A Generic Approach to the Security of Multi-Threaded Programs

Heiko Mantel
German Research Center
for Artificial Intelligence
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
E-mail: mantel@dfki.de

Andrei Sabelfeld
Department of Computer Science
Chalmers University of Technology
and University of Göteborg
41296 Göteborg, Sweden
E-mail: andrei@cs.chalmers.se

Abstract

The security of computation at the level of a specific programming language and the security of complex systems at a more abstract level are two major areas of current security research. With the objective to integrate the two, this article proposes a translation of a timing-sensitive security property for simple multi-threaded programs into a more general security framework. Interestingly, our notion of security for programs is bisimulation-based while the security framework is trace-based. Nevertheless, we show that the translation is sound *and* complete *in the sense that the trace-based specification which results from the translation of a multi-threaded program is secure if and only if the original program is secure. The translation is presented as a two-step process where the first step is independent from the concrete programming language.*

1. Introduction

1.1. Motivation

An important step in the specification of secure information flow in a complex distributed system where local parts are written in a particular programming language is to combine two types of security. Namely, the first type is the security of communication between local computations and the second type is the security of the local computations themselves. The former is often defined as security of an event-based system (as in the underlying model of [18]) whereas the latter relies on the security specification of the programming language (as in the underlying model of [27] for a simple imperative multi-threaded language). Embracing the two kinds of security into a single security framework is the motivation of this paper.

1.2. Background

There is a large body of research on information flow control aiming at specifying, verifying, and analyzing security. In the traditional abstract view, security is often defined for an abstract *trace-based* model of computation. In particular, a system can be represented as a set of its traces and, thus, security is a property that can be true or false for a given set of traces. In a distributed setting, these traces can be viewed as sequences of events like, e.g., communication of local processes in a distributed network. Many different approaches to this type of general information flow control have been proposed (e.g., [13, 30, 12, 22, 16, 14, 33, 25, 26]), which increased the need to unify and to compare. This has led to uniform frameworks and detailed comparisons [23, 10, 34, 18].

Another line of research that is becoming increasingly popular is information flow control in a setting of a concrete programming language. The efforts in this area are focused on determining whether a given program written in a particular programming language has secure information flow. More concrete assumptions are usually made about local computations. For example, one might assume that the program runs on a partition of data on high (private) and low (public) security data (although a more general lattice of security levels can be considered). The program is not trusted (possibly received over the Internet). The program's low output is publicly available (e.g., sent over the Internet) as well as, perhaps, timing information about the program's execution (e.g., times when the program makes Internet accesses are observable).

Originating from early work of Denning [8, 9] and Cohen [5, 6], secure information flow in programming languages received its recent reincarnation in work of Volpano et al. [32] with the main contribution being soundness proofs for a Denning-style security analysis. Many other

In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 11–13, 2001.

researchers have investigated the problem of secure information flow including Joshi and Leino's equational specification [17], a single calculus for security, binding-time analysis, program slicing and call-tracking (DCC) by Abadi et al. [1], Heintze and Riecke's Secure Lambda Calculus (Slam) [15], Volpano and Smith's investigations on security of concurrent programs [29, 31], and Sabelfeld and Sands's security formalization based on partial equivalence relations [28] and a scheduler-independent probability-sensitive security specification for multi-threaded programs [27].

The security formalization in the studies mentioned founds on the extensional approach to security, namely *non-interference* [13]. The idea behind non-interference is that a system is considered secure if high inputs do not interfere with low-observable behavior of the system (low outputs, timing, etc.).

It has often been claimed that extensional programming-language-based security can be viewed as a form of non-interference (e.g., in [32]), especially since the revival of the interest in language-based security. Nevertheless, for the language-based extensional security models that have been proposed since the mid-nineties a rigorous connection to non-interference-like properties has not so far been established to the best of our knowledge. This paper is a step in this direction.

Our choice for the abstract event-based framework is Mantel's assembly kit [18]. Adapting the assembly kit allows picking the appropriate security property from the assembly kit rather than inventing a new one. This also allows for combining the security of programs with the security of other components in a (potentially distributed system) using the assembly kit as an interface. This means integrating programming-language-based security at a higher level of abstraction, opening the opportunity for plugging the security of sub-systems written in a particular programming language to the global security of the system defined in a general event-based framework.

Finally, the assembly kit enjoys a number of useful extensions including local verification conditions [19], intransitive security policies [20], and refinement operators [21], which potentially enables us to use these verification techniques, to apply intransitive security policies, and to do stepwise development in the setting of secure information flow in programs (although these issues are outside the scope of the present article).

The focus of this paper is on a simple multi-threaded language (MWL) and a timing-sensitive security specification (*strong security* [27]) that implies robust security independently of a particular scheduler. We translate MWL programs into state-event systems, pick an appropriate definition of security from the assembly kit, and establish a precise correspondence between the security of MWL programs and their translations. Namely, that the translation is *sound* in the sense that the translation of any secure MWL program is secure as a state-event system; and *complete* in the sense that if the translation of an MWL program is secure as a state-event system then the original program is secure.

1.3. Overview

After recalling some preliminaries in Section 2, we introduce the concept of thread pools in Section 3. In Section 4, we specialize this generic model according to the syntax and semantics of the MWL programming language. That this specialization indeed reflects the semantics of MWL, is ensured by a collection of theorems in Section 5. The key contribution of our translation is that it preserves the specification of secure information flow. Section 6 shows that a thread pool is considered to be secure in the MWL programming language if and only if the corresponding state-event system is also considered to be secure in the assembly kit. We conclude by a discussion in Section 7.

2. Preliminaries

2.1. System Specifications

The behavior of systems can often be adequately specified by the set of its possible execution sequences. We follow this trace-based approach throughout this article (with the exception of parts where we use a concrete programming language). A *trace* is a sequence of events that models a possible execution sequence of the system. An *event* is an atomic action like, e.g., the sending or receiving of a message on some channel. We distinguish between input and output events. The underlying intuition is that input events are controlled by the environment of a system while output events are controlled by the system. The distinction between input and output events is somewhat fuzzy. When a system is capable to prevent the occurrence of input events, then this can be interpreted as a signal to the environment. To avoid this kind of communication, *input totality* is often assumed, i.e., that a system cannot prevent the occurrence of input events. Since input totality is quite restrictive, we refrain from making this assumption in this article. In complex systems, communication between components is done by synchronization on the occurrence of shared events (usually output events of the one component that are input events of others).

For specifying systems, we do not define the set of traces directly but rather use states as an auxiliary concept. This allows us to define the possible traces inductively by a transition relation. The system model, we use for specification, are state-event systems. This system model allows for the specification of non-deterministic systems where the non-determinism is reflected by the choice between different events that are enabled. For simplicity, any non-determinism in the effects of events is ruled out.

Definition 1 *Let S be a set of states, E be a set of events, and $T \subseteq S \times E \times S$ be a transition relation. A* state-event system *SES is a tuple (S, S_I, E, I, O, T) where $S_I \subseteq S$ are the initial states and $I, O \subseteq E$ respectively are the input and output events. Throughout this paper we assume that S_I is a singleton set and that for a given state s and event e there is at most one state s' with $(s, e, s') \in T$.*

Let $s_1, s_2, s' \in S$, $e \in E$, and $\gamma \in E^*$. Instead of $(s_1, e, s_2) \in T$ we sometimes use the notation $s_1 \xrightarrow{e}_T s_2$. For multi-event transitions, we use the notation $s_1 \overset{\gamma}{\Longrightarrow}_T s'$. If T is obvious from the context then we omit the index and write $s_1 \xrightarrow{e} s_2$ or $s_1 \overset{\gamma}{\Longrightarrow} s'$. The relation $\overset{\gamma}{\Longrightarrow}_T$ is formally defined as follows:

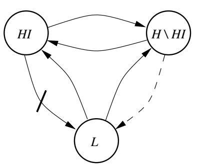
$$
\begin{array}{ll}
s_1 \overset{\langle\rangle}{\Longrightarrow}_T s' & \text{, if } s_1 = s' \\
s_1 \overset{e.\gamma}{\Longrightarrow}_T s' & \text{, if } \exists s_2 \in S. s_1 \xrightarrow{e}_T s_2 \wedge s_2 \overset{\gamma}{\Longrightarrow}_T s'
\end{array}
$$

A sequence $\tau \in E^*$ of events is a *trace* of a state-event system *SES* $= (S, \{s_0\}, E, I, O, T)$ if it is accepted in the initial state, i.e., $\exists s' \in S. s_0 \overset{\tau}{\Longrightarrow}_T s'$. The set of all traces for *SES* is denoted by Tr_{SES}. We omit the index and simply write *Tr* if the state-event system is obvious from the context. The tuple (E, I, O, Tr_{SES}) is referred to as the *event system* corresponding to *SES*. A state $s \in S$ is *reachable*, denoted by *reachable*(s), if there exists a trace $\tau \in E^*$ such that $s_0 \overset{\tau}{\Longrightarrow} s$. The *projection* $\alpha|_{E'}$ of a sequence $\alpha \in E^*$ to the events in $E' \subseteq E$ results from α by deleting all events not in E'.

2.2. Security Properties

Security requirements can be expressed as restrictions on the information flow within a system. To express confidentiality or integrity by such restrictions is the key idea of information flow control. A *security property* $SecProp_{TP}$ consists of three elements: a flow policy *FP*, a domain assignment *dom*, and a security predicate *SP*.

