Verification of Confidentiality of Multi-threaded Programs

Ngo Minh Tri
University of Twente
Formal Methods and Tools Group
tringominh@gmail.com

1 Introduction

1.1 General description of the project

The goal of SlaLoM (Security by Logic for Multi-threaded Applications) project[1] is the development of a *verification framework* for the protection of data. Typical security properties relevant to the protection of data are *confidentiality*, *integrity* and *availability*. Confidentiality means that no private information can be derived from public data. Program's integrity is defined as the independence of the value of trusted data on untrusted sources. Availability means that an output of a program will be produced eventually.

The key idea on which this project is based is the notion of *self-composition*. Self-composition means that we compose an application (program) with itself, i.e. we execute a program and its copy in parallel, in such a way that the original two programs still can be distinguished. We rephrase the security requirements as temporal logic properties over a single execution of this self-composed program. This allows the use of standard program verification techniques, which have the advantage that the verification is both automatic and precise.

1.2 Confidentiality of multi-threaded programs

In the first part of this project, we investigate confidentiality. Different definitions exist to capture confidentiality such as *observational determinism* (a generalization of classical noninterference) and *probabilistic noninterference*.

Classical noninterference [2] expresses that a program is considered secure whenever varying the initial values of confidential (high) variables cannot change its publicly observable output behavior[2]. For example, suppose $h \in H$ and $l \in L$ are high and low variables, respectively. The program: 'if $h = 1$ then $l := 1$ else $l := 0$' is not secure because the value of the low variable depends on the value of the high variable. This is an example of an indirect information flow from the initial value of h to the final value of l.

The definition of noninterference only considers the *input* and *output* of a program. However, for concurrent and reactive systems, intermediate configurations can be observed, therefore it is necessary to also look at the *intermediate states* of the program, and to require that the private data are never revealed. Observational determinism is a generalized notion of noninterference that is defined over execution traces. Observational determinism defines that a multi-threaded program is secure when its publicly observable traces are independent of its confidential data and independent of the scheduler policy.

The definition of observational determinism only considers non-probabilistic schedulers. A non-probabilistic scheduler chooses which thread to execute next with the same

[1] SlaLoM is funded by NWO and started in March 2010.

[2] For simplicity, we consider a simple two-point security lattice, where the data is divided into private (high level) H and public (low level) L data

probability. When a scheduler's behavior is *probabilistic*, some threads will be executed more often than the other ones. This opens up the possibility of a probabilistic attack as in the following example.

$$h := h \bmod 2; \left(l := h \mid_{\frac{1}{2}} (l := 0 \mid_{\frac{1}{2}} l := 1) \right);$$

Here, $C_1 \mid_p C_2$ means that the probability of the next transition corresponding to a transition of C_1 is p. The value of p is determined by the scheduler. Again, h is a private variable, while l is a public (observable) variable. After executing $h := h \bmod 2$, the value of h will be either 0 or 1. For example, the initial value of h is 3, then after executing the command $h := h \bmod 2$, $h = 1$. If the attacker executes this program often enough, such as 100 times, he will get 100 values of l in which approximate 75 values are 1. Therefore, the final value of l in this program will reveal information about h with a probability of $\frac{3}{4}$.

In order to cope with this kind of attack, different theories of probabilistic noninterference are discussed [7, 5]. In particular, Sabelfeld and Sands [5] developed a probabilistic noninterference criterion based on a partial probabilistic low bisimulation which is an adaptation of Larsen and Skou's notion of probabilistic bisimulation [4].

The rest of this report is organized as follows. Section 2 introduces the formal definition of observational determinism and investigates its properties while section 3 discusses probabilistic noninterference and proposes a way to characterize partial probabilistic low bisimulation. Section 4 presents our plans for future work.

2 Observational Determinism

2.1 Definition

First, we let *Config* denote the set of configurations. A configuration $c = \langle \boldsymbol{C}, s \rangle$ consists of a program $\boldsymbol{C} \in \boldsymbol{Com}$ and a store $s \in \mathbf{St}$ where $\boldsymbol{Com}$ is the set of programs and $\mathbf{St}$ is the set of stores. A store is a finite mapping from variables to values. We define low-equivalence $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same. Given configuration $\langle \boldsymbol{C}, s \rangle$, an infinite list of configurations $T = c_0, c_1, c_2, ...$ is a program trace of $\langle \boldsymbol{C}, s \rangle$, denoted $\langle \boldsymbol{C}, s \rangle \Downarrow T$, iff $c_0 = \langle \boldsymbol{C}, s \rangle$ and $\forall i \in N. c_i \rightarrow c_{i+1}$. We use $T|_s$ to denote the projection of a program trace to the store. $T|_L$ denotes the low store trace which is the projection of $T|_s$ to all variable locations in the set of low variables L.

