask) = 0 then trigger0 := 1
                        else trigger1 := 1;
    while maintrigger != 2 do; /* busy waiting */
    mask := mask / 2;
    trigger0 := 1;
    trigger1 := 1
```

Assume that `PIN` is a secret variable and `result` is a public variable. If the scheduling of the threads is fair, that is, each thread gets its turn infinitely often and the program starts in an initial state where `maintrigger=0`, `trigger0=0`, `trigger1=0` and `result=0`, the program leaks some bits of `PIN` into `result`. If `PIN` is n bits long and the initial value of `mask` is equal to 2^k ($k < n$), then SmithVolpano leaks $k + 1$ bits of `PIN` into `result`.

The PRISM description for the SmithVolpano program is given in Appendix A. It is composed of three modules `Alpha`, `Beta` and `Gamma`, where each module models a thread. For the sake of brevity, we only allow modules to change turn when they are in the busy waiting state. The global variables `result`, `mask`, `pin`, `trigger0`, `trigger1`, `maintrigger` and `turn` encode the shared variables (memory) of the program. States of the model are represented by the values of its variables, that is, (`result`, `mask`, `pin`, `trigger0`, `trigger1`, `maintrigger`, `turn`, `c1`, `c2`, `c3`).

For $n = 2$, the Kripke structure of the SmithVolpano program has 228 states and 236 transitions. It contains 4 initial states, in which `result` (the public variable) is 0 and `PIN` (the secret variable) varies between 0 and 3. Thus, there is one initial state cluster and a path, containing 46 states, is extracted and added to the Kripke structure. Then, the quotient is computed. It contains 16 blocks. Since the initial state of the extracted path and all initial states of the Kripke do not belong to the same block, then the approach labels the program as BOD-insecure.

Each increase in n (i.e., the bit size of `PIN`) doubles the number of states and transitions of the Kripke structure. However, in all cases the computed quotient contains 16 blocks and the program is labeled as BOD-insecure.

Since the approach uses binary decision diagrams to construct the Kripke model and a sparse matrix is utilized to access the Kripke's transitions, it computes the quotient and verifies BOD in seconds.

To our knowledge, no other evaluations of BOD have been published and we cannot perform a quantitative comparison of our implementation to other algorithms. However, we compare our approach to a closely related work that has been done by Ngo [32]. Ngo discusses the SmithVolpano program as a case study for his work on observational determinism. He formalizes observational determinism by stutter equivalence on traces of each public variable and also traces of all public variables together. Then, he presents two algorithmic verification methods for checking observational determinism. He has implemented his methods on the LTSmin-check tool [33]. As a case study, he specifies the SmithVolpano program in the PRISM language and uses the PRISM tool to export its state machine into three text files. His method then reads the state machine from the text files and uses the LTSmin-check tool to verify observational determinism. The verification algorithms have exponential time complexity, while our algorithm is polynomial. Furthermore, Ngo's method reads the input model from three text files, while our method builds the model on-the-fly using binary decision diagrams. This makes the time complexity of our method far faster than Ngo's methods.

5. Conclusions and Future Work

Aiming at a widely-applicable scheduler-independent analysis for secure information flow, a bisimulation-based foundation was proposed in terms of the semantics of state transition systems. Concretely, BOD was formalized to specify secure information flow by requiring indistinguishability of executions of the program. Indistinguishability was characterized in terms of divergence weak low-bisimulation between paths of the transition system of the program. Then, a model checking algorithm was proposed to verify BOD. The algorithm constructs an abstraction of the input model and then checks BOD on it. We proved the correctness of the algorithm and discussed its time complexity. Finally, the implementation and a case study was discussed. We showed how the proposed approach can be used to model and analyze secure information flow of multi-threaded programs.

As future work, we plan to develop a compositional analysis for multi-threaded programs, as compositionality can be important in making the analysis scale. We believe thread-modular verification [34] is one good candidate for such an analysis. One can also use temporal logics, such as HyperLTL [26], to logically specify BOD and then use model checking methods and tools to check BOD.

Author Contributions: Supervision, J.K. and A.I.; Writing, review and editing, A.A.N.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

The PRISM description for the SmithVolpano program in the case where the initial value of `mask` equals to 2 is given below.