A *flow policy* specifies restrictions on the information flow within a system. For this purpose, firstly, a set of security domains is chosen. Typical domains are, e.g., groups of users, collections of files, or memory sections. Secondly, relations $\not\rightsquigarrow, \rightsquigarrow_V, \rightsquigarrow_N \subseteq \mathcal{D} \times \mathcal{D}$ are defined. The *non-interference relation* $\not\rightsquigarrow$ specifies where information flow between domains is forbidden. E.g., $D_1 \not\rightsquigarrow D_2$ expresses that there must be *no information flow* from D_1 to D_2. The *interference relation* $\rightsquigarrow_V$ specifies that certain domains are *visible* for others. $D_1 \rightsquigarrow_V D_2$ expresses that D_1 is visible for D_2. Finally, the relation $\rightsquigarrow_N$, specifies between which



	$(V,$	$N,$	$C\,)$
$\mathcal{V}_{HI}$	$(E,$	$\emptyset,$	$\emptyset\,)$
$\mathcal{V}_{L}$	$(L,$	$H \backslash HI,$	$HI)$
$\mathcal{V}_{H \backslash HI}$	$(E,$	$\emptyset,$	$\emptyset\,)$

Figure 1. The flow policy FP_{TP} and the views of all domains

domains information flow is *not* restricted. $D_1 \rightsquigarrow_N D_2$ expresses that D_1 is not visible for D_2 but that information about D_1 may be deducible for D_2.

Definition 2 *A* flow policy *FP is a tuple $(\mathcal{D}, \rightsquigarrow_V, \rightsquigarrow_N, \not\rightsquigarrow)$ where $\rightsquigarrow_V, \rightsquigarrow_N, \not\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$ form a disjoint partition of $\mathcal{D} \times \mathcal{D}$ and $\rightsquigarrow_V$ is reflexive. FP is called* transitive *if $\rightsquigarrow_V$ is transitive and, otherwise,* intransitive*.*

A *domain assignment* links a flow policy to a system specification by associating domains to events. We often denote the set of all events with a given domain D also by D, the name of the security domain, and use that name in lower case, possibly with indices or primes, e.g., $d, d_1, \ldots$, to denote events with that domain.

Definition 3 *A* domain assignment *dom* $: E \to \mathcal{D}$ *is a function that assigns domains to events.*

We depict flow policies as graphs where each node corresponds to a security domain. The relations $\rightsquigarrow_V$, $\rightsquigarrow_N$, and $\not\rightsquigarrow$ are respectively depicted as solid, dashed, and crossed arrows. For the sake of readability, the reflexive sub-relation of $\rightsquigarrow_V$ is usually omitted. This graphical representation is shown on the left hand side of Figure 1 for the flow policy FP_{TP}, which consists of three domains *HI* (high-level input events), L (low-level events), and $H \backslash HI$ (high-level internal and output events). According to FP_{TP}, occurrences of low-level events are visible for both high-level domains. High-level inputs must not be deducible for the low-level (*HI* $\not\rightsquigarrow L$). Other high-level events may be deducible ($H \backslash HI \rightsquigarrow_N L$), if this does not reveal information about high-level inputs.

Traditionally, FP_{TP} would be defined as a policy with two domains L, H and the relations $H \not\rightsquigarrow L$, $L \rightsquigarrow H$. This leaves it implicit that occurrences of events in $H \backslash HI$ may be deducible for L. Our distinction between $\not\rightsquigarrow$ and $\rightsquigarrow_N$ allows us to make such assumptions explicit in the flow policy.

A *security predicate* specifies under which conditions a system specification satisfies a flow policy for some domain assignment. It can also be understood as a *definition of what information flow means*. *SP* must be satisfied for the view of each domain, whereas the *view* $\mathcal{V}_D = (V, N, C)$

for a domain $D \in \mathcal{D}$ in *FP* is defined by $V = \bigcup\{D' \in \mathcal{D} \mid D' \rightsquigarrow_V D\}$, $N = \bigcup\{D' \in \mathcal{D} \mid D' \rightsquigarrow_N D\}$, and $C = \bigcup\{D' \in \mathcal{D} \mid D' \not\rightsquigarrow D\}$. Basically, V contains all events that are *visible* for D, C contains all events that are *confidential* for D, and N contains all events that are *neither* visible nor confidential. The views for all domains of FP_{TP} are depicted on the right-hand side of Figure 1. Among these, the view of domain L is the only interesting one because it gives rise to a non-trivial proof obligation. The precise proof obligation, of course, depends on the security predicate.

An assembly kit that allows for the uniform and modular representation of security predicates, has been previously proposed by one of the authors [18]. It simplifies the comparison among the existing security predicates and a goal-directed construction of new ones. In the assembly kit, security predicates are composed by conjunction from one or more basic security predicates (abbreviated by *BSP*).

For the purposes of the current paper, a simple security predicate suffices which consists only of a single *BSP*, *backwards strict insertion of admissible confidential events* (abbreviated by *BSIA*). $BSIA_{\mathcal{V}}$ requires that the occurrence of an event from C does *not remove* possible low-level observations. Considering the system after a trace β has occurred, any observation $\overline{\alpha} \in V^*$ that is possible must also be possible after an arbitrary confidential event $c \in C$ has occurred. If the observation $\overline{\alpha}$ results from $\alpha \in (V \cup N)^*$, i.e., $\alpha|_V = \overline{\alpha}$, then some $\alpha' \in (V \cup N)^*$ must be possible after c has occurred where α' may differ from α only in events from N. The premise $\beta.c \in Tr$ ensures that the event c is admissible after β. For a given view $\mathcal{V} = (V, N, C)$, $BSIA_{\mathcal{V}}$ is formally defined as follows:

$$
\begin{aligned}
&BSIA_{V,N,C}(Tr) \equiv \\
&\forall \alpha, \beta \in E^*. \forall c \in C. ((\beta.\alpha \in Tr \wedge \alpha|_C = \langle\rangle \wedge \beta.c \in Tr) \\
&\implies \exists \alpha' \in E^*. (\alpha'|_V = \alpha|_V \wedge \alpha'|_C = \langle\rangle \wedge \beta.c.\alpha' \in Tr))
\end{aligned}
$$

The security guarantee provided by *BSIA* is: if an adversary observes $\overline{\alpha}$ starting in some state then he or she cannot deduce that a confidential event c *has not* occurred. Clearly, it could also be important to prevent an adversary from deducing that a confidential event *has* occurred. For *BSPs* which provide this type of guarantee (and others), we refer to [18, 19, 20].

3. Generic Thread Pools

For distributed programming, the use of multi-threaded programming languages has become extremely popular [4]. The use of concurrent threads that operate in the same address space appears to be the adequate approach for applications that are, e.g., based on the client-server paradigm. For example, this allows one to program a file server that creates, for every incoming request, a new thread that handles this request and terminates afterwards. Compared to parallelism at the level of processes, an important advantage is that context switching is far less expensive for threads.

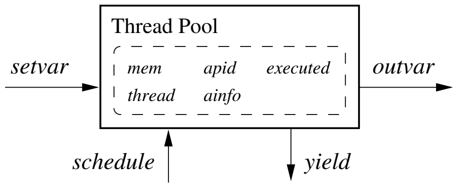


Figure 2. Generic thread pool with interface events and state objects

To model the behavior of multi-threaded processes in state-event systems is technically somewhat difficult.[1] The main difficulty is that threads communicate with each other asynchronously via shared memory, while state-event systems are based on a synchronous, message-passing-like communication paradigm (cf. Section 2.1). However, to specify processes with these formalisms is very natural because inter-process communication is usually synchronous.

In this section, we demonstrate how the behavior of multi-threaded processes can be modeled using state-event systems. The proposed specification is highly generic because it is not only parametric in the particular program but also in the programming language. How to instantiate this specification for the concrete programming language MWL will be demonstrated in Section 4.

3.1. Trace-Based Formal Specification

In our specification, a multi-threaded process is modeled as a collection of threads that shares a global memory. We refer to such a collection as a *thread pool*. As depicted in Figure 2, a thread pool has five state objects (*mem*, *thread*, *apid*, *ainfo*, *executed*) and can communicate with the environment by four kinds of interface events (*setvar*-, *outvar*-, *schedule*- and *yield*-events).

The *shared memory* of a thread pool is modeled by the function $mem : \mathtt{VAR} \to \mathtt{VAL}$ that assigns values (from VAL) to variables (from VAR). The shared memory can be updated at the interface of a thread pool by *setvar*-events. If an event $setvar(var, val)$ occurs then variable *var* is assigned value *val*. *outvar*-events output the value of variables to the environment. An event $outvar(var, val)$ is only enabled if *var* currently has value *val*. For simplicity, we assume that *outvar*-events have no other preconditions and that *setvar*-events are always enabled.

The *local state of threads* is modeled by the function $thread : \mathtt{PID} \to (\mathtt{THREAD} \cup \{\bot, \top, \langle\rangle\})$. $thread(pid)$ re-

[1] Similar problems occur when using process algebras like CSP or CCS.

turns a local state (from THREAD) for the identifier *pid* $\in$ PID. The results $\perp$, $\top$, and $\langle\rangle$ do not denote a proper local state but have a special meaning. If a thread with identifier *pid* has never existed then $thread(pid) = \perp$ holds. After a thread has spawned child processes, the identifier of the parent thread is modified and *thread* returns $\top$ for the old identifier. If $thread(pid) = \langle\rangle$ then a thread with identifier *pid* has existed but has already terminated.

The remaining state objects are used for controlling the execution of threads. The value of $apid \in \mathtt{PID} \cup \{\perp\}$ denotes the identifier of the thread that is currently active in the thread pool. $apid = \perp$ indicates that no thread is active. For simplicity, we assume that there is at most one active thread at any point of time. *ainfo* is a buffer in which information is collected that shall be send to the scheduler. Note that the scheduler is external to a thread pool. The flag *executed* : BOOL is used for managing context switching. Thread execution proceeds as follows.