According to Terauchi [6], a program is observationally deterministic iff given any two initial low equivalent stores s_1 and s_2, any two low traces are equivalent upto stuttering and prefixing. Two traces T^1 and T^2 are *stuttering equivalent* if we can partition T^1 and T^2 into blocks of states, such that the states in the k^{th} block of T^1 are labelled the same as the states in the k^{th} block of T^2. Two states are labelled the same iff the values of low variables in two stores are the same. Corresponding blocks may have different lengths. T^1 and T^2 are equivalent upto *stuttering and prefixing* if there is a prefix of one trace that is stuttering equivalent to the other trace. Given two traces T^1 and T^2, we write $T^1 \approx T^2$ if T^1 and T^2 are stuttering and prefixing equivalent.

Definition 1. ***Observational Determinism****:*
A program $\boldsymbol{C}$ is observationally deterministic w.r.t. L iff for all stores s_1, s_2 such that $s_1 =_L s_2$, and for all traces T^1 and T^2,

$$\langle \boldsymbol{C}, s_1 \rangle \Downarrow T^1 \wedge \langle \boldsymbol{C}, s_2 \rangle \Downarrow T^2 \Rightarrow T^1|_L \approx T^2|_L.$$

2.2 Characterization

Now, we investigate the properties of observational determinism and characterize them by temporal logic formulas. The low store trace is denoted by a sequence of low stores which are the set of the values of the low variables. Suppose that we let symbols $\mathbf{a}, \mathbf{b}, \mathbf{c}, ...$ represent low stores in low store traces. Low stores with the *same values of low variables* are indistinguishable; therefore, they will be represented by the *same symbol*, e.g. in low store trace **cc**, the program just manipulates high variables and the values of the low variables remain *unchanged*. Given $T^1|_L = \mathbf{aabbbbbbbcddefffgh}\cdots$ and $T^2|_L = \mathbf{aaaabcdef}$, which start from two low equivalent stores, they are equivalent upto stuttering and prefixing.

From these two low store traces, we can observe the following property which should be expressed by the temporal logic formula:

a. If there is a value change in *low variables* and this change occurs *first* in trace T^m $(m = 1, 2)$, i.e. at the index i_1, then in trace T^{3-m}, at the states T_i^{3-m} with the index $i \geq i_1$, the total number of value changes counted from the first state is *strictly smaller* than the total number of value changes at the states T_i^m. This proposition holds until the *same value change* occurs in trace T^{3-m}.

We also need an extra property called *mutual fairness condition*:

b. *Mutual fairness condition*: It cannot be the case that from some point on trace T^1 (T^2), the program has a *possible next state* in which it changes the values of the low variables, while the program in trace T^2 (T^1) *never changes* the values of low variables.

We need this extra property because of a need to reject a program like this:

$$\texttt{if}\ (\texttt{h})\ \texttt{then}\ \texttt{l} := \texttt{7}\ \texttt{else}\ \texttt{while}\ (\texttt{true});$$

Suppose we execute this program with the initial store s_1 where the private variable $h = true$. We obtain a trace T^1 where the initial low store will change following the execution of the command $\texttt{l} := \texttt{7}$, i.e. $T^1|_L = \mathbf{ab}$. However, when we execute this program with another initial low equivalent store s_2 where $h = false$, we obtain another trace, T^2, where the initial low store remains unchanged because of the infinite empty while loop, $T^2|_L = \mathbf{aaaa}\cdots$. This program is *not secure* because depending on whether it finishes or goes in an infinite empty loop, the attacker knows about the sign of the initial value of h. These two low store traces are stuttering and prefixing equivalent and thus it *cannot be rejected* by (a). However, it will be *rejected* by condition (b).

Based on these two properties, we characterize observational determinism by temporal logic properties for which model checking algorithms exist. This allows the reuse of standard program verification techniques, thus resulting in a *sound* and potentially *complete* verification technique.

3 Probabilistic Noninterference

3.1 Definition

In this section, we discuss about probabilistic noninterference. Sabelfeld and Sands developed a probabilistic noninterference criterion based on a *partial probabilistic bisimulation* [5]. The aim of Sabelfeld and Sands' paper is to describe a *modification* of probabilistic bisimulation of Larsen and Skou [4] to reflect the "equivalence" of program behavior that is visible to the attackers.

Define semantics transitions from a configuration c to a set of configurations S by:

$$c \rightarrow_p S \Leftrightarrow p = \sum \{q | c \rightarrow_q d,\ d \in S\}.$$

where $c \rightarrow_p S$ denotes that the sum of probabilities of all transitions from configuration c to configurations in S is precisely p.

A *partial equivalence relation* (per) on a set A is a binary relation on A which is both symmetric and transitive.

Definition 2. ***Partial probabilistic low bisimulation*** *:*

A per R is a partial probabilistic low bisimulation on commands iff whenever $\boldsymbol{C}R\boldsymbol{D}$ then

$$\begin{aligned}
&\forall s_1 =_L s_2.\ \langle \boldsymbol{C}, s_1\rangle \rightarrow \langle \boldsymbol{C'}, s_1'\rangle \ \Rightarrow \\
&\quad \exists \boldsymbol{D'}, s_2'.\ \langle \boldsymbol{D}, s_2\rangle \rightarrow \langle \boldsymbol{D'}, s_2', \rangle \\
&\wedge\, \boldsymbol{C'}R\boldsymbol{D'} \wedge s_1' =_L s_2', \\
&\wedge \sum \{|p|\ \langle \boldsymbol{C}, s_1\rangle \rightarrow_p \langle \boldsymbol{S}, s\rangle, \boldsymbol{S} \in [\boldsymbol{C'}]_R, s =_L s_1'\} = \\
&\quad \sum \{|p|\ \langle \boldsymbol{D}, s_2\rangle \rightarrow_p \langle \boldsymbol{S}, s\rangle, \boldsymbol{S} \in [\boldsymbol{D'}]_R, s =_L s_2'\}.
\end{aligned}$$

where $[\boldsymbol{E}]_R$ represents the R-equivalence class which contains $\boldsymbol{E}$.

