Model-Checking Secure Information Flow for Multi-threaded Programs*

Marieke Huisman[1] and Henri-Charles Blondeel[2]

[1] University of Twente, Netherlands
[2] INRIA Grenoble - Rhône-Alpes

Abstract. This paper shows how secure information flow properties of multi-threaded programs can be verified by model checking in a precise and efficient way, by using the idea of self-composition.

It discusses two properties that aim to capture secure information flow for multi-threaded programs, and it shows how these properties can be characterised in modal μ-calculus. For this characterisation, a self-composed model of the program is constructed. More precisely, this is a model that contains two copies of the labelled transition system induced by the program, so that the program is executed in parallel with itself. The self-composed model allows to compare two program executions in a single temporal formula that characterises a secure information flow property.

Both the formula and model are translated into the input language for the Concurrency Workbench model checker. We discuss this encoding, and use it for some practical experiments on several simple examples.

1 Introduction

One of the major challenges in the field of application security is multi-threading: the possible interactions between different threads can make the behaviour of an application highly intractable, and therefore multi-threaded applications are notoriously hard to write correctly. Nevertheless, multi-threaded software is omnipresent, and thus the search for formal techniques to establish security properties of multi-threaded software continues. In particular, the following two questions have to be answered: (*i*) what does it mean for a multi-threaded application to respect a security property, and (*ii*) how can we verify this?

This paper concentrates on the latter question: how can we develop a sound and complete technique for the verification of secure information flow (or confidentiality) properties of multi-threaded applications? The most common technique to verify secure information flow properties is to use an information flow type system [18,17,3]; type systems have the advantage that they are efficient, but they are not precise because they use syntactic equalities, and do not consider dependencies between values (see *e.g.*, [1] for more details).

* This work is partially funded by the EC under the IST-FET-2005-015905 Mobius project, and by NWO under the SlaLoM project. Part of the work done while both authors were at INRIA Sophia Antipolis.

S. Mödersheim and C. Palamidessi (Eds.): TOSCA 2011, LNCS 6993, pp. 148–165, 2011.

Therefore, as an alternative approach, the use of self-composition has been advocated. Self-composition recasts the problem of security verification into a standard program verification problem [1,6]. Originally, this was used for the verification of non-interference [8], a technical property that defines secure information flow of *sequential programs*. Traditionally, non-interference is expressed as a property over two program executions. However, if a program is composed with an independent copy of itself – *i.e.*, where all variables are marked to be different – then non-interference can be stated as a safety property over a single execution of this self-composed program. More precisely, suppose we have a statement S with a single low variable l. Non-interference states that if we have two initial states in which l has the same value, then in the final states, after execution of S, l should still have the same value. More formally: S is non-interfering iff $\forall s, s'.s(l) = s'(l) \wedge S(s) \rightsquigarrow t \wedge S(s') \rightsquigarrow t' \Rightarrow t(l) = t'(l)$. This is a property about two program executions, but self-composition allows to express this as a property over a single program execution. Let S' be a copy of S where all variable names are primed. Thus in particular l in S becomes l' in S'. Then we can say that S is non-interfering iff $\{l = l'\}S; S'\{l = l'\}$, *i.e.*, if we have a pre-state where l and l' are equal and we execute first S and then S', then in the post-state l and l' still have to be equal.

This idea has been exploited further for other definitions of secure information flow. Terauchi and Aiken describe how self-composition of sequential programs can be combined with a type system to characterise non-interference relaxed with information downgrading [21]. Huisman *et al.* [11] describe how secure information flow of multi-threaded applications is characterised by a temporal logic formula. The advantage of the self-composition approach is that since the characterisation is exact, soundness and completeness only depends on soundness and completeness of the verification method for the logic. In particular, if secure information flow is characterised by a temporal logic formula, a model checker can be used to automatically verify secure information flow. In that case, the temporal formula expressing the security property should be defined over a model that is the product of two or more basic models representing a program.

The current paper follows up on the earlier paper by Huisman *et al.* [11]. This earlier paper discusses the definition of observational determinism. Observational determinism was introduced by Zdancewic and Myers as a generalisation of non-interference for multi-threaded programs [23]. Huisman *et al.* show that this definition is not precise, as it accepts programs that leak information, and they propose an improved version. This definition has been further improved by Terauchi [20] — this is the definition we will use in this paper[1]. In addition, Huisman *et al.* also show a CTL* formula that precisely characterises the improved definition of observational determinism. However, there are several shortcomings to the approach: the model over which the property is expressed uses a non-standard composition operator to compose the two independent program copies; and in addition there does not exist a ready-made model checker

[1] Terauchi's definition is very restrictive, therefore we have recently proposed an alternative formalisation of observational determinism [10].

for CTL*. To overcome these problems, Huisman *et al.* suggested also characterisations in the modal μ-calculus [12]; however these characterisations turned out not to be correct: they would reject for example a program that looped for ever, while never changing a public variable.

The present paper overcomes these shortcomings as follows:

- It presents a characterisation of observational determinism as proposed by Terauchi [20], in the modal μ-calculus, using a standard composition operator to compose the two program copies;
- It shows that the approach also can be applied to other secure information flow properties, concretely eager trace invariance, as proposed by Roscoe [16];
- The characterisation goes all the way to the model checker: both the program model and the temporal logic formulae are encoded in the input language for the CWB model checker [15];

Several simple example programs are model checked, to show that this approach accepts secure programs that are typically rejected by a type system. From this experience, we draw lessons on what has to be done to make this approach scale to large-scale programs.

Organisation. The remainder of this paper is organised as follows. First, Section 2 introduces the program model. Next, Section 3 presents eager trace invariance and observational determinism. Then, Section 4 discusses their characterisation as temporal logic formulae, and Section 5 discusses how the characterisations are expressed in CWB. Finally, Section 6 concludes, and discusses future work.

Running Example. To illustrate the different definitions and encodings in the paper, throughout we will use the following example programs.

$$h := 0; \texttt{if } (h = 3) \texttt{ then } l := h' \texttt{ else } \epsilon \texttt{ fi} \parallel l := 3 \qquad (\textit{Program 1})$$
$$h := 0; \texttt{if } (h = 3) \texttt{ then } l := h' \texttt{ else } \epsilon \texttt{ fi} \parallel h := 3 \qquad (\textit{Program 2})$$

We use the convention that variables h and h' contain private data, while the value of l is publicly visible. The first program is secure, but to determine this statically, one has to consider that h is always set to 0, thus the value of h' will never be assigned to l. The second program is not secure: in some interleavings variable h', containing private data, is assigned to the publicly visible variable l.