```
dtmc

const int n = 3; // num of bits of the pin variable

global result : [0..pow(2, n)-1];
global mask : [0..pow(2, n)-1];
global pin : [0..pow(2, n)-1];
global trigger0 : [0..1];
global trigger1 : [0..1];
global maintrigger : [0..2];
global turn : [1..3];

module Alpha
    c1 : [0..5];
    [] turn=1 & c1=0 & mask!=0 -> (c1'=1);
    [] turn=1 & c1=1 & trigger0=0 & trigger1=1 -> (turn'=2);
    [] turn=1 & c1=1 & trigger0=0 & trigger1!=1 -> (turn'=3);
    [] turn=1 & c1=1 & trigger0!=0 -> (c1'=2);
    [] turn=1 & c1=2 & mod(floor(result/mask),2)=0 & result+mask<=pow(2,n)-1 ->
                                                        (result'=result+mask) & (c1'=3);
    [] turn=1 & c1=2 & mod(floor(result/mask),2)=1 -> (c1'=3);
    [] turn=1 & c1=3 -> (trigger0'=0) & (c1'=4);
    [] turn=1 & c1=4 & maintrigger<2 -> (maintrigger'=maintrigger+1) & (c1'=5);
    [] turn=1 & c1=5 & maintrigger=1 -> (trigger1'=1) & (c1'=0);
    [] turn=1 & c1=5 & maintrigger!=1 -> (c1'=0);
endmodule

module Beta
    c2 : [0..5];
    [] turn=2 & c2=0 & mask!=0 -> (c2'=1);
    [] turn=2 & c2=1 & trigger1=0 & trigger0=1 -> (turn'=1);
    [] turn=2 & c2=1 & trigger1=0 & trigger0!=1 -> (turn'=3);
    [] turn=2 & c2=1 & trigger1!=0 -> (c2'=2);
    [] turn=2 & c2=2 & mod(floor(result/mask),2)=1 -> (result'=result-mask) & (c2'=3);
    [] turn=2 & c2=2 & mod(floor(result/mask),2)=0 -> (c2'=3);
    [] turn=2 & c2=3 -> (trigger1'=0) & (c2'=4);
    [] turn=2 & c2=4 & maintrigger<2 -> (maintrigger'=maintrigger+1) & (c2'=5);
    [] turn=2 & c2=5 & maintrigger=1 -> (trigger0'=1) & (c2'=0);
    [] turn=2 & c2=5 & maintrigger!=1 -> (c2'=0);
endmodule

module Gamma
    c3 : [0..6];
    [] turn=3 & mask!=0 & c3=0 -> (c3'=1);
    [] turn=3 & mask=0 & c3=0 -> (c3'=5);
    [] turn=3 & c3=1 -> (maintrigger'=0) & (c3'=2);
    [] turn=3 & c3=2 & mod(floor(pin/mask),2)=0 -> (trigger0'=1) & (c3'=3);
    [] turn=3 & c3=2 & mod(floor(pin/mask),2)=1 -> (trigger1'=1) & (c3'=3);
    [] turn=3 & c3=3 & maintrigger=2 -> (c3'=4);
    [] turn=3 & c3=3 & maintrigger!=2 -> 0.5:(turn'=1) + 0.5:(turn'=2);
    [] turn=3 & c3=4 -> (mask'=floor(mask/2)) & (c3'=0);
    [] turn=3 & c3=5 -> (trigger0'=1) & (c3'=6);
    [] turn=3 & c3=6 -> (trigger1'=1) ;
endmodule