- If no thread is active (indicated by $apid = \perp$) then *schedule*-events are enabled. After an occurrence of $schedule(pid)$, *apid* is set to *pid*, and the thread with local state $thread(pid)$ becomes active. $schedule(pid)$ is only enabled if the thread is alive ($thread(pid) \notin \{\perp, \top, \langle\rangle\}$).
- If there is an active thread (indicated by $apid \neq \perp \wedge executed = ff$) then this thread can run. Thread execution is formally modeled by the occurrence of events that are internal to the thread pool. Since these internal events depend closely on the particular instantiation of a generic thread pool, especially on the programming language, they are intentionally not modeled at the generic level. During execution, a thread can affect the state objects *mem* and *thread*. Additionally, information for the scheduler is stored in *ainfo*. Eventually, the active thread stops executing (indicated by $executed = tt$).
- After the active thread has stopped ($executed = tt$), the scheduler can be informed about this by a *yield*-event. $yield(info)$ is only enabled if *info* corresponds to the actual scheduler information ($info = ainfo$). A *yield*-event resets the *executed*-flag, *apid*, and *ainfo*.

For the initial state, we assume that all variables are initialized with the same value *initval*. Moreover, we assume that there is exactly one thread. This thread has *initpid* as identifier and *initthread* as local state. In the initial state, the *executed*-flag, *apid*, and *ainfo* are reset.

Generic thread pools are formalized as state-event systems in the following definition.

Definition 4 *Let* VAR, VAL, PID, THREAD, *and* INFO *be types. Let* S, s_0, E_{pool}, I_{pool}, O_{pool}, *and* T_{pool} *be defined as depicted in Figure 3. Let* $initval \in \mathtt{VAL}$, $initpid \in \mathtt{PID}$, $initthread \in \mathtt{THREAD}$, E_{local} *be a set of events that is disjoint from* E_{pool}, *and* $T_{local} \subseteq S \times E_{local} \times S$ *be a transition relation.*

The generic thread pool *which is parametric in* VAR, VAL, PID, THREAD, INFO, *initval, initpid, initthread,* E_{local}, *and* T_{local}, *is defined by the following state-event system:*

$$\begin{aligned} &GenPool(\mathtt{VAR}, \mathtt{VAL}, \mathtt{PID}, \mathtt{THREAD}, \mathtt{INFO}, \\ &\qquad initval, initthread, initpid, E_{local}, T_{local}) \\ &= (S, \{s_0\}, E_{pool} \cup E_{local}, I_{pool}, O_{pool}, T_{pool} \cup T_{local}) \end{aligned}$$

3.2. Security of Thread Pools

The problem of information flow control in multi-threaded programming languages is to prevent information flow from high to low variables. For this purpose, a security level (*low* or *high*) is assigned to each variable by a function $dom_{var} : var \to \{low, high\}$. This differs from the event-based approach, in which information flow control prevents that occurrences or non-occurrences of confidential events affect the possibility of observable behaviors. Although both approaches share the same intuitive motivation, i.e., that there should be no information flow from high to low, this technical difference complicates an integration of the two approaches. However, an integration is very desirable because it allows for a uniform investigation of information flow at the level of processes as well as at the level of threads.

The key observation, which will allow us to integrate the two approaches, is that high-level data can only be introduced into a thread pool by occurrences of *setvar*-events that change the value of high-level variables. All other events can change the state of the thread pool but cannot increase the confidentiality of data. Thus, we can express the security requirement by demanding that the occurrences of these *setvar*-events must not influence the possibility of low-level observations.

The flow policy FP_{TP} (cf. left-hand side of Figure 1) expresses the necessary restrictions on information flow. We assume that a (malicious) low-level user has complete knowledge about the definition of thread pools (as usual), can observe the occurrence of *schedule*- and *yield*-events, and can observe the occurrences of *outvar*- and *setvar*-events that involve only low-level variables. Consequently, all these events are assigned domain L (cf. Figure 4). *setvar*-events that involve high-level variables are assigned domain *HI* because the occurrence of these events must not be deducible by a low-level user. Occurrences of all other events must not be observable by the low-level user. They may be deducible. However, such deductions must not reveal any information about occurrences of events in *HI*.

$$S = \{\mathit{mem}, \mathit{thread}, \mathit{apid}, \mathit{ainfo}, \mathit{executed} \mid \mathit{mem} : \mathtt{VAR} \to \mathtt{VAL}, \mathit{thread} : \mathtt{PID} \to \mathtt{THREAD} \cup \{\bot, \top, \langle\rangle\}, \mathit{apid} : \mathtt{PID} \cup \{\bot\}, \mathit{ainfo} : \mathtt{INFO} \cup \{\bot\}, \mathit{executed} : \mathtt{BOOL}\}$$

$$s_0 = \{\forall \mathit{var} \in \mathtt{VAR}.\, \mathit{mem}(\mathit{var}) = \mathit{initval}, \mathit{thread}(\mathit{initpid}) = \mathit{initthread}, \forall \mathit{pid} \in \mathtt{PID}.\, \mathit{pid} \neq \mathit{initpid} \implies \mathit{thread}(\mathit{pid}) = \bot, \mathit{apid} = \bot, \mathit{ainfo} = \bot, \mathit{executed} = \mathit{ff}\}$$

$$E_{pool} = I_{pool} \cup O_{pool}$$

$$I_{pool} = \{\mathit{setvar}(\mathit{var}, \mathit{val}) \mid \mathit{var} \in \mathtt{VAR} \wedge \mathit{val} \in \mathtt{VAL}\} \cup \{\mathit{schedule}(\mathit{pid}) \mid \mathit{pid} \in \mathtt{PID}\}$$

$$O_{pool} = \{\mathit{outvar}(\mathit{var}, \mathit{val}) \mid \mathit{var} \in \mathtt{VAR} \wedge \mathit{val} \in \mathtt{VAL}\} \cup \{\mathit{yield}(\mathit{info}) \mid \mathit{info} \in \mathtt{INFO}\}$$

T_{pool} is defined by

- $\mathit{setvar}(\mathit{var}, \mathit{val})$ affects $\mathit{mem}(\mathit{var})$
  Pre : *true*
  Post: $\mathit{mem'}(\mathit{var}) = \mathit{val}$
- $\mathit{schedule}(\mathit{pid})$ affects *apid*
  Pre : $\mathit{apid} = \bot \wedge \mathit{thread}(\mathit{pid}) \notin \{\bot, \top, \langle\rangle\}$
  Post: $\mathit{apid'} = \mathit{pid}$
- $\mathit{yield}(\mathit{info})$ affects *executed*, *apid*, *ainfo*
  Pre : $\mathit{executed} = \mathit{tt} \wedge \mathit{ainfo} = \mathit{info}$
  Post: $\mathit{executed'} = \mathit{ff} \wedge \mathit{apid'} = \bot \wedge \mathit{ainfo'} = \bot$
- $\mathit{outvar}(\mathit{var}, \mathit{val})$ affects —
  Pre : $\mathit{mem}(\mathit{var}) = \mathit{val}$
  Post: *true*

Figure 3. Definition of fixed components of a generic thread pool

e	$dom_{TP}(e)$	
$\mathit{schedule}(\mathit{pid})$ $\mathit{yield}(\mathit{info})$	L	
$\mathit{setvar}(\mathit{var}, \mathit{val})$ $\mathit{outvar}(\mathit{var}, \mathit{val})$	L	, if $dom_{var}(\mathit{var}) = \mathit{low}$
$\mathit{setvar}(\mathit{var}, \mathit{val})$	HI	, if $dom_{var}(\mathit{var}) = \mathit{high}$
$\mathit{outvar}(\mathit{var}, \mathit{val})$	$H \backslash HI$	, if $dom_{var}(\mathit{var}) = \mathit{high}$
e	$H \backslash HI$	, if $e \in E_{local}$

Figure 4. Domain assignment dom_{TP}

Definition 5 *The security property $SecProp_{TP}$ for thread pools is $(FP_{TP}, dom_{TP}, BSIA)$.*

According to FP_{TP}, proof obligations arise only for the view of domain L. Thus, a thread pool *satisfies* $SecProp_{TP}$ if $BSIA_{\mathcal{V}}$ holds for the view $\mathcal{V}_L = (L, H\backslash HI, HI)$. Note that *BSIA* (cf. Section 2.2) is indeed an appropriate definition of information flow for this application. The argument is as follows: if changing the value of high-level variables does not eliminate the possibility of low-level behaviors, then there is no information flow from high to low because high-level variables could have any value at any given point of time. Technically, a similar effect could be achieved by demanding a *BSP* that deletes confidential events, like, e.g., *BSD* (cf. [19]). However, this possibility is not important for the purposes of this paper.

In general, choosing a definition of information flow closely depends on the particular application under consideration and there appears not to be a single "right" definition (as, e.g., also observed in [26]). The assembly kit offers a (still growing) collection of very primitive definitions of information flow (*BSPs*) and allows one to assemble these to more complex definitions (security predicates). This fine-grained view has proved to be very helpful for determining $SecProp_{TP}$.

4. MWL Thread Pools

In this section, we revisit the simple multi-threaded while-language (abbreviated by MWL) along with the timing-sensitive definition of security for MWL from [27]. Further, we demonstrate how our generic specification of thread pools from Section 3 can be instantiated for MWL.

4.1. The Multi-Threaded While-Language MWL

MWL is a shared-variable multi-threaded while-language with dynamic thread creation. The syntax of MWL commands is given by the grammar in Figure 5. As usual, boolean expressions B range over $\mathtt{BOOL}$ and arithmetic expressions Exp range over $\mathtt{EXP}$. Let $C, D, E, \ldots$ range over commands (MWL threads) $\mathtt{CMD}$, and let $\vec{C}$ denote a vector of commands of the form $\langle C_1 \ldots C_n \rangle$. Vectors $\vec{C}, \vec{D}, \vec{E}, \ldots$ range over $\vec{\mathtt{CMD}} = \cup_{n \in \mathbb{N}} \mathtt{CMD}^n$, the set of multi-threaded programs.