2 Program Model

This section formally defines syntax and semantics of a simple while language with parallel execution. Individual transitions of the operational semantics are assumed to be atomic. Execution is defined as an infinite sequence of configurations, where configurations contain the (remaining) program to be executed and the global memory. Parallel threads communicate via the global memory. For

$$\frac{\langle S_1,\mu\rangle \to \langle \epsilon,\mu'\rangle}{\langle S_1\,;S_2,\mu\rangle \to \langle S_2,\mu'\rangle} \qquad \frac{\langle S_1,\mu\rangle \to \langle S_1',\mu'\rangle}{\langle S_1\,;S_2,\mu\rangle \to \langle S_1';S_2,\mu'\rangle}\ \text{if } S_1' \neq \epsilon$$

$$\frac{\langle S_1,\mu\rangle \to \langle \epsilon,\mu'\rangle}{\langle S_1 \parallel S_2,\mu\rangle \to \langle S_2,\mu'\rangle} \qquad \frac{\langle S_2,\mu\rangle \to \langle \epsilon,\mu'\rangle}{\langle S_1 \parallel S_2,\mu\rangle \to \langle S_1,\mu'\rangle}$$

$$\frac{\langle S_1,\mu\rangle \to \langle S_1',\mu'\rangle}{\langle S_1 \parallel S_2,\mu\rangle \to \langle S_1' \parallel S_2,\mu'\rangle}\ \text{if } S_1' \neq \epsilon \qquad \frac{\langle S_2,\mu\rangle \to \langle S_2',\mu'\rangle}{\langle S_1 \parallel S_2,\mu\rangle \to \langle S_1 \parallel S_2',\mu'\rangle}\ \text{if } S_2' \neq \epsilon$$

$$\langle \texttt{if (}b\texttt{) then } S_1 \texttt{ else } S_2 \texttt{ fi},\mu\rangle \to \langle S_1,\mu\rangle \quad \text{if } b(\mu)$$

$$\langle \texttt{if (}b\texttt{) then } S_1 \texttt{ else } S_2 \texttt{ fi},\mu\rangle \to \langle S_2,\mu\rangle \text{ if } \neg b(\mu)$$

$$\langle \texttt{while (}b\texttt{) do } S \texttt{ od},\mu\rangle \to \langle S\texttt{;while (}b\texttt{) do } S \texttt{ od},\mu\rangle \quad \text{if } b(\mu)$$

$$\langle \texttt{while (}b\texttt{) do } S \texttt{ od},\mu\rangle \to \langle \epsilon,\mu\rangle \text{ if } \neg b(\mu)$$

$$\langle x := E,\mu\rangle \to \langle \epsilon,\mu[x \mapsto E(\mu)]\rangle \qquad \langle \epsilon,\mu\rangle \to \langle \epsilon,\mu\rangle$$

Fig. 1. Operational Semantics

simplicity, we do not consider procedure calls, local memory, or synchronisation between threads. Adding these would add more details to the program model, but not essentially change the technical results (but it might of course influence efficiency and performance of the verification). In particular, the characterisation of observational determinism would not change, but only the possible executions that have to be considered for its verification. To characterise eager trace invariance, the operational semantics is extended with extra information, which can straightforwardly be defined for more complex statements.

2.1 Syntax

First we define the syntax of the programming language. Let *Var* be a set of variables, and $\mathsf{dom}(x)$ the domain of a variable $x \in \mathit{Var}$. Each variable in *Var* has a security-level *high* or *low* assigned to it[2]. This assignment divides the set *Var* into two disjoint subsets H and L, containing the variables with high and low security level, respectively.

We do not give any concrete grammar for expressions; we assume that we can write all the usual side-effect-free boolean and integer expressions. Statements ($\in \mathit{Stmt}$) are defined by the following grammar, where $S \in \mathit{Stmt}$, $x \in \mathit{Var}$, e is any expression, b is a boolean expression, and ϵ is the empty statement.

$$S ::= x := e \mid S;S \mid \texttt{if (}b\texttt{) then } S \texttt{ else } S \texttt{ fi} \mid \texttt{while (}b\texttt{) do } S \texttt{ od} \mid S \parallel S \mid \epsilon$$

2.2 Semantics

Next we define the semantics of the programming language.

[2] As usual, we only consider two security levels, but the approach can easily be generalised to an arbitrary security lattice.

Stores. A store $\in$ *Store* maps *Var* to values, such that each value v belongs to the domain of the corresponding variable x. Formally:

$$\mathit{Store} = \{\mu : \mathit{Var} \to \bigcup_{x \in \mathit{Var}} \mathsf{dom}(x) \mid x \mapsto v\ ,\ v \in \mathsf{dom}(x)\}$$

For $\mu \in \mathit{Store}$, $\mu_{|_L}$ denotes the *restriction* of μ to L, *i.e.*, $\forall x \in L.\mu_{|_L}(x) = \mu(x)$, and $\forall x \in H.\mu_{|_L} x = \bot$. Stores μ and μ' are *L-equivalent*, denoted $\mu \approx_L \mu'$, if $\mu_{|_L} = \mu'_{|_L}$ (*i.e.*, $\forall l \in L.\mu(l) = \mu'(l)$).

Operational semantics. Figure 1 presents the rules of the small step operational semantics of our programming language. Transitions relate program configurations ($\in$ *Conf*), where a configuration $\langle S, \mu\rangle$ consists of a statement S and a store μ. For convenience we use accessor function $\mathsf{store}(\langle S, \mu\rangle) = \mu$. The last (identity) transition rule in Figure 1 applies in case the program has terminated, ensuring that there always is a transition enabled. Thus program behaviour can be considered as a Kripke structure (which makes it suitable for model checking).

Traces. A trace ($\in$ *Trace*) is an infinite sequence of configurations. Given trace $T \in \mathit{Trace}$, T_i denotes the $(i+1)^{th}$ configuration of $T \in \mathit{Trace}$, that is $T = T_0, T_1, \ldots, T_i, T_{i+1}, \ldots$.

Trace T is a *program trace of* S, starting in the initial store μ, denoted $\langle S, \mu\rangle \Downarrow T$, if (*i*) $T_0 = \langle S, \mu\rangle$, and (*ii*) $\forall i \in \mathbb{N}.\, T_i \to T_{i+1}$. Notice that there always is a transition enabled, thus for any initial configuration, an infinite trace exists. Finally, the set of *reachable configurations w.r.t.* a statement S, and a set of stores $\Sigma \subseteq \mathit{Store}$ is formally defined as: $\mathsf{reach}(S, \Sigma) = \{T_i \in \mathit{Conf} \mid \mu \in \Sigma \wedge \langle S, \mu\rangle \Downarrow T \wedge i \in \mathbb{N}\}$.