init
    mask=2 & result=0 & maintrigger=0 & trigger0=0 & trigger1=0 & c1=0 & c2=0 & c3=0 & turn=3
endinit
```

References

1. Mastroeni, I.; Pasqua, M. Statically Analyzing Information Flows: An Abstract Interpretation-based Hyperanalysis for Non-interference. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, Limassol, Cyprus, 8–12 April 2019; pp. 2215–2223. [CrossRef]
2. Smith, G. Principles of secure information flow analysis. In *Malware Detection*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 291–307.
3. Sabelfeld, A.; Myers, A.C. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **2003**, *21*, 5–19. [CrossRef]
4. McLean, J. Proving noninterference and functional correctness using traces. *J. Comput. Secur.* **1992**, *1*, 37–57. [CrossRef]
5. Roscoe, A.W. CSP and determinism in security modelling. In Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 8–10 May 1995; pp. 114–127.
6. Zdancewic, S.; Myers, A.C. Observational determinism for concurrent program security. In Proceedings of the 16th IEEE Computer Security Foundations Workshop, CSFW'03, Pacific Grove, CA, USA, 30 June–2 July 2003; pp. 29–43.
7. 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'06, Venice, Italy, 5–7 July 2006.
8. Terauchi, T. A type system for observational determinism. In Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF'08, Pittsburgh, PA, USA, 23–25 June 2008; pp. 287–300.
9. Huisman, M.; Blondeel, H.C. Model-checking secure information flow for multi-threaded programs. In Proceedings of the Joint Workshop on Theory of Security and Applications, TOSCA'11, Saarbrücken, Germany, 31 March–1 April 2011; Springer: Berlin/Heidelberg, Germany, 2012; pp. 148–165.
10. Ngo, T.M.; Stoelinga, M.; Huisman, M. Effective verification of confidentiality for multi-threaded programs. *J. Comput. Secur.* **2014**, *22*, 269–300. [CrossRef]
11. Bischof, S.; Breitner, J.; Graf, J.; Hecker, M.; Mohr, M.; Snelting, G. Low-deterministic security for low-nondeterministic programs. *J. Comput. Secur.* **2018**, 1–32.
12. Datta, A.; Franklin, J.; Garg, D.; Jia, L.; Kaynar, D. On Adversary Models and Compositional Security. *IEEE Secur. Priv.* **2011**, *9*, 26–32. [CrossRef]
13. Sabelfeld, A.; Sands, D. Probabilistic noninterference for multi-threaded programs. In Proceedings of the 13th IEEE Workshop on Computer Security Foundations, CSFW'00, Cambridge, UK, 3–5 July 2000; pp. 200–214.
14. Balliu, M. Logics for Information Flow Security: From Specification to Verification. Ph.D. Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2014.
15. Russo, A.; Hughes, J.; Naumann, D.; Sabelfeld, A. Closing internal timing channels by transformation. In Proceedings of the Annual Asian Computing Science Conference, Doha, Qatar, 9–11 December 2006; pp. 120–135.
16. Mantel, H.; Sudbrock, H. Flexible scheduler-independent security. In Proceedings of the 15th European Conference on Research in Computer Security, ESORICS'10, Athens, Greece, 20–22 September 2010; pp. 116–133.
17. Boudol, G.; Castellani, I. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.* **2002**, *281*, 109–130. [CrossRef]
18. Smith, G. Probabilistic noninterference through weak probabilistic bisimulation. In Proceedings of the 16th IEEE Workshop on Computer Security Foundations, CSFW'03, Pacific Grove, CA, USA, 30 June–2 July 2003; pp. 3–13.
19. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT Press: Cambridge, MA, USA, 2008.
20. Dabaghchian, M.; Abdollahi Azgomi, M. Model checking the observational determinism security property using PROMELA and SPIN. *Form. Asp. Comput.* **2015**, *27*, 789–804. [CrossRef]
21. Giffhorn, D.; Snelting, G. A new algorithm for low-deterministic security. *Int. J. Inf. Secur.* **2015**, *14*, 263–287. [CrossRef]
22. Volpano, D.; Irvine, C.; Smith, G. A sound type system for secure flow analysis. *J. Comput. Secur.* **1996**, *4*, 167–187. [CrossRef]

23. 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 Programming Languages, POPL'98, San Diego, CA, USA, 19–21 January 1998; pp. 355–364.
24. Volpano, D.; Smith, G. Probabilistic noninterference in a concurrent language. *J. Comput. Secur.* **1999**, *7*, 231–253. [CrossRef]
25. 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'04, Washington, DC, USA, 28–30 June 2004; pp. 100–114.
26. Clarkson, M.R.; Finkbeiner, B.; Koleini, M.; Micinski, K.K.; Rabe, M.N.; Sánchez, C. Temporal logics for hyperproperties. In Proceedings of the Third International Conference on Principles of Security and Trust, POST'14, Grenoble, France, 5–13 April 2014; pp. 265–284.
27. Pnueli, A. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), Providence, RI, USA, 30 September–31 October 1977; pp. 46–57. [CrossRef]
28. Finkbeiner, B.; Hahn, C.; Stenger, M.; Tentrup, L. Monitoring Hyperproperties. In *Runtime Verification*; Lahiri, S., Reger, G., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 190–207.
29. Hahn, C.; Stenger, M.; Tentrup, L. Constraint-Based Monitoring of Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*; Vojnar, T., Zhang, L., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 115–131.
30. Groote, J.F.; Vaandrager, F. An efficient algorithm for branching bisimulation and stuttering equivalence. In Proceedings of the 17th International Colloquium on Automata, Languages and Programming, Coventry, UK, 16–20 July 1990; pp. 626–638.
31. Kwiatkowska, M.; Norman, G.; Parker, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In Proceedings of the 23rd International Conference on Computer Aided Verification, CAV'11, Snowbird, UT, USA, 14–20 July 2011; Gopalakrishnan, G., Qadeer, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6806, pp. 585–591.
32. Ngo, T.M. Qualitative and Quantitative Information Flow Analysis for Multi-Thread Programs. Ph.D Thesis, University of Twente, Enschede, The Netherlands, 2014.
33. Blom, S.; van de Pol, J.; Weber, M. LTSmin: Distributed and Symbolic Reachability. In *Computer Aided Verification*; Touili, T., Cook, B., Jackson, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 354–359.
34. Flanagan, C.; Freund, S.N.; Qadeer, S. Thread-modular verification for shared-memory programs. In Proceedings of the 11th European Symposium on Programming Languages and Systems, ESOP'02, Grenoble, France, 8–12 April 2002; Springer: Berlin/Heidelberg, Germany, 2002; pp. 262–277.