MWL programs execute under a shared memory on a single processor (or in a single process) such that at most one thread can be active at any given point of time. A *configuration* $\langle\!| C, mem |\!\rangle$ (or $\langle\!| \vec{C}, mem |\!\rangle$) is a pair, consisting of a command $C \in \mathtt{CMD}$ (or a vector of commands $\vec{C} \in \vec{\mathtt{CMD}}$) and a memory $mem \in \mathtt{VAR} \to \mathtt{VAL}$. *mem* is a finite mapping from variables to values, as in Section 3. The set of variables is partitioned into high and low security classes. For simplicity (but without loss of generality), we will assume that there is only one variable for each security class, h and l, respectively. We will often write the memory simply as a pair (val_h, val_l) with the values val_h for h and val_l for l. Further, we define *low-equivalence* on memories by "$mem_1 =_L mem_2$ if and only if the values of l for mem_1 and mem_2 are the same". The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 6. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\twoheadrightarrow$-transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, mem |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem' |\!\rangle$, which means termination with the final memory mem', or $\langle\!| C, mem |\!\rangle \twoheadrightarrow \langle\!| C'\vec{D}, mem' |\!\rangle$. Here, one step of computation starting with command C in a memory *mem* gives a new main thread C', a vector $\vec{D}$ of spawned threads (possibly empty), and a new memory mem'. The command $\mathsf{fork}(C\vec{D})$, where $\vec{D}$ is required to be non-empty, dynamically creates a new vector $\vec{D}$ of threads that, afterwards, run in parallel with the main thread C. This has the effect of adding the vector $\vec{D}$ to the configuration. The rule [Pick] in Figure 7 defines the concurrent semantics of MWL. Whenever the scheduler picks a thread C_i for execution, then a $\rightarrow$-transition takes place updating the command pool and the shared memory according to a (small) computation step of C_i. Let $\rightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$.

We can extract a simple model of the timing behavior of multi-threaded programs from the small-step semantics. This is done by the assumption that each $\twoheadrightarrow$-transition takes a single unit of time to execute. This approach gives only a rough approximation of real timing behavior, but simple extensions are possible in order to make it sensitive to the timing behavior of particular commands (cf. [2]).

4.2. Definition of Security for MWL

Now, we define the security of MWL programs and motivate the choice of this definition. The central idea of *extensional* security, as opposed to *intensional* security, is that confidentiality should not be specified by a special-purpose security formalism, but, rather, should be defined in terms of a standard semantics as a dependency property (more precisely, absence of dependence). If direct, indirect, and timing flows are considered, then, intuitively, a program has the extensional *noninterference* property, if varying the high input will not change the possible low-level observations, i.e., low inputs/outputs and timing. This differs from intensional security which relies on particular security primitives that are only motivated by intuition rather than a mathematical justification. Many investigations have successfully followed the extensional view including [6, 32, 15, 29, 1, 31, 17, 27, 2, 28] for justification

$$\texttt{CMD} ::= \textsf{skip} \mid \texttt{VAR} := \texttt{EXP} \mid \texttt{CMD}_1 ; \texttt{CMD}_2 \mid \textsf{if } \texttt{BOOL} \textsf{ then } \texttt{CMD}_1 \textsf{ else } \texttt{CMD}_2$$
$$\mid \textsf{while } \texttt{BOOL} \textsf{ do } \texttt{CMD} \mid \textsf{fork}(\texttt{CMD}\ \vec{\texttt{CMD}})$$

Figure 5. Command syntax

[Skip] $$\langle\!| \textsf{skip}, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| \langle\rangle, \mathit{mem} |\!\rangle$$

[Assign] $$\frac{Exp \downarrow^{\mathit{mem}} = n}{\langle\!| Id := Exp, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| \langle\rangle, [Id = n]\mathit{mem} |\!\rangle}$$

[Seq_1] $$\frac{\langle\!| C_1, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| \langle\rangle, \mathit{mem}' |\!\rangle}{\langle\!| C_1 ; C_2, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| C_2, \mathit{mem}' |\!\rangle}$$

[Seq_2] $$\frac{\langle\!| C_1, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| C_1' \vec{D}, \mathit{mem}' |\!\rangle}{\langle\!| C_1 ; C_2, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| (C_1' ; C_2)\vec{D}, \mathit{mem}' |\!\rangle}$$

[$\mathrm{If}_{\mathit{tt}}$] $$\frac{B \downarrow^{\mathit{mem}} = \mathit{tt}}{\langle\!| \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| C_1, \mathit{mem} |\!\rangle}$$

[$\mathrm{If}_{\mathit{ff}}$] $$\frac{B \downarrow^{\mathit{mem}} = \mathit{ff}}{\langle\!| \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| C_2, \mathit{mem} |\!\rangle}$$

[$\mathrm{While}_{\mathit{tt}}$] $$\frac{B \downarrow^{\mathit{mem}} = \mathit{tt}}{\langle\!| \textsf{while } B \textsf{ do } C, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| C ; \textsf{while } B \textsf{ do } C, \mathit{mem} |\!\rangle}$$

[$\mathrm{While}_{\mathit{ff}}$] $$\frac{B \downarrow^{\mathit{mem}} = \mathit{ff}}{\langle\!| \textsf{while } B \textsf{ do } C, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| \langle\rangle, \mathit{mem} |\!\rangle}$$

[Fork] $$\langle\!| \textsf{fork}(C\vec{D}), \mathit{mem} |\!\rangle \rightharpoonup \langle\!| C\vec{D}, \mathit{mem} |\!\rangle$$

Figure 6. Small-step deterministic semantics of commands

[Pick] $$\frac{\langle\!| C_i, \mathit{mem} |\!\rangle \rightharpoonup \langle\!| \vec{C}, \mathit{mem}' |\!\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, \mathit{mem} |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} \vec{C} C_{i+1} \dots C_n \rangle, \mathit{mem}' |\!\rangle}$$

Figure 7. Concurrent semantics of programs

of security analysis and verification techniques for different languages. We follow the extensional approach and focus on extensional security for MWL.

A previous investigation [27] gives an account on choosing an adequate definition of extensional security for multi-threaded programs. Which definition is appropriate depends on, for instance, whether a particular scheduler is assumed, or a robust scheduler-independent security is wanted. The central idea of the bisimulation-based approach is to define a *low-bisimulation* on commands such that the indistinguishability of the behaviors of two programs C and D for the attacker is formalized by $C \sim_L D$, where $\sim_L$ is a *low-bisimulation*. Such an approach is flexible in the choice of an appropriate low-bisimulation (different low-bisimulations are available for different degrees of security). For a given low-bisimulation $\sim_L$, the definition of security is simply: "C is secure iff $C \sim_L C$". For the purpose of this paper we adapt the *strong low-bisimulation* [27].

Definition 6 *Define* strong low-bisimulation $\cong_L$ *to be the union of all symmetric relations R on MWL command pools (programs) of equal size for which whenever $\langle C_1 \dots C_n \rangle$ R $\langle D_1 \dots D_n \rangle$ then*

$$\begin{aligned} \forall mem_1, mem_2, i. \langle\!| C_i, mem_1 |\!\rangle &\twoheadrightarrow \langle\!| \vec{C}', mem_1' |\!\rangle \\ &\wedge\ mem_1 =_L mem_2 \Longrightarrow \\ \exists \vec{D}', mem_2'. \langle\!| D_i, mem_2 |\!\rangle &\twoheadrightarrow \langle\!| \vec{D}', mem_2' |\!\rangle \\ &\wedge\ mem_1' =_L mem_2' \wedge \vec{C}' \ R\ \vec{D}' \end{aligned}$$

Our definition of security for MWL programs is based on strong low-bisimulations. The choice of this particular bisimulation results in a definition of security that is timing-sensitive and scheduler-independent. *Strong bisimulation captures timing flows.* If two commands might have a different timing behavior depending on high data (which would result in information flow from high to low) then they are not low-bisimilar. *Strong bisimulation is scheduler-independent.* Thus, our notion of security is robust with the respect to any choice of a particular scheduler (including probabilistic schedulers as shown in [27]). Although these features impose restrictions on what can be considered low-bisimilar, the choice of *strong low-bisimulation is adequate* (not too restrictive), e.g., for the type-based analysis that is proposed in [27]. This analysis is sound with respect to the security definition, i.e., if a program passes the analysis, then it must be secure. For more details on the power of this type of security definition to capture insecure programs and examples of secure programming with common algorithms, such as sorting and searching, we refer to [27, 3].

Definition 7 *An MWL program $\vec{C}$ is* secure *if and only if $\vec{C} \cong_L \vec{C}$.*

In order to illustrate Definitions 6 and 7 we give some examples of secure and insecure information flow which may occur in MWL programs.

$l := h$ This is an example of a *direct* flow. To see that this program is insecure according to Definition 7, choose $mem_1 = (0,0)$ and $mem_2 = (1,0)$. Since $\langle\!| l := h, (0,0) |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, (0,0) |\!\rangle$ and $\langle\!| l := h, (1,0) |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, (1,1) |\!\rangle$ holds, the resulting memories are not low-equivalent $(0,0) \neq_L (1,1)$. Thus, there cannot be a relation with the properties necessary for strong low-bisimilarity.

if $h = 1$ **then** $l := 1$ **else** $l := 0$ This exemplifies an *indirect* flow through branching on a high condition. If the computation starts with low-equivalent memories $(0,0)$ and $(1,0)$, then, after one step of the computation (the test of the condition), the memories are still low-equivalent. However, after another computation step they become different depending on the initial value of h. There cannot be a relation with the properties necessary for strong low-bisimilarity.

if $h = 1$ **then** (**while** $l < MaxInt$ **do** $l := l + 1$) **else skip** From the timing behavior of the program the attacker may deduce secret information. This is an instance of a *timing* leak. Clearly, the timing behavior of the branches is different. This is captured by Definition 7. Indeed, in case the then-branch of the if is chosen, there will be no transition in the other branch to match the transitions of the while-loop.

if $h = 1$ **then** (**while** *true* **do skip**) **else skip** is a variation of the timing leak called a *termination* leak.