Example 1. Consider Program 1. Its variables are divided in the sets $H = \{h, h'\}$ and $L = \{l\}$. Suppose we execute this program in initial state $\mu = (h \mapsto 1, h' \mapsto 1, l \mapsto 1)$. A possible execution of this program is (where P_1 denotes the full program):

$$\langle P_1, \mu\rangle \to \langle \texttt{if} \ldots \parallel l := 3, \mu[h \mapsto 0]\rangle \to \langle l := 3, \mu[h \mapsto 0]\rangle \to$$
$$\langle \epsilon, (h \mapsto 0, h' \mapsto 1, l \mapsto 3)\rangle \to \langle \mathit{epsilon}, (h \mapsto 0, h' \mapsto 1, l \mapsto 3)\rangle \to \ldots$$

Two other executions are possible, corresponding to the possible interleavings of the two parallel statements. Considering all these executions results in the set:

$$\begin{aligned}\mathsf{reach}(P_1, \{\mu\}) = \{\ & \langle P_1, \mu\rangle, \langle \texttt{if} \ldots \parallel l := 3, \mu[h \mapsto 0]\rangle, \langle l := 3, \mu[h \mapsto 0]\rangle, \\ & \langle h := 0; \texttt{if} \ldots, \mu[l \mapsto 3]\rangle, \langle \texttt{if} \ldots, (h \mapsto 0, h' \mapsto 1, l \mapsto 3)\rangle, \\ & \langle \epsilon, (h \mapsto 0, h' \mapsto 1, l \mapsto 3)\rangle\}\end{aligned}$$

3 Secure Information Flow

3.1 Eager Trace Invariance

In 1995, Roscoe observed that one way to guarantee that no private data is leaked, is to require that the public data is deterministic [16]. He defined determinism of public data in two ways: (*i*) *eager trace invariance:* the program's

behaviour stripped from all knowledge about private data should be deterministic, or (*ii*) *lazy trace invariance:* the program's behaviour, interleaved with any arbitrary manipulations of private data should be deterministic. In this paper, we further discuss only eager trace invariance[3]. Roscoe's formal definition of eager trace invariance - re-casted for programs - expresses the following: given program P and two sequences of actions (histories) $\mathcal{H}$ and $\mathcal{H}'$ that are equal *w.r.t.* the low actions, *i.e.*, the set of actions associated with low variables, then after P has executed $\mathcal{H}$ or $\mathcal{H}'$, respectively, any possible subsequent sequence of actions should be equal *w.r.t.* the low actions.

To define this formally, we first define the actions of a program. In our program model, parallel threads communicate by reading and writing from the shared store. A sequence of communications describes what a statement "knows" at a particular point, and reading a variable before *or* after a write action on this variable will thus make a difference. Therefore, both read and write actions have to be considered, and we define the following set of *actions* (divided by the security level assignment of variables into Act_L and Act_H):

$$Act = \{\mathsf{write}_{x,v} \mid v \in \mathsf{dom}(x) \wedge x \in Var\} \cup \{\mathsf{read}_{x,v} \mid v \in \mathsf{dom}(x) \wedge x \in Var\}$$

We believe that this choice for the set of actions reflects the definition of eager trace invariance most faithfully in our program model.

Example 2. The different executions of Program 1 from the initial store where all variables are 1 can produce the following actions: $\mathsf{write}_{h,0}, \mathsf{read}_{h,0}, \mathsf{write}_{l,3}$. For Program 2 this would be: $\mathsf{write}_{h,0}, \mathsf{read}_{h,0}, \mathsf{write}_{h,3}, \mathsf{read}_{h,3}, \mathsf{read}_{h',1}, \mathsf{write}_{l,1}$.

To capture the sequence of actions that has been executed, we extend the operational semantics with a history of actions. Single steps can cause multiple actions, or no actions at all to happen, therefore we associate with each step a set of actions. Configurations are extended to the form $\langle S, \mu, \mathcal{H}\rangle$, with accessor function hist, where a history ($\in$ *Hist*) is a sequence of sets of actions. The operational semantics is adjusted to add information to the history; rules that evaluate or write an expression, such as assignment, add new values to the current history.

Example 3. In the extended operational semantics, the first execution of Program 1 becomes (where ϵ is the empty sequence):

$$\begin{aligned}
&\langle P_1, \mu, \epsilon\rangle \rightarrow \langle \texttt{if} \ldots \parallel l := 3, \mu[h \mapsto 0], \{\mathsf{write}_{h,0}\}\rangle \rightarrow \\
&\langle l := 3, \mu[h \mapsto 0], \{\mathsf{write}_{h,0}\}.\{\mathsf{read}_{h,0}\}\rangle \rightarrow \\
&\langle \epsilon, (h \mapsto 0, h' \mapsto 1, l \mapsto 3), \{\mathsf{write}_{h,0}\}.\{\mathsf{read}_{h,0}\}.\{\mathsf{write}_{l,3}\}\rangle \rightarrow \\
&\langle \epsilon, (h \mapsto 0, h' \mapsto 1, l \mapsto 3), \{\mathsf{write}_{h,0}\}.\{\mathsf{read}_{h,0}\}.\{\mathsf{write}_{l,3}\}.\{\}\rangle \rightarrow \ldots
\end{aligned}$$

Reachability is extended in the obvious way, *i.e.*, $\mathsf{reach}(S, \Sigma, \mathcal{H})$ is the set of reachable configurations from S and Σ *whose history equals* $\mathcal{H}$.

[3] Lazy trace invariance can also be model checked, but this requires that a special operation is added to the program model that models the arbitrary manipulation of private data.

Example 4. Consider Program 1 and let *Store* be the set of all possible stores. Then for example:

$$\begin{aligned}
&\mathsf{reach}(P_1, \mathit{Store}, \{\mathsf{write}_{h,0}\}) = \{\langle \texttt{if} \ldots \parallel l := 3, \mu[h \mapsto 0], \{\mathsf{write}_{h,0}\}) \mid \mu \in \mathit{Store}\}\rangle \\
&\mathsf{reach}(P_1, \mathit{Store}, \{\mathsf{write}_{l,3}\}) = \{\langle h := 0; \texttt{if} \ldots, \mu[l \mapsto 3], \{\mathsf{write}_{l,3}\}) \mid \mu \in \mathit{Store}\})\}\rangle \\
&\mathsf{reach}(P_1, \mathit{Store}, \{\mathsf{read}_{h,0}\}) = \{\}
\end{aligned}$$

Two histories $\mathcal{H}_1$ and $\mathcal{H}_2$ are *equivalent w.r.t. a set of actions* A, denoted $\mathcal{H}_1 \equiv_A \mathcal{H}_2$, if they are equivalent up to empty sets, after removing all actions that are not in A. Now we can define eager trace invariance in the context of our program model.