We write $\boldsymbol{C}\, R\, \boldsymbol{D}$ ($\boldsymbol{C}$ and $\boldsymbol{D}$ are probabilistically low-bisimilar) iff there exists a partial probabilistic low bisimulation that relates program $\boldsymbol{C}$ to program $\boldsymbol{D}$.

Definition 3. ***The security specification ([5]):*** *$\boldsymbol{C}$ is secure iff $\boldsymbol{C}\, R\, \boldsymbol{C}$.*

The intuition behind this definition is that a program is secure iff for any two low equivalent stores, two configurations containing the program and each of the stores, execute in such a way that their resulting behavior is *indistinguishable* from the attacker's observation of *low stores* and the *probability* with which they occur.

3.2 Characterization of partial probabilistic low bisimulation

Larsen and Skou state that two states are *probabilistically bisimilar* only if they satisfy exactly the same *Probabilistic Modal Logic (PML) formulas* [4]. Therefore, we think that we can use the set of PML formulas to characterize the partial probabilistic low bisimulation. One technical problem is that Sabelfeld and Sands' definition of partial probabilistic low bisimulation is defined over *unlabelled probabilistic transition systems* (unlabelled PTS), while the definition of probabilistic bisimulation of Larsen and Skou [4] is defined over *labelled probabilistic transition systems* (labelled PTS) in which each transition is labelled by an action. Another problem is that whether low bisimulation can be characterized by PML formulas with/without some *adjustments*.

Therefore, first we need to show that there is a relation between unlabelled PTS and labelled PTS. We argue that in case we just consider whether the values of the low variables are changed or not, then the set of actions in labelled PTS can be restricted to only two actions: an observable action indicates a change in low values and a hidden action indicates no change in low variables. Next, we define a store memorizing transition relation as follows:

Definition 4. ***Store memorizing transition relation****: Let $\rightarrow \subseteq \mathbf{St} \times \mathbf{St}$ be a store transition relation. The store memorizing transition relation $\overset{m}{\rightarrow} \subseteq (\mathbf{St} \times \mathbf{St}) \times (\mathbf{St} \times \mathbf{St})$ is defined as*

$$(s_1, t_1) \overset{m}{\rightarrow} (s_2, t_2) \ \Leftrightarrow\ s_1 \rightarrow s_2 \wedge t_2 = s_1.$$

where t is the *additive* store which is used to memorize the previous store. Thus, (s_1, t_1) makes a transition to (s_2, t_2) if s_1 makes a transition to s_2 in the original system, and t_2 remembers the old store s_1.

Based on the *store memorizing transition relation*, we define *unlabelled/labelled store memorizing probabilistic transition systems* (unlabelled/labelled SMPTS) which

are variants of unlabelled/labelled PTS. Two actions in labelled SMPTS can be characterized by atomic propositions in unlabelled SMPTS because at each configuration, we also have the previous values of the low variables. Thus, the model of a program in unlabelled SMPTS is equivalent to its model in labelled SMPTS. After that, we rephrase the partial probabilistic bisimulation on commands over an unlabelled SMPTS into an *equivalent one* over a labelled SMPTS. We argue that PML formulas with the set of restricted actions can be used to characterize partial probabilistic low bisimulation.

4 Future plans

We plan to use a model checker to verify whether a multi-threaded application, i.e. a Java program, satisfies the security specifications or not. We believe that PRISM is a suitable tool. The reason is that PML is a subset of PRISM's property specification language which incorporates Continuous Stochastic Logic (CSL) [1], and Probabilistic real time Computation Tree Logic (PCTL) [3] temporal logics We also plan to consider how to scale with the large applications and other security properties such as integrity, availability and anonymity.

References

1. J. Desharnais and P. Panangaden. Continuous stochastic logic characterizes bisimulation of continuous-time markov processes. In *JLAP special issue on Probabilistic Techniques for the Design and Analysis of Systems*, volume 56 (1-2), pages 99–115, 2003.
2. J. Goguen and J. Meseguer. Security policies and security model. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press.
3. H. Hansson and B. Jonsson. A logic for reasoning about time and realizability. In *Formal Aspects of Computing*, volume 6(5), pages 512–535, 1994.
4. K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Information and Computation*, volume 94(1), pages 1–28, 1992.
5. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.
6. T. Terauchi. A type system for observational determinism. In *Computer Science Foundation (CSF 2008)*, 2008.
7. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Journal of Computer Security*, volume 7(2-3), pages 231–253, November 1999.