All examples above are insecure according to our definition. Here is an instance of a secure program:

if $h = 1$ **then** $h := h + 1$ **else skip** Indeed, the timing behavior is independent of the value of h, as well as the low variable l. A suitable symmetric relation that makes this program low-bisimilar to itself is, e.g., the relation $\{(\text{if } h = 1 \text{ then } h := h + 1 \text{ else skip}, \text{if } h = 1 \text{ then } h := h + 1 \text{ else skip}), (h := h + 1, \text{skip}), (\text{skip}, h := h + 1), (h := h + 1, h := h + 1), (\text{skip}, \text{skip}), (\langle\rangle, \langle\rangle)\}$.

4.3. Instantiating Generic Thread Pools

We now instantiate our generic model for thread pools from Section 3 in order to model the behavior of the multi-threaded programs of MWL. Recall, that, according to Definition 4, the following parameters must be actualized:

- types: `VAR`, `VAL`, `PID`, `THREAD`, `INFO`

- initial values: *initval*, *initthread*, *initpid*,
- internal events: E_{local}; and their behavior: T_{local}.

Consistently with the simplification of Section 4.1, the set VAR of variables consists of only two variables h and l (having in mind that h is a high-level and l a low-level variable). We do not further specify the set VAL of values. However, we assume that there is a set EXP of expressions. $Exp \downarrow^{mem}= val$ denotes that $Exp \in \texttt{EXP}$ evaluates to *val* where the memory *mem* in the index is only important if Exp contains variables. Moreover, assume a set BOOL of boolean expressions. $B \downarrow^{mem}= tt$ and $B \downarrow^{mem}= ff$ denote, respectively, that $B \in \texttt{BOOL}$ evaluates to *true* or *false*. PID is specialized to the set of sequences of natural numbers ($\texttt{PID} = \mathbb{N}^*$). The set THREAD is specialized to CMD, i.e., the local state of a thread is simply an MWL command. INFO is specialized to VAL $\times$ INT where VAL is the value of the priority variable (which is adapted to be l for simplicity) and the INT part says whether the process has been killed (value -1), continues running (value 0) or has spawned $n > 0$ new processes (value n).

We do not further specify *initval*, the initial value of all variables. The identifier of the (unique) initial thread is zero, i.e., $initpid = 0$. MWL thread pools shall be parametric in the initial thread (parameter *initthread*).

We now introduce two auxiliary functions $first : \texttt{CMD} \to \texttt{CMD}$ and $rest : \texttt{CMD} \to \texttt{CMD} \cup \{\langle\rangle\}$. The purpose of *first* and *rest* is to decompose sequential compositions. In the definition given in Figure 8 we assume that C_1 does *not* have the form $D_1; D_2$, i.e., C_1 is not a sequential composition on the top level. The set E^{MWL}_{local} of internal events of an MWL thread pool is defined in Figure 9. Note that for each of these events there is a corresponding rule of the small-step semantics (cf. Figure 6). E.g., the *assign*-events correspond to rule Assign and the events ite^{tt} and ite^{ff} respectively correspond to If_{tt} and If_{ff}. With the exception of the rules Seq_1 and Seq_2, there are corresponding events in E^{MWL}_{local} for each rule in Figure 6. The reason for this correspondence is that, on the one hand side, events model atomic actions and, on the other hand, rules of a small-step semantics model atomic transitions between states (or configurations – in the case of MWL). The atomic actions that can occur during the execution of an MWL thread pool include, taking up time (caused by skip), assignments to variables, branching in the control flow depending on boolean tests (if then else or while do), or spawning of threads (fork). Note that, we do not consider the decomposition of sequentially composed commands as a separate action. Thus, there are no corresponding events.

The behavior of internal events is defined by the transition relation T^{MWL}_{local} (cf. Figure 10). Clearly, T^{MWL}_{local} should reflect the semantics of MWL. The pre- and postcondition of each event shall capture the corresponding rule of the small-step semantics. E.g., the precondition of $assign(var, val)$ requires that there is an active thread ($apid \neq \bot$) that has not already executed a command ($executed = ff$), the current command must be an assignment ($first(thread(apid)) = var := Exp$), and the expression Exp must evaluate to *val* under the current memory ($Exp \downarrow^{mem}= val$). Note that, when new threads are spawned, then the generation of identifiers is managed in such a way that no pid is used for two different processes (cf. postcondition of *fork*). That T^{MWL}_{local} indeed reflects the semantics of MWL will be proved in Section 5.

The instantiation of generic thread pools for MWL is summarized in the following definition.

Definition 8 *Let initthread* $\in$ CMD*. The* MWL thread pool *for initthread results from the following instantiation of generic thread pools:*

$$MWLPool(initthread) = GenPool(\{l, h\}, \texttt{VAL}, \mathbb{N}^*, \texttt{CMD}, \texttt{VAL} \times \texttt{INT}, initval, initthread, 0, E^{\text{MWL}}_{local}, T^{\text{MWL}}_{local})$$

5. Semantic Relation between MWL Programs and MWL Thread Pools

The objective of our specification of MWL thread pools was to provide an adequate model of MWL programs and their behavior. Firstly, any behavior of an MWL thread pool should comply with the MWL semantics. Secondly, any behavior that complies with the MWL semantics should be possible for an MWL thread pool. That our specification, indeed, is adequate is ensured by the results presented in the current section.

Recall that the system models that, respectively, underly MWL programs and MWL thread pools are somewhat different. The model underlying MWL programs is based on *trees of states* (to be precise, configurations). It is possible to enrich these trees with events but from the perspective of the underlying paradigm these events would be mere decorations. Since the model of computation is state-based, the natural communication paradigm is via shared memory. The system model underlying MWL thread pools is based on *sequences of events*. It is possible to enrich these sequences with states but from the perspective of the underlying model these states would be mere decorations. Since the system model is event-based, the natural communication paradigm is via message passing. These differences between the system models on which MWL programs and MWL thread pools are based, somewhat complicate the proofs of the following theorems.

5.1. Adequateness of MWL Thread Pools

In Theorem 1 we will show that every trace of an MWL thread pool models a behavior that complies with the semantics of MWL. We define the function cseq, which

$$\langle \mathit{first}(C), \mathit{rest}(C)\rangle = \begin{cases} \langle C, \langle\rangle\rangle & \text{, if } C \in \{\mathsf{skip}, \mathit{var} := Exp, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \\ & \qquad \mathsf{while}\ B\ \mathsf{do}\ C, \mathsf{fork}(C\vec{D})\} \\ \langle C_1, C_2\rangle & \text{, if } C = (C_1; C_2) \end{cases}$$

Figure 8. Definition of *first* and *rest*

$$\begin{aligned} E^{\text{MWL}}_{\mathit{local}} \quad = \quad & \{\mathit{skip}\} \cup \{\mathit{assign}(\mathit{var}, \mathit{val}) \mid \mathit{var} \in \mathtt{VAR} \wedge \mathit{val} \in \mathtt{VAL}\} \\ & \cup \{\mathit{ite}^{\mathit{tt}}(B, C_1, C_2), \mathit{ite}^{\mathit{ff}}(B, C_1, C_2) \mid B \in \mathtt{BOOL} \wedge C_1, C_2 \in \mathtt{CMD}\} \\ & \cup \{\mathit{while}^{\mathit{tt}}(B, C_1), \mathit{while}^{\mathit{ff}}(B, C_1) \mid B \in \mathtt{BOOL} \wedge C_1 \in \mathtt{CMD}\} \\ & \cup \{\mathit{fork}(C, \vec{D}) \mid C \in \mathtt{CMD} \wedge \vec{D} \in \vec{\mathtt{CMD}}\} \end{aligned}$$

Figure 9. Definition of local events $E^{\text{MWL}}_{\mathit{local}}$ of an MWL thread pool