Definition 1 (Eager trace invariance). *Statement S is* eagerly trace invariant *w.r.t. L if*

$$\begin{aligned}
&\forall\, \mathcal{H}, \mathcal{H}' \in \mathit{Hist}. \mathcal{H} =_{\mathit{Act}_L} \mathcal{H}'. \\
&\quad \forall c \in \mathsf{reach}(S, \mathit{Store}, \mathcal{H}). c' \in \mathsf{reach}(S, \mathit{Store}, \mathcal{H}'). \\
&\qquad \forall T \in \mathit{Trace}. c \Downarrow T \Rightarrow \\
&\qquad\quad \exists T' \in \mathit{Trace}. c' \Downarrow T' \ \wedge\ \forall m \in \mathbb{N}.\, \exists n \in \mathbb{N}.\, \mathsf{hist}(T_m) \equiv_{\mathit{Act}_L} \mathsf{hist}(T'_n)
\end{aligned}$$

This definition states the following. Suppose we have two histories $\mathcal{H}$ and $\mathcal{H}'$ that correspond to initial executions of S, *i.e.*, there are configurations c and c' reachable by these histories. Then any possible continuation of c can be matched by a continuation of c' - where matching is understood as that the low actions should coincide.

Notice that configurations c and c' are only constrained by histories $\mathcal{H}$ and $\mathcal{H}'$, not by any initial store.

Example 5. Consider again Program 1. The histories that match on the low actions either (*i*) have no low actions at all, or (*ii*) contain the action $\mathsf{write}_{l,3}$. In case (*i*), any possible continuation will contain the low action $\mathsf{write}_{l,3}$; in case (*ii*), any possible continuation will not produce any low action anymore. Thus the program is eagerly trace invariant.

However, if we consider Program 2, the histories $\mathcal{H} = \mathsf{write}_{h,0}.\mathsf{write}_{h,3}$ and $\mathcal{H}' = \mathsf{write}_{h,3}.\mathsf{write}_{h,0}$ are equivalent *w.r.t.* the low actions (as there are none), but their possible continuations are not. For any initial store μ, the first history leads to the configuration $\langle \texttt{if} \ \ldots, \mu[h \mapsto 3], \mathcal{H}\rangle$ and this will be continued by the actions $\mathsf{read}_{h,3}.\mathsf{write}_{l,h'}$ (for whatever the value of h' is). However, the history $\mathcal{H}'$ leads to a configuration $\langle \texttt{if} \ \ldots, \mu[h \mapsto 0], \mathcal{H}'\rangle$ that will only be continued by the action $\mathsf{read}_{h,0}$. Clearly, these continuations are not equivalent *w.r.t.* the low actions. Thus Program 2 is not eagerly trace invariant.

Notice that if we would change the `then` branch in Program 2 to a statement that would only read the value of l, *e.g.*, $h' := l$, then the program would still not be eagerly trace invariant, because reading of low variables is considered to be a visible action.

3.2 Observational Determinism

Inspired by Roscoe's observation about determinism, Zdancewic and Myers [23] propose that a program has secure information flow if the low traces with public data are independent of the private data, *i.e.*, for any two low-equivalent stores, the traces of low variables are the same, up to stuttering[4]. They call this *observational determinism*.

Several variations of observational determinism have been proposed in the literature. These vary in the definition of low trace equivalence. Zdancewic and Myers define trace equivalence by requiring that the trace for each low variable should be equivalent up to stuttering and prefixing [23]. Later, Huisman *et al.* have shown that this definition is insecure, even for sequential programs [11]. However, Terauchi showed that also Huisman *et al.*'s definition is still insecure: if location traces are considered independently, information can be deduced from the relative order in which two locations are updated. He defines trace equivalence as equality up to stuttering and prefixing of the complete low stores. However, in a forthcoming paper, Huisman and Ngo [10] show that allowing prefixing makes security scheduler-dependent. Therefore, in the definition of observational determinism, we define low trace equivalence, denoted $T \simeq_L T'$ as *equality of the low stores up to stuttering*.

Definition 2. *Statement S is* observationally deterministic *w.r.t. L if*

$$\begin{array}{l}\forall \mu, \mu' \in \mathit{Store}.\, \forall T, T' \in \mathit{Trace}.\\ \quad \mu \approx_L \mu' \;\wedge\; \langle S, \mu\rangle \Downarrow T \;\wedge\; \langle S, \mu'\rangle \Downarrow T' \;\Rightarrow\; T \simeq_L T'\end{array}$$

Example 6. Consider again Program 1. For any two low equivalent stores μ and μ', with initial value l_0 for the variable l, the low store traces are of the following shape: $(l \mapsto l_0) \ldots (l \mapsto 3) \ldots$. Thus clearly, any two traces will be low equivalent, and the program is observationally deterministic.

Consider Program 2. Two low equivalent stores μ and μ' that differ in the value of h' can have traces that are not low equivalent. Suppose that h' is 1 in μ and 2 in μ'. Then a low store trace starting from μ is of the shape $(l \mapsto l_0) \ldots (l \mapsto 1) \ldots$ or $(l \mapsto l_0) \ldots$, while a low store trace starting from μ' is of the shape $(l \mapsto l_0) \ldots (l \mapsto 2) \ldots$ or $(l \mapsto l_0) \ldots$. Thus, clearly not all traces are low store equivalent, and the program is not observationally deterministic.

However, if in Program 2, the `then` branch would be changed to for example $h' := l$ - thus only reading the value of l, then the program would be observationally deterministic. This illustrates the difference with eager trace invariance, where also reading of variables is considered important (*cf.* Example 5).

[4] Two traces are said to be equivalent *up to stuttering* if they are the same if all subsequent duplicates are removed (*e.g.*, $xxyyz$ and $xyyyzzz$ are stuttering equivalent, because in both cases removing the subsequent duplicates results in the trace xyz).

$$
\begin{array}{lll}
s \models^T \mathsf{true} & \stackrel{def}{\Leftrightarrow} \mathit{true} & s \models^T \mathsf{false} \stackrel{def}{\Leftrightarrow} \mathit{false} \\
s \models^T p & \stackrel{def}{\Leftrightarrow} p \in \lambda(s) & \\
s \models^T \neg\Phi & \stackrel{def}{\Leftrightarrow} \neg\,(s \models^T \Phi) & s \models^T \Phi \wedge \Psi \stackrel{def}{\Leftrightarrow} s \models^T \Phi \ \wedge\ s \models^T \Psi \\
s \models^T \langle\alpha\rangle\,\Phi & \stackrel{def}{\Leftrightarrow} \exists s' \in S.\, (s \xrightarrow{\alpha} s' \ \wedge\ s' \models^T \Phi) & s \models^T [\alpha]\,\Phi \stackrel{def}{\Leftrightarrow} \forall s' \in S.\, (s \xrightarrow{\alpha} s' \ \Rightarrow\ s' \models^T \Phi) \\
s \models^T \mu X.\, \Phi & \stackrel{def}{\Leftrightarrow} \exists k \in \mathbb{N}.\, s \models^T \mu X^k.\Phi & s \models^T \nu X.\, \Phi \stackrel{def}{\Leftrightarrow} \forall k \in \mathbb{N}.\, s \models^T \nu X^k.\Phi \\
\mu X^0.\, \Phi & \stackrel{def}{=} \mathsf{false} & \mu X^{k+1}.\, \Phi \stackrel{def}{=} \Phi[\mu X^k.\Phi/X] \\
\nu X^0.\, \Phi & \stackrel{def}{=} \mathsf{true} & \nu X^{k+1}.\, \Phi \stackrel{def}{=} \Phi[\nu X^k.\Phi/X]
\end{array}
$$

Fig. 2. Semantics of modal μ-calculus

4 A Temporal Logic Characterisation of Secure Information Flow

This section first presents the modal μ-calculus [12], the temporal logic used for the characterisation. Then it shows how observational determinism and eager trace invariance are characterised using this logic. The next section shows how the properties and the model are encoded in the input language of the CWB model checker, and uses this to verify secure information flow of some simple examples.

4.1 Modal μ-Calculus

As mentioned above, in earlier work Huisman *et al.* proposed a characterisation of observational determinism, using CTL*. However, no readily available model checker for CTL* exists. Moreover, the characterisation in CTL* used a non-standard composition operator, tailored to the specific property at hand. To make the approach generally applicable, therefore this paper uses the modal μ-calculus [12] instead (whereas the modal μ-calculus characterisation in [11] was not precise enough).

The modal μ-calculus is an extension of Hennessy-Milner logic with fixed-point operators that allow to express recursion. Let $\mathcal{N}$ be a set of variable names, ranged over by X. Let Lab be the set of actions labels, ranged over by α, and let $\mathcal{A}$ be the set of atomic propositions, ranged over by p. Then the syntax of modal μ-calculus formulae is given by the following grammar:

$$\Phi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid X \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \langle\alpha\rangle\,\Phi \mid [\alpha]\,\Phi \mid \mu X.\, \Phi \mid \nu X.\, \Phi$$

Figure 2 defines the semantics of modal μ-calculus formulae, *w.r.t.* a labelled transition system $T = (S, Lab, \rightarrow, A, \lambda)$, where S is the set of states, Lab the set of transition labels, $\rightarrow \subseteq S \times Lab \times S$ the transition relation, A the set of atomic propositions, and $\lambda : S \rightarrow 2^A$ is the valuation, describing for each state which atomic propositions hold. The symbol s ranges over S. The semantics of fixed-point formulae uses (inductively) defined *fixed-point approximants* [5].

4.2 Observational Determinism in Temporal Logic

To characterise observational determinism in the modal μ-calculus, we first define a set of action labels: $Act = \{\mathsf{c}_{x,v} \mid x \in Var \wedge v \in \mathsf{dom}(x)\} \cup \{\tau\}$. Intuitively, a transition is labelled with $\mathsf{c}_{x,v}$ if it changes the value of variable x to the value v. Given a set of variables X, we use c_X to abbreviate the set of labels that encode changes to $x \in X$: $\mathsf{c}_X = \{\mathsf{c}_{x,v} \mid x \in X \wedge v \in \mathsf{dom}(x)\}$.

The operational semantics is updated with these labels: each transition that assigns v to variable x (where v is different from x's former value) is labelled $\mathsf{c}_{x,v}$; all other transitions are labelled with the silent transition label τ. Notice that assignment of a non-changed value is not considered as a change – it will be labelled τ. Sequential and parallel composition propagate transition labels.

Example 7. Consider the example execution of Program 1 in example 1. In the updated operational semantics, this execution becomes:

$$\langle P_1, \mu\rangle \xrightarrow{\mathsf{c}_{h,0}} \langle \texttt{if} \ldots \parallel l := 3, \mu[h \mapsto 0]\rangle \xrightarrow{\tau} \langle l := 3, \mu[h \mapsto 0]\rangle \xrightarrow{\mathsf{c}_{l,3}}$$
$$\langle \epsilon, (h \mapsto 0, h' \mapsto 1, l \mapsto 3)\rangle \xrightarrow{\tau} \langle \epsilon, (h \mapsto 0, h' \mapsto 1, l \mapsto 3)\rangle \xrightarrow{\tau} \ldots$$

We wish to check whether a program is observationally deterministic. In order to do this, we need to compare two program executions. The trick of self-composition is to compose the program with itself in such a way that the execution of the self-composed program corresponds to the two executions of the individual program copies (originally proposed in [1,6]). In our case, we do this by executing the two program copies in parallel. To be able to extract the two program executions, we clearly separate the program configurations of the two programs in every state.

Thus, the self-composed program model is defined as the labelled transition system $T = (S, Lab, \rightarrow, A, \lambda)$, where we define:

- the set of states $S = Conf \times Conf$, *i.e.*, states contain configurations for both program copies,
- the set of action labels $Lab = \{(a)_j \mid a \in Act \wedge j \in \{1, 2\}\}$, where the index j denotes which program copy performs the action,
- the transition relation $\rightarrow \subseteq S \times Lab \times S$ using the labelled operational semantics described above:

$$\frac{c_1 \xrightarrow{a} c_1'}{(c_1, c_2) \xrightarrow{(a)_1} (c_1', c_2)} \qquad \frac{c_2 \xrightarrow{a} c_2'}{(c_1, c_2) \xrightarrow{(a)_2} (c_1, c_2')}$$

- the set of atomic propositions $A = \{\mathsf{eq}_L\}$, and
- the valuation $\lambda : S \rightarrow P(A)$ such that $\mathsf{eq}_L \in \lambda((c1, c2)) \Leftrightarrow \forall l \in L.\mathsf{store}(c1)(l) = \mathsf{store}(c2)(l)$.

Theorem 1. *A program S is observationally deterministic if and only if, for all stores μ and μ',*

$$(\langle S, \mu\rangle, \langle S, \mu'\rangle) \models^T \Phi_{OD}$$

$$\begin{aligned}
\textit{where:}\quad \Phi_{OD} &= \mathsf{eq}_L \Rightarrow \nu X.\mathsf{always}^{(\overline{\mathsf{c}_L}_1)}\,([(\mathsf{c}_L)_1]\,\Upsilon)\\
\Upsilon &= \mathsf{eventually}^{(-)^L_2}\,(\mathsf{eq}_L)\ \wedge \mathsf{always}^{(\overline{\mathsf{c}_L})_2}\,([(\mathsf{c}_L)_2]\,(\mathsf{eq}_L \wedge X))\\
(-)^L_i &= \{(a)_i \mid \exists l \in L.\, a = \mathsf{c}_l \vee a = \tau\}, i \in \{1,2\}\\
(\overline{\mathsf{c}_L})_i &= \{(a)_i \mid a \neq \mathsf{c}_L\}, i \in \{1,2\}\\
\mathsf{always}^A(\phi) &= \nu Y.\phi \wedge (\bigwedge_{a\in A} [a]\,Y) \qquad \mathsf{eventually}^A(\phi) = \mu Y.\phi \vee (\bigwedge_{a\in A} [a]\,Y)
\end{aligned}$$

Proof. For space reasons, we refer to Blondeel's Master's thesis [2] for the proof of this theorem[5].