$T^{\text{MWL}}_{\mathit{local}}$ is defined by

- skip affects $\mathit{thread}(\mathit{apid})$, $\mathit{executed}$, ainfo
  $\mathit{Pre}:\ \mathit{executed} = \mathit{ff} \wedge \mathit{apid} \neq \bot \wedge \mathit{first}(\mathit{thread}(\mathit{apid})) = \mathsf{skip}$
  $\mathit{Post}:\ \mathit{thread}'(\mathit{apid}) = \mathit{rest}(\mathit{thread}(\mathit{apid})) \wedge \mathit{executed}' = \mathit{tt}$
  $\wedge \mathit{ainfo}' = (\mathit{mem}(l), \mathit{terminates}(\mathit{thread}(\mathit{apid})))$
- $\mathit{assign}(\mathit{var}, \mathit{val})$ affects $\mathit{mem}(\mathit{var})$, $\mathit{thread}(\mathit{apid})$, $\mathit{executed}$, ainfo
  $\mathit{Pre}:\ \mathit{executed} = \mathit{ff} \wedge \mathit{apid} \neq \bot \wedge Exp \downarrow^{\mathit{mem}} = \mathit{val} \wedge \mathit{first}(\mathit{thread}(\mathit{apid})) = \mathit{var} := Exp$
  $\mathit{Post}:\ \mathit{mem}'(\mathit{var}) = \mathit{val} \wedge \mathit{thread}'(\mathit{apid}) = \mathit{rest}(\mathit{thread}(\mathit{apid})) \wedge \mathit{executed}' = \mathit{tt}$
  $\wedge \mathit{ainfo}' = (\mathit{mem}(l), \mathit{terminates}(\mathit{thread}(\mathit{apid})))$
- $\mathit{ite}^{\mathit{tt}}(B, C_1, C_2)$ affects $\mathit{thread}(\mathit{apid})$, $\mathit{executed}$, ainfo
  $\mathit{Pre}:\ \mathit{executed} = \mathit{ff} \wedge \mathit{apid} \neq \bot \wedge B \downarrow^{\mathit{mem}} = \mathit{tt} \wedge \mathit{first}(\mathit{thread}(\mathit{apid})) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$
  $\mathit{Post}:\ \mathit{thread}'(\mathit{apid}) = C_1; \mathit{rest}(\mathit{thread}(\mathit{apid})) \wedge \mathit{executed}' = \mathit{tt} \wedge \mathit{ainfo}' = (\mathit{mem}(l), 0)$
- $\mathit{ite}^{\mathit{ff}}(B, C_1, C_2)$ affects $\mathit{thread}(\mathit{apid})$, $\mathit{executed}$, ainfo
  $\mathit{Pre}:\ \mathit{executed} = \mathit{ff} \wedge \mathit{apid} \neq \bot \wedge B \downarrow^{\mathit{mem}} = \mathit{ff} \wedge \mathit{first}(\mathit{thread}(\mathit{apid})) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$
  $\mathit{Post}:\ \mathit{thread}'(\mathit{apid}) = C_2; \mathit{rest}(\mathit{thread}(\mathit{apid})) \wedge \mathit{executed}' = \mathit{tt} \wedge \mathit{ainfo}' = (\mathit{mem}(l), 0)$
- $\mathit{while}^{\mathit{tt}}(B, C_1)$ affects $\mathit{thread}(\mathit{apid})$, $\mathit{executed}$, ainfo
  $\mathit{Pre}:\ \mathit{executed} = \mathit{ff} \wedge \mathit{apid} \neq \bot \wedge B \downarrow^{\mathit{mem}} = \mathit{tt} \wedge \mathit{first}(\mathit{thread}(\mathit{apid})) = \mathsf{while}\ B\ \mathsf{do}\ C_1$
  $\mathit{Post}:\ \mathit{thread}'(\mathit{apid}) = C_1; \mathsf{while}\ B\ \mathsf{do}\ C_1; \mathit{rest}(\mathit{thread}(\mathit{apid})) \wedge \mathit{executed}' = \mathit{tt} \wedge \mathit{ainfo}' = (\mathit{mem}(l), 0)$
- $\mathit{while}^{\mathit{ff}}(B, C_1)$ affects $\mathit{thread}(\mathit{apid})$, $\mathit{executed}$, ainfo
  $\mathit{Pre}:\ \mathit{executed} = \mathit{ff} \wedge \mathit{apid} \neq \bot \wedge B \downarrow^{\mathit{mem}} = \mathit{ff} \wedge \mathit{first}(\mathit{thread}(\mathit{apid})) = \mathsf{while}\ B\ \mathsf{do}\ C_1$
  $\mathit{Post}:\ \mathit{thread}'(\mathit{apid}) = \mathit{rest}(\mathit{thread}(\mathit{apid})) \wedge \mathit{executed}' = \mathit{tt}$
  $\wedge \mathit{ainfo}' = (\mathit{mem}(l), \mathit{terminates}(\mathit{thread}(\mathit{apid})))$
- $\mathit{fork}(C, D_1 \ldots D_n)$ affects $\mathit{thread}(\mathit{apid})$, $\mathit{thread}(\mathit{apid}.0) \ldots \mathit{thread}(\mathit{apid}.n)$, $\mathit{executed}$, ainfo
  $\mathit{Pre}:\ \mathit{executed} = \mathit{ff} \wedge \mathit{apid} \neq \bot \wedge \mathit{first}(\mathit{thread}(\mathit{apid})) = \mathsf{fork}(CD_1 \ldots D_n)$
  $\mathit{Post}:\ \mathit{thread}'(\mathit{apid}) = \top \wedge \mathit{thread}'(\mathit{apid}.0) = C; \mathit{rest}(\mathit{thread}(\mathit{apid}))$
  $\wedge \forall i \in \{1, \ldots, n\} : \mathit{thread}'(\mathit{apid}.i) = D_i \wedge \mathit{executed}' = \mathit{tt} \wedge \mathit{ainfo}' = (\mathit{mem}(l), n)$

where $\mathit{terminates}(\mathit{thread}(\mathit{apid}))$ equals -1 if $\mathit{rest}(\mathit{thread}(\mathit{apid})) = \langle\rangle$ and 0 otherwise.

Figure 10. Definition of transition relation $T^{\text{MWL}}_{\mathit{local}}$ of an MWL thread pool

$$\mathtt{cseq_{aux}}(\mathit{pid}, \mathit{thread}) = \begin{cases} \langle\rangle & \text{, if } \mathit{thread}(\mathit{pid}) \in \{\bot, \langle\rangle\} \\ \mathit{thread}(\mathit{pid}) & \text{, if } \mathit{thread}(\mathit{pid}) \in \mathtt{CMD} \\ \mathtt{cseq_{aux}}(\mathit{pid}.0, \mathit{thread}) \ldots & \\ \ldots \mathtt{cseq_{aux}}(\mathit{pid}.n, \mathit{thread}) & \text{, if } \mathit{thread}(\mathit{pid}) = \top \\ & \quad n \in \mathbb{N} \text{ is chosen maximal such that } \mathit{thread}(\mathit{pid}.n) \neq \bot \end{cases}$$

Figure 11. Definition of $\mathtt{cseq_{aux}}$

translates a function $thread : \texttt{PID} \rightarrow (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\})$ into a corresponding vector of MWL commands. Note that this definition exploits that identifiers are chosen incrementally by *fork*-events and that $thread(pid) = \langle\rangle$ holds after termination of a thread with identifier *pid*.

Definition 9 $\texttt{cseq} : (\texttt{PID} \rightarrow (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\}) \rightarrow \vec{\texttt{CMD}}$ *returns a corresponding vector of MWL commands for each function thread* $: \texttt{PID} \rightarrow (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\})$. $\texttt{cseq}$ *is defined by* $\texttt{cseq}(thread) = \texttt{cseq}_{\texttt{aux}}(0, thread)$ *where* $\texttt{cseq}_{\texttt{aux}} : \texttt{PID} \rightarrow (\texttt{PID} \rightarrow (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\}) \rightarrow \vec{\texttt{CMD}}$ *is defined in Figure 11.*

We now present two lemmas that are helpful for proving Theorems 1, 3, and 4. The proofs of lemmas and theorems that are omitted in the present paper can be accessed via the authors' homepages. Throughout this section, we assume that $SES = (S, S_I, E, I, O, T)$ models an MWL thread pool, i.e., that $SES = MWLPool(initthread)$ holds for some command $initthread \in \texttt{CMD}$.

Lemma 1 *If $reachable(s)$ holds in a state s of SES then*

- $executed_s = ff \wedge apid_s \neq \bot \wedge thread_s(apid_s) \notin \{\bot, \top, \langle\rangle\}$,
- $executed_s = tt \wedge apid_s \neq \bot$, *or*
- $executed_s = ff \wedge apid_s = \bot$ *holds.*

Lemma 2 *Let s, s' be states of SES with $executed_s = ff$, $apid_s \neq \bot$, $thread_s(apid_s) \notin \{\bot, \top, \langle\rangle\}$, and $e \in E^{MWL}_{local}$ be an event such that $s \xrightarrow{e} s'$ holds.*

- *If* $e \neq fork(C, D_1 \dots D_n)$ *then* $\langle\!|thread_s(apid_s), mem_s|\!\rangle \twoheadrightarrow \langle\!|thread_{s'}(apid_{s'}), mem_{s'}|\!\rangle$.
- *If* $e = fork(C, D_1 \dots D_n)$ *then* $\langle\!|thread_s(apid_s), mem_s|\!\rangle \twoheadrightarrow \langle\!|thread_{s'}(apid_s.0) \dots thread_{s'}(apid_s.n), mem_{s'}|\!\rangle$.

Theorem 1 *Let $s, s' \in S$ be states of SES, $\gamma \in E^*$ be a sequence of events, and $\vec{D}_s, \vec{D}_{s'}$ be vector of MWL commands with $\vec{D}_s = \texttt{cseq}(thread_s)$ and $\vec{D}_{s'} = \texttt{cseq}(thread_{s'})$. If $reachable(s)$, $s \stackrel{\gamma}{\Longrightarrow} s'$, and γ contains no setvar-events then $\langle\!|\vec{D}_s, mem_s|\!\rangle \rightarrow^* \langle\!|\vec{D}_{s'}, mem_{s'}|\!\rangle$.*

In Theorem 2, we will show that for every behavior that complies with the semantics of MWL, there is a corresponding trace of an MWL thread pool which models that behavior. We now present a lemma that is helpful for proving that theorem and also Theorems 3 and 4.

Lemma 3 *Let $C' \in \texttt{CMD}$, $D_1 \dots D_n \in \vec{\texttt{CMD}}$, and $mem' : \texttt{VAR} \rightarrow \texttt{VAL}$. Moreover, let s be a state of SES with $executed_s = ff$, $apid_s \neq \bot$, and $thread_s(apid_s) \notin \{\bot, \top, \langle\rangle\}$.*

1. *If $\langle\!|thread_s(apid_s), mem_s|\!\rangle \twoheadrightarrow \langle\!|C', mem'|\!\rangle$ then there exists an event $e \in E^{MWL}_{local}$ and a state s' of SES with $s \xrightarrow{e} s'$, $mem_{s'} = mem'$, $thread_{s'}(apid_s) = C'$, $apid_{s'} = apid_s$, and $executed_{s'} = tt$. Moreover, for all $pid \in \texttt{PID}$ with $pid \neq apid_s$ holds $thread_{s'}(pid) = thread_s(pid)$.*
2. *If $\langle\!|thread_s(apid_s), mem_s|\!\rangle \twoheadrightarrow \langle\!|C' D_1 \dots D_n, mem'|\!\rangle$ (with $n \geq 1$) then there exists an event $e \in E^{MWL}_{local}$ and a state s' of SES with $s \xrightarrow{e} s'$, $mem_{s'} = mem'$, $thread_{s'}(apid_s) = \top$, $apid_{s'} = apid_s$, and $executed_{s'} = tt$. Moreover, $thread_{s'}(apid_s.0) = C'$, $thread_{s'}(apid_s.i) = D_i$ holds for all $i \in \{1, \dots, n\}$, and $thread_{s'}(pid) = thread_s(pid)$ holds for all $pid \in \texttt{PID}$ with $pid \notin \{apid_s, apid_s.0, \dots, apid_s.n\}$.*

Theorem 2 *Let s be a state of SES such that $apid_s = \bot$, $executed_s = ff$, and $reachable(s)$. Let $\vec{D}_s = \texttt{cseq}(thread_s)$, $\vec{D'} \in \vec{\texttt{CMD}}$, and $mem' : var \rightarrow val$. If $\langle\!|\vec{D}_s, mem_s|\!\rangle \rightarrow^* \langle\!|\vec{D'}, mem'|\!\rangle$ then there exists a sequence $\gamma : E^*$ that contains no setvar-events and a state $s' \in S$ such that $s \stackrel{\gamma}{\Longrightarrow} s'$, $mem_{s'} = mem'$, and $\vec{D'} = \texttt{cseq}(thread_{s'})$.*

Theorem 1 and 2 ensure that MWL thread pools are an adequate specification of MWL programs and their behavior. All behaviors of an MWL thread pool comply with the semantics of MWL and all behaviors that comply with the semantics of MWL are possible for an MWL thread pool.