Intuitively, formula Φ_{OD} expresses that if the low stores of the two program copies are the same (eq_L), then the trace corresponding to the transitions of the first part and the trace corresponding to the transitions of the second part are stuttering equivalent. Stuttering equivalence says that whenever the first part changes a variable in L ($[(\mathsf{c}_L)_1]\ldots$), then Υ has to hold, expressing that: (*i*) there is always a point reachable where the second program copy will change a variable in L such that the low stores become equal again ($\mathsf{eventually}^{(-)^L_2}(\mathsf{eq}_L)$), and (*ii*) if the second program copy is the only one to take transitions and those transitions do not change low variables, ($\mathsf{always}^{(\overline{\mathsf{c}_L})_2}(\ldots)$), then after the second program copy changes a low variable for the first time ($[(\mathsf{c}_L)_2]\ldots$), the two stores will be equal and the whole formula will hold again ($\mathsf{eq}_L \wedge R$).

4.3 Eager Trace Invariance in Temporal Logic

In a similar spirit, eager trace invariance can be characterised. However, this requires to compose the program with itself *thrice*: a program is eager trace invariant if for every two executions that have the same initial low actions, there exists a third execution that performs *all* initial actions of the second execution and then mimics all future low actions of the first execution. This makes it necessary that the initial store of the third model can remain *undetermined* for a while, therefore we add an uninitialised store $\perp$ to the model, defining $\mathit{Conf}_\perp = \mathit{Stmt} \times (\mathit{Store} \cup \{\perp\})$, together with an explicit initialisation label init.

The temporal logic characterisation of eager trace invariance does not use atomic propositions, so the model is of the form $(S, \mathit{Lab}, \rightarrow)$, where

- states $S = \mathit{Conf}_\perp \times \mathit{Conf}_\perp \times \mathit{Conf}_\perp$,
- labels[6] $\mathit{Lab} = \{\tau\} \cup \{(a)_j \mid a \in \mathit{Act} \cup \{\mathsf{init}\} \wedge j \in \{1,2,3\}\}$, and
- transitions $\rightarrow$ are defined as the obvious lifting of the standard operational semantics, extended with explicit initialisation.

The formula abstracts away from the particular kind of high transitions that occur. To model this, we define a so-called high transition relation $\Rightarrow_H$, with corresponding modalities $\langle\!\langle a \rangle\!\rangle_H$ and $[\![a]\!]_H$, respectively, as a variation of standard

[5] In fact, this is a proof for the case where the location traces have to be stuttering equivalent, instead of the complete traces - but the main structure of the proof remains unchanged.

[6] Where *Act* is as defined in the definition of eager trace invariance, page 153.

weak transitions and modalities (that abstract over internal transitions). Let a_l be a low action label, and let $\overset{(a_h)_j}{\Rightarrow}$ be the standard weak transition relation. Then the high transition relation $\Rightarrow_H$ is defined as follows.

$$s \overset{\tau}{\Rightarrow}_H s' \Leftrightarrow s(\Rightarrow_{H'})^* s' \qquad s \Rightarrow_{H'} s' \Leftrightarrow \exists a_h \in Act_H.\exists j \in \{1,2,3\}.s \overset{(a_h)_j}{\Rightarrow} s'$$
$$s \overset{a_l}{\Rightarrow}_H s' \Leftrightarrow s \Rightarrow_H \overset{a_l}{\Rightarrow} \Rightarrow_H s'$$

Now we can characterise eager trace invariance as well in modal μ-calculus.

Theorem 2. *A program S is eager invariant if and only if*

$$(\bot, \bot, \bot) \models^{T_S} [(init)_1]\,[(init)_2]\,\Phi_{ETI}$$

where

$$\begin{aligned}
\Phi_{ETI} = {} & [\mathsf{init}_1]\,[\mathsf{init}_2]\,(\nu X.\,\langle \mathsf{init}_3\rangle\, mimic_{3,1} \wedge \bigwedge_{a_h \in Act_H} [\![(a_h)_1]\!]\, X \\
& \wedge \bigwedge_{a_h \in Act_H} [\![(a_h)_2]\!]\,\langle \mathsf{init}_3\rangle\,\langle\!\langle (a_h)_3 \rangle\!\rangle\, \Psi \\
& \wedge \bigwedge_{a_l \in Act_L} [\![(a_l)_1]\!]\,[\![(a_l)_2]\!]\,\langle \mathsf{init}_3\rangle\,\langle\!\langle (a_l)_3 \rangle\!\rangle\, \Psi) \\
\Psi = {} & \nu Y.\, mimic_{3,1} \wedge \bigwedge_{a_h \in Act_H} [\![(a_h)_1]\!]\, Y \wedge \bigwedge_{a_h \in Act_H} [\![(a_h)_2]\!]\,\langle\!\langle (a_h)_3 \rangle\!\rangle\, Y \\
& \wedge \bigwedge_{a_l \in Act_L} [\![(a_l)_1]\!]\,[\![(a_l)_2]\!]\,\langle\!\langle (a_l)_3 \rangle\!\rangle\, Y \\
mimic_{3,1} = {} & \nu Z.\, \bigwedge_{a_l \in Act_L} [\![(a_l)_1]\!]_H\,\langle\!\langle (a_l)_3 \rangle\!\rangle_H\, Z
\end{aligned}$$

Proof. See Blondeel's Master thesis [2] for the proof.

Formula $mimic_{3,1}$ expresses that for all histories generated by model 1, model 3 can generate a history which is low equivalent. Formula Φ_{ETI} and Ψ are identical, except that Ψ assumes that the store of the third model is already initialised. Intuitively, we loop in Φ_{ETI} until init_3 has happened, and then we loop in Ψ. Formula Φ_{ETI} and Ψ define all states where $mimic_{3,1}$ should hold. These are all states where (*i*) model 1 and 2 have communicated low equivalent histories, and (*ii*) model 3 has communicated exactly the same history (including high actions) as model 2. In other words, formula Φ_{ETI} and Ψ express that as long as model 1 and model 2 have low equivalent histories (*i.e.*, one of them does a high action, or they do the same low action), model 3 can reproduce the actions that model 2 has done so far (including high actions), and then mimic model 1 in its future low actions.

5 Encoding in the Concurrency WorkBench

As mentioned above, in earlier work, Huisman *et al.* characterised observational determinism using CTL* [11]. However, there is no readily available model checker for CTL*, therefore they experimented with Evaluator in the CADP tool set [7] to model check the property. But since Evaluator only supports alternation-free modal μ-calculus, while observational determinism (as defined

by Huisman) only can be expressed as a μ-calculus formula with alternation of greatest and least fixed points, only a stronger property could actually be verified. Thus, it is preferable to use a model checker that supports full modal μ-calculus, such as Concurrency WorkBench (CWB) [15]. This expressiveness is needed, because the properties typically express requirements such as: if one model can do a certain step, the other model (the program copy) has to be able to mimic this step.

We encode our program model and the modal μ-calculus formulations of observational determinism and eager trace invariance in CWB's specification language. The encoding is quite straightforward, to be able to quickly get experimental results.

CWB allows to define agents (or processes) in basic CCS, the Calculus of Communicating Systems [14]. CWB's specification language is quite restrictive, and it does not provide any support for data. Thus there are no parametrised actions, nor conditional statements, and we have to use basic CCS agents to update and lookup variables.

In CCS, when a process performs action a, some parallel process or the environment must simultaneously perform a co-action $'a$ (a corresponds to *receiving* on channel a and $'a$ corresponds to *sending* on channel a). If $'a$ is performed by a parallel process, then a and $'a$ together form a silent action τ. This action corresponds to an *internal choice*, and it is ignored by the weak modalities $\langle\!\langle \alpha \rangle\!\rangle$ and $[\![\alpha]\!]$ of the modal μ-calculus. Internal actions are used to control the behaviour of the agents. All other actions communicate with the environment (*external choices*). For each model, we have exactly one input action: input-mi for model i. After this action, all variables in model i are initialised. The other actions, with "output" in their name, denote a message that is sent to the environment.

Observational determinism and trace invariance assume different actions, therefore we have to give different CCS models. In the sequel, we describe the most important aspects of the modelling of observational determinism. The modelling of eager trace invariance uses a similar approach (and reuses part of the CWB modelling for observational determinism); we refer to Blondeel's Master thesis [2] for details about this.

5.1 CWB Encoding of the Program Model

The first step to encode the program model is to model the store using CCS agents. Each agent is of the form $x{-}v{-}\mathsf{m}i$, where x is a variable, i a program copy number and v a value in the (finite) domain of x. It is necessary to enumerate all possible values, because CCS can not be parametrised with data. Each agent can output the value, either to the environment, or internally. These actions return the original agent. Further, we model updates, that return a different agent, related with the new value of the variable. Every change is output, both externally and internally. The internal communication ensures that the model of the store is updated. As the updates consist of several actions, we have to ensure that the variable cannot be changed in between. To do this, we introduce 'begin' and 'end' labels for variable updates, that ensure that each complete update is

executed atomically. Consequently, the properties that we want to verify have to be adapted for this: instead of checking for a single transition that corresponds to a variable change, they have to match pairs of labels.

Also the individual transitions in the operational semantics (Figure 1) are not atomic in the CCS model. To ensure atomicity of the steps in the operational semantics, a special lock is defined per program model. Each transition in the program model first acquires the lock, then executes the corresponding CCS actions, and then releases the lock. In each model, we have one agent for the assignment of a constant (**AssignValue-m**i) and one agent for the assignment of the value stored in another variable (**AssignVar-m**i), for example:

agent **AssignValue-m**i(output-begin-change-x-to-v_1-mi, change-x-v_1-mi,
value-x-v_1-mi, value-x-v_2-mi, Follow-mi) =
takeLock-mi. (value-x-v_2-mi. 'output-begin-change-x-to-v_1-mi.
'change-x-v_1-mi. 'output-end-change-mi. +
value-x-v_1-mi. 'output-nochange-mi).
'releaseLock-mi.Follow-mi;

This agent should be understood as follows: first the lock for model mi is acquired. If the current value of x in the model mi is v_2, then a change to the value v_1 is communicated (both internally and externally), and then the change has finished. If the value of x is already v_1, then no change is communicated. Then the lock is released, and the remainder of the model mi is executed.

All transitions of the operational semantics are modelled, except for those when the program is terminated; this case is handled by the encoding of observational determinism. Each program copy is modelled as the parallel composition of agents modelling the program, the store and the lock mechanism. The complete program is modelled as the parallel composition of two copies of such models.

Example 8. Consider again Program 1. Using our CWB encoding, the first program copy is modelled as the following CCS agent. Notice that instead of using integer values we explicitly encode Boolean values because we do not have any data in CCS, and the modelling is intended as a proof of concept. All text preceded by `*` are comments:

```
agent Pr1-m1 =
  * h := false
  (AssignValue-m1( c-h-false-out-m1,            * output-begin-change-x-to-v1-m1
                   c-h-false-m1,                * change-x-v1-m1
                   v-h-false-m1, v-h-true-m1, * value-x-v1-m1, value-x-v2-m1
  * if(h = true) then ... else ϵ fi
  If-m1( v-h-true-m1, v-h-false-m1, * then-condition, else condition
      * l := h', then branch
      AssignVar-m1( c-l-true-out-m1, c-l-true-m1, c-l-false-out-m1, c-l-false-m1,
                    v-l-true-m1, v-l-false-m1,
                    v-hprime-true-m1, v-hprime-false-m1, 0),
      0))) | * else branch
  * l := true
  (AssignValue-m1( c-l-true-out-m1, c-l-true-m1, v-l-true-m1, v-l-false-m1, 0));
```

This is executed in parallel with the locking mechanism to make the transition steps of the program copy atomic and with the agent modelling the store of the first program copy, after hiding the internal communication actions. Together this results in the agent describing the program model for the first program copy.

agent **ExPr1-m1** =
(**Lock-m1** | **StoreLHHprime-m1** | **Pr1-m1)** \ **InternActions-m1** ;

Program copy 2 is exactly the same, with all **m1** replaced by **m2**. Their parallel composition - the program model of the self-composed program - is then defined as **ExPr1-m1**|**ExPr1-m2**.