6. Soundness and Completeness Results

The aim of this section is to establish the soundness and completeness results. First, we recall the definition of the translation of a program in MWL into a state-event system and then proceed by proving soundness (if C is secure as an MWL program then its translation is secure as a state-event system) and completeness (if C's translation is secure as a state-event system then C is secure).

According to Definition 8 from Section 4, the translation $MWLPool(C)$ of an MWL program C is the thread pool with $initthread = C$. The following two sub-sections present the soundness and completeness results respectively.

6.1. Soundness

Before we present the soundness theorem we state a security invariant lemma. Intuitively, the lemma says that if computation starts with a secure program then all the threads in the thread pool are secure at all times. Define an auxiliary boolean function $live(s, pid) = thread_s(pid) \notin \{\bot, \top, \langle\rangle\}$ that takes the value *tt* whenever thread at *pid* in the state s is alive (exists and has not terminated). Note that $live(s, pid) = tt$ is the precondition for scheduling the thread at *pid* in s.

Lemma 4 *Assume an MWL program C is secure and $\beta \in Tr$ is a trace for MWLPool(C) such that $s_0 \stackrel{\beta}{\Longrightarrow} s$. Then $\forall pid.\, live(s, pid) \Longrightarrow thread_s(pid) \cong_L thread_s(pid)$.*

Theorem 3 (Soundness) *If an MWL program C is secure then the MWL thread pool MWLPool(C) satisfies the security property $SecProp_{TP}$.*

6.2. Completeness

Let us first recall some facts from standard bisimulation theory before we turn to proving completeness. Restating Definition 6, two thread pools $\vec{C} = \langle C_1 \dots C_n \rangle$ and $\vec{D} = \langle D_1 \dots D_n \rangle$ are strongly low-bisimilar $\vec{C} \cong_L \vec{D}$ iff $\exists R.\, R \subseteq F(R)$ where function F from pers to pers (partial equivalence relations over $\vec{\mathrm{CMD}}$) is given by: $\vec{C}\ F(R)\ \vec{D}$ iff

$$\begin{aligned} \forall mem_1, mem_2, i. \langle\!| C_i, mem_1 |\!\rangle \twoheadrightarrow \langle\!| \vec{C'}, mem'_1 |\!\rangle \\ \wedge\ mem_1 =_L mem_2 \Longrightarrow \\ \exists \vec{D'}, mem'_2. \langle\!| D_i, mem_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{D'}, mem'_2 |\!\rangle \\ \wedge\ mem'_1 =_L mem'_2 \wedge \vec{C'}\ R\ \vec{D'} \end{aligned}$$

Let us state two lemmas that give an alternative representation for the strong low-bisimulation. The proof of the lemmas is a standard argument, by appeal to the Knaster-Tarski fixed-point theorem (see, e.g., [7]).

Lemma 5 *Function F is ω-cocontinuous, i.e., for a non-increasing ω-chain of pers $R_0 \supseteq \dots \supseteq R_i \supseteq \dots$, F preserves colimits:*

$$F(\cap_{i<\omega} R_i) = \cap_{i<\omega} F(R_i).$$

Lemma 6 (Fixed point) *The relation $\cong_L$ is the greatest fixed point of F in the lattice of pers. It can be alternatively represented by $\cong_L = \cap_{i<\omega} \cong^i_L$ where $\cong^{i+1}_L = F(\cong^i_L)$ and $\cong^0_L$ is the total relation $\vec{\mathrm{CMD}} \times \vec{\mathrm{CMD}}$.*

We are now ready to present the completeness result.

Theorem 4 (Completeness) *An MWL program C is secure whenever MWLPool(C) satisfies the security predicate $SecProp_{TP}$.*

Proof. [Sketch] Due to the space restrictions we present a detailed sketch of the proof technique rather than giving the complete proof. Assuming that *MWLPool*(C) satisfies $BSIA_{\{L\},\{H \setminus HI\},\{HI\}}$ we need to show that C is secure, i.e., $C \cong_L C$ (by Definition 7). Let us prove this statement by contraposition. In other words, assuming $C \not\cong_L C \wedge SecProp_{TP}(MWLPool(C))$ we aim to arrive at a contradiction.

By Lemma 6 $C \cong_L C \iff C(\cap_{i<\omega} \cong^i_L)C$. Assuming $C \not\cong_L C$ implies $\exists i.\, C \not\cong^i_L C$. Take $k = \min\{i \mid C \cong^i_L C \wedge C \not\cong^{i+1}_L C\}$. Note that $k \geq 0$ since, obviously, $C \cong^0_L C$. Assume for simplicity that no fork-command occurs in C, i.e., C never spawns new threads. Along the way, we discuss how the proof can be modified to go through without the assumption. We consider two sequences of transitions of the form given in Figure 12. Note that each element of the sequences inherits the command in the configuration from the previous element. Observe that the low parts of the memory progress in both sequences in the same way. The sequences continue as shown in Figure 13. These sequences must exist due to $\forall i \in \{0 \dots k\}.\, C \cong^i_L C$ and $C \not\cong^{k+1}_L C$. Matching the first k steps in both sequences and the low-equivalence of the memories during the first k steps are guaranteed by $\forall i \in \{0 \dots k\}.\, C \cong^i_L C$. However, at step $k+1$ we have $\forall D_{k+1}, \hat{l}'_{k+1}.\, \langle\!| D_k, (h'_k, l_k) |\!\rangle \twoheadrightarrow \langle\!| D_{k+1}, (\hat{h}'_{k+1}, \hat{l}'_{k+1}) |\!\rangle \Longrightarrow \hat{l}_{k+1} \neq \hat{l}'_{k+1}$. In case C may spawn new threads, the difference is that instead of inheriting the commands from the previous element in the sequences Seq1 and Seq2, the next command is chosen from the command in the previous configuration by selecting the thread that is the counterexample for the low-bisimulation of thread pools obtained at the previous step. Importantly, the sequences of *pid*'s chosen in both Seq1 and Seq2 are then identical. We will use this observation later.

We proceed by constructing two traces of *MWLPool*(C) that correspond to the two sequences. We will transform one trace into the other using $SecProp_{TP}$ such that the low-equivalences and step matching is preserved. This will take us to a contradiction at step $k+1$. Start off by constructing a trace of *MWLPool*(C) that corresponds to Seq1. We appeal to Lemma 3 to obtain step-by-step construction of a trace γ of the form given in Figure 14 for some $pid_0, \dots, pid_k, info_1, \dots, info_{k+1}$ where each e_i ($i = 1, \dots, k+1$) is the internal event that corresponds to the $\twoheadrightarrow$-transition in Seq1 according to Lemma 3. In case no threads are spawned $pid_i = 0$ for all $i = 0, \dots, k$. As we noted, in case C may spawn new threads the sequences of *pid*'s chosen in both Seq1 and Seq2 are identical. By a similar argument the information contained in *info* sequences must also be identical for Seq1 and Seq2 up to $info_k$.

Due to $SecProp_{TP}$ we can insert high events into right tails of γ that do not contain any high events. We get a legitimate trace after the insertion. Let us insert the $setvar(h, \hat{h}_k)$ event between $setvar(h, h_k)$ and e_{k+1} in β. Define $c = setvar(h, \hat{h}_k)$ $\alpha = e_{k+1}.yield(info_{k+1}).outvar(l, \hat{l}_{k+1})$ and $\beta = \delta.setvar(h, h_k)$ for some δ such that $\gamma = \beta.\alpha$. We have $\alpha|_{HI} = \langle\rangle$. By $SecProp_{TP}$ we have $\exists \alpha'.\, \alpha'|_L = \alpha|_L \wedge \alpha'|_{HI} = \langle\rangle \wedge \delta.setvar(h, h_k).setvar(h, \hat{h}_k).\alpha' \in Tr$. Observe that setting h to $\hat{h}_k$ means restoring the value of h

$$\text{Seq1: } \langle\!| C,(h_0,l_0)|\!\rangle \rightharpoonup \langle\!| C_1,(\hat{h}_1,\hat{l}_1)|\!\rangle \quad \langle\!| C_1,(h_1,l_1)|\!\rangle \rightharpoonup \langle\!| C_2,(\hat{h}_2,\hat{l}_2)|\!\rangle \quad \langle\!| C_2,(h_2,l_2)|\!\rangle \rightharpoonup \langle\!| C_3,(\hat{h}_3,\hat{l}_3)|\!\rangle \dots$$
$$\text{Seq2: } \langle\!| C,(h'_0,l_0)|\!\rangle \rightharpoonup \langle\!| D_1,(\hat{h}'_1,\hat{l}_1)|\!\rangle \quad \langle\!| D_1,(h'_1,l_1)|\!\rangle \rightharpoonup \langle\!| D_2,(\hat{h}'_2,\hat{l}_2)|\!\rangle \quad \langle\!| D_2,(h'_2,l_2)|\!\rangle \rightharpoonup \langle\!| D_3,(\hat{h}'_3,\hat{l}_3)|\!\rangle \dots$$