5.2 CWB Encoding of Observational Determinism

To model the observational determinism property in CWB, we first model equality of variables $x \in Var_L$. Because we currently only encode Boolean values, it is sufficient to check whether x in m1 is true if and only if x in m2 is true. This results in the following property definition for **Eq** (where T is CWB notation for *true*, and & is conjunction):

prop **Eq** =
$\bigcup_{x \in Var_L}$ (⟨ 'output-value-x-*true*-m1⟩T ⇒ ⟨ 'output-value-x-*true*-m2 ⟩ T) &
(⟨ 'output-value-x-*true*-m2⟩T ⇒ ⟨ 'output-value-x-*true*-m1 ⟩ T);

To handle termination according to the operational semantics, we express explicitly when a model mi cannot do any action corresponding to the labelled transitions by defining a set **ProgressActions-m**i. We explicitly add a liveness requirement ∼**CanHoldBeforeEnd-m**i, ensuring that there is no path on which Phi always holds until the program terminates (where ∼ is CWB notation for negation, and | for disjunction).

prop **Finished-m**i = [[**ProgressActions-m**i]]F;
set **ProgressActions-m**i =
{ 'output-begin-change-x-to-v-mi | $x \in Store \wedge v \in \mathsf{dom}(x)$} ∪
{ 'output-end-change-mi, 'output-nochange-mi};
prop **CanHoldBeforeEnd-m**i(Phi) =
min(X. (Phi & **Finished-m**i**)** | (Phi & ⟪**ProgressActions-m**i⟫X));

Now we can model observational determinism and its subexpressions.

prop **ObervationalDeterminism** = [[init-m1]][[init-m2]]**Eq** ⇒ **TraceInd**;
prop **TraceInd** = max(R. Always-x-m1(
[[**BeginChangeLowActions-m**1]] [[**'output-end-change-m**1]]
Eventually-m2(**Eq**) & ∼**CanHoldBeforeEnd-m**2(∼**Eq**) &
Always-x-m2([[**BeginChangeLowActions-m**2]]
[[**'output-end-change-m**2]](**Eq** & R))
set **BeginChangeLowActions-m**i =
{ 'output-begin-change-x-to-v-mi | $x \in Store_L \wedge v \in \mathsf{dom}(x)$}

set **Compl-change-x-mi** =
{ 'output-begin-change-y-to-v-mi | $y \in \mathit{Store} - \{x\} \wedge v \in \mathsf{dom}(x)\}$ ∪
{ 'output-end-change-mi, 'output-nochange-mi};
prop **Always-x-mi**(Phi) = max(X. Phi & [[**Compl-change-x-mi**]]X);
prop **Eventually-mi**(Phi) = min(X. Phi | [[**ProgressActions-mi**]]X);

We have verified this property on several simple example programs, including running examples Program 1 and Program 2. Program 1 is observationally deterministic, but typically rejected by a type checker because of the information-leaking then-branch that depends on a private variable - even though the condition will never be true, thus the then branch will never be executed. This is correctly accepted by CWB. Program 2 is not observationally deterministic, and this is indeed rejected by CWB. We have tried the model checker on about 20 small example programs. In all cases, the model checker returns the (correct) answer within milliseconds.

To try the encoding on more realistic examples, the encoding has to be improved, because we would need more than just Boolean values.

6 Conclusions and Future Work

This paper describes a practical exercise in using the self-composition approach to model check secure information flow for multithreaded programs. Concretely, we show how eager trace invariance, proposed by Roscoe [16], and observational determinism, in the version of Terauchi [20], can be characterised as temporal logic formulae and encoded in the Concurrency WorkBench [15]. The encoding can be used to check security of several simple example programs, including examples that would be rejected by a type checker.

As future work, we plan to make the approach scale. For this, we need to improve the modelling of the program model, without an explicit encoding of the data domain. We will study whether parametrised boolean equation systems [4,9] are appropriate for this. If so, we will develop a translation from a program in a general-purpose programming language into such a system.

The properties that we studied in this paper are classical definitions of confidentiality in a multithreaded program. However, they can be overly restrictive, because they require the program behaviour to be completely deterministic. An alternative approach is to define a probabilistic confidentiality property that restricts the likelihood of a certain trace occurring. The literature contains several examples of probabilistic secure information flow properties, *e.g.*, [22,19,17]. We are currently extending our approach to such probabilistic properties, using probabilistic temporal logics and a probabilistic model checker, such as PRISM [13].

Acknowledgements. We thank Ngo Minh Tri and the anonymous reviewers for their useful feedback on earlier versions of this paper.

References

1. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Computer Security Foundation Workshop (CSFW 2017). IEEE Press, Los Alamitos (2004)
2. Blondeel, H.-C.: Security by logic: characterizing non-interference in temporal logic. Master's thesis, KTH Sweden (2007), `ftp://ftp-sop.inria.fr/everest/Marieke.Huisman/blondeel.pdf`
3. Boudol, G., Castellani, I.: Noninterference for concurrent programs and thread systems. Theor. Comput. Sci. 281(1-2), 109–130 (2002)
4. Chen, T., Ploeger, S.C.W., van de Pol, J.C., Willemse, T.A.C.: Equivalence checking for infinite systems using parameterized boolean equation systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 120–135. Springer, Heidelberg (2007)
5. Dam, M., Gurov, D.: mu-calculus with explicit points and approximations. Journal of Logic and Computation 12, 43–57 (2002)
6. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
7. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
8. Goguen, J., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
9. Groote, J.F., Orzan, S.: Parameterised anonymity. In: Degano, P., Guttman, J.D., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 177–191. Springer, Heidelberg (2009)
10. Huisman, M., Ngo, M.T.: A new definition of confidentiality for multi-threaded programs (2010) (manuscript)
11. Huisman, M., Worah, P., Sunesen, K.: A temporal logic characterisation of observational determinism. In: Computer Security Foundations Workshop (2006)
12. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
13. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Performance Evaluation Review 36(4), 40–45 (2009)
14. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980)
15. Moller, F., Stevens, P.: Edinburgh Concurrency Workbench user manual (version 7.1), `http://homepages.inf.ed.ac.uk/perdita/cwb/`
16. Roscoe, A.: CSP and determinism in security modelling. In: Symposium on Security and Privacy, pp. 114–127. IEEE Computer Society Press, Los Alamitos (1995)
17. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Computer Security Foundations Workshop, pp. 200–215. IEEE Press, Los Alamitos (2000)
18. Smith, G., Volpano, D.: Secure Information Flow in a Multi-threaded Imperative Language. In: Principles of Programming Languages, pp. 355–364 (1998)
19. Smith, G., Volpano, D.: Confinement properties for multi-threaded programs. Electronic Notes in Theoretical Computer Science 20 (1999)

20. Terauchi, T.: A type system for observational determinism. In: Computer Security Foundation, CSF 2008 (2008)
21. Terauchi, T., Aiken, A.: Secure Information Flow as a Safety Problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
22. Volpano, D., Smith, G.: Probabilistic noninterference in a concurrent language. Journal of Computer Security 7, 231–253 (1999)
23. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: 16th IEEE Computer Security Foundations Workshop (2003)