Figure 12. Sequences Seq1 and Seq2

$$\text{Seq1: } \dots \langle\!| C_{k-1},(h_{k-1},l_{k-1})|\!\rangle \rightharpoonup \langle\!| C_k,(\hat{h}_k,\hat{l}_k)|\!\rangle \quad \langle\!| C_k,(h_k,l_k)|\!\rangle \rightharpoonup \langle\!| C_{k+1},(\hat{h}_{k+1},\hat{l}_{k+1})|\!\rangle$$
$$\text{Seq2: } \dots \langle\!| D_{k-1},(h'_{k-1},l_{k-1})|\!\rangle \rightharpoonup \langle\!| D_k,(\hat{h}'_k,\hat{l}_k)|\!\rangle \quad \langle\!| D_k,(h'_k,l_k)|\!\rangle \not\rightharpoonup \langle\!| D_{k+1},(\hat{h}'_{k+1},\hat{l}_{k+1})|\!\rangle$$

Figure 13. The continuation of Seq1 and Seq2

$$\begin{aligned}\gamma = &\mathit{schedule}(\mathit{pid}_0).\mathit{setvar}(l,l_0).\mathit{setvar}(h,h_0).e_1.\mathit{yield}(\mathit{info}_1).\mathit{outvar}(l,\hat{l}_1).\\ &\mathit{schedule}(\mathit{pid}_1).\mathit{setvar}(l,l_1).\mathit{setvar}(h,h_1).e_2.\mathit{yield}(\mathit{info}_2).\mathit{outvar}(l,\hat{l}_2).\dots\\ &\dots\mathit{schedule}(\mathit{pid}_{k-1}).\mathit{setvar}(l,l_{k-1}).\mathit{setvar}(h,h_{k-1}).e_k.\mathit{yield}(\mathit{info}_k).\mathit{outvar}(l,\hat{l}_k).\\ &\mathit{schedule}(\mathit{pid}_k).\mathit{setvar}(l,l_k).\mathit{setvar}(h,h_k).e_{k+1}.\mathit{yield}(\mathit{info}_{k+1}).\mathit{outvar}(l,\hat{l}_{k+1})\end{aligned}$$

Figure 14. Sequence γ

$$\begin{aligned}\gamma' = &\mathit{schedule}(\mathit{pid}_0).\mathit{setvar}(l,l_0).e'_1.\mathit{yield}(\mathit{info}_1).\mathit{outvar}(l,\hat{l}_1).\\ &\mathit{schedule}(\mathit{pid}_1).\mathit{setvar}(l,l_1).e'_2.\mathit{yield}(\mathit{info}_2).\mathit{outvar}(l,\hat{l}_2).\dots\\ &\dots\mathit{schedule}(\mathit{pid}_{k-1}).\mathit{setvar}(l,l_{k-1}).e'_k.\mathit{yield}(\mathit{info}_k).\mathit{outvar}(l,\hat{l}_k).\\ &\mathit{schedule}(\mathit{pid}_k).\mathit{setvar}(l,l_k).e'_{k+1}.\mathit{yield}(\mathit{info}_{k+1}).\mathit{outvar}(l,\hat{l}_{k+1})\end{aligned}$$

Figure 15. Sequence γ'

$$\begin{aligned}\gamma'' = &\mathit{schedule}(\mathit{pid}_0).\mathit{setvar}(l,l_0).\mathit{setvar}(h,h'_0).e''_1.\mathit{yield}(\mathit{info}_1).\mathit{outvar}(l,\hat{l}_1).\\ &\mathit{schedule}(\mathit{pid}_1).\mathit{setvar}(l,l_1).\mathit{setvar}(h,h'_1).e''_2.\mathit{yield}(\mathit{info}_2).\mathit{outvar}(l,\hat{l}_2).\dots\\ &\dots\mathit{schedule}(\mathit{pid}_{k-1}).\mathit{setvar}(l,l_{k-1}).\mathit{setvar}(h,h'_{k-1}).e''_k.\mathit{yield}(\mathit{info}_k).\mathit{outvar}(l,\hat{l}_k).\\ &\mathit{schedule}(\mathit{pid}_k).\mathit{setvar}(l,l_k).\mathit{setvar}(h,h'_k).e''_{k+1}.\mathit{yield}(\mathit{info}_{k+1}).\mathit{outvar}(l,\hat{l}_{k+1})\end{aligned}$$

Figure 16. Sequence γ''

$$\text{Seq2: } \langle\!| C,(h'_0,l_0)|\!\rangle \rightharpoonup \langle\!| D_1,(\hat{h}'_1,\hat{l}_1)|\!\rangle \quad \langle\!| D_1,(h'_1,l_1)|\!\rangle \rightharpoonup \langle\!| D_2,(\hat{h}'_2,\hat{l}_2)|\!\rangle \quad \langle\!| D_2,(h'_2,l_2)|\!\rangle \rightharpoonup \langle\!| D_3,(\hat{h}'_3,\hat{l}_3)|\!\rangle \dots$$
$$\dots \langle\!| D_{k-1},(h'_{k-1},l_{k-1})|\!\rangle \rightharpoonup \langle\!| D_k,(\hat{h}'_k,\hat{l}_k)|\!\rangle \quad \langle\!| D_k,(h'_k,l_k)|\!\rangle \rightharpoonup \langle\!| D_{k+1},(\hat{h}_{k+1},\hat{l}_{k+1})|\!\rangle$$

Figure 17. New form of Seq2

from the result of the previous transition in Seq1. We can just as well omit both updates of h implying $\delta.\alpha' \in \mathit{Tr}$.

Carrying on with the elimination of the rightmost event $\mathit{setvar}(h, h_i)$ for $i = k, \ldots, 0$ we get a trace $\gamma' \in \mathit{Tr}$ that (after removing all occurrences of $\mathit{outvar}(h, \cdot)$-events without loss of generality) has the form depicted in Figure 15. Due to the low events embracing each local event e_i for $i = 1, \ldots, k+1$ it must be the case that there is a one-to-one correspondence between e_i and e'_i for $i = 1, \ldots, k+1$ (although they are not necessarily identical).

The next pass is the insertion of $\mathit{setvar}(h, \cdot)$ following the sequence Seq2. Let us construct new versions of c, α, β in order to apply $\mathit{SecProp}_{TP}$. Let $c = \mathit{setvar}(h, h'_0)$, $\beta = \mathit{schedule}(\mathit{pid}_0).\mathit{setvar}(l, l_0)$ and α is such that $\gamma' = \beta.\alpha$. By $\mathit{SecProp}_{TP}$ we have $\exists \alpha'.\, \alpha'|_L = \alpha|_L \wedge \alpha'|_{HI} = \langle\rangle \wedge \beta.\mathit{setvar}(h, h'_0).\alpha' \in \mathit{Tr}$. Continuing rightmost $\mathit{setvar}(h, h'_i)$ $(i = 0, \ldots, k+1)$ insertion we get a trace $\gamma'' \in \mathit{Tr}$ that (again, after removing all occurrences of $\mathit{outvar}(h, \cdot)$-events without loss of generality) has the form depicted on Figure 16. Due to the low *schedule*- and *yield*-events embracing each local event e'_i $(i = 1, \ldots, k+1)$ it must be the case that there is a one-to-one correspondence between e'_i and e''_i for $i = 1, \ldots, k+1$ (although they are not necessarily identical).

According to Lemma 2 we can now convert the trace γ'' into a sequence of $\twoheadrightarrow$-transitions. The crucial property is that these transitions are deterministic, i.e., if $\langle\!| E, \mathit{mem} |\!\rangle \twoheadrightarrow \langle\!| E', \mathit{mem}' |\!\rangle$ then $\forall E'', \mathit{mem}''.\, \langle\!| E, \mathit{mem} |\!\rangle \twoheadrightarrow \langle\!| E'', \mathit{mem}'' |\!\rangle \implies E' = E'' \wedge \mathit{mem}' = \mathit{mem}''$. In case C may spawn processes, we also need the observation we made about the same sequences of *pid*'s and *info*'s that are used in the construction of Seq1 and Seq2. This is important to restore the branching behavior of traces as it was in Seq1 and Seq2. The fact that only programs with the same branching structure can be low-bisimilar is reflected in the traces, because the branching behavior is recorded in the low *schedule*-events. Thus, by induction, we can restore the sequence Seq2, as depicted in Figure 17 for some command D_{k+1}, which contradicts our original assumption about Seq2. $\square$

7. Discussion and Future Work

Contributions. We have established a one-to-one correspondence between a time-sensitive definition of security for the multi-threaded programs of MWL (from [27]) and a security property based on traces of events that was originally developed in the context of a general security framework – the assembly kit (from [18, 19]). As a prerequisite for this, we had to model the semantics of MWL using state-event systems, which resulted in the specification of MWL thread pools. The development of this specification has been straightforward (although technically subtle). To us, it is appealing that generic thread pools, which served as an intermediate step in this process, are independent of MWL. We expect that this will allow the adaptation of other multi-threaded programming languages, e.g., Slam [15].

The main motivation of our work has been the objective to integrate the two kinds of security: the security of local computations and the security of their communications. Event-based security aims at protecting occurrences of events and programming-language-based security aims at protecting secret values. Our work is a step to aid in the systematic security analysis of complex (potentially distributed) system where some of the components are (or shall be) implemented in a specific programming language. To the best of our knowledge this article is the first attempt to establish a rigorous connection between these two notions of security. The connection suggests directions for mutual benefits where the two areas can borrow from each other (cf. Future Work).

As a side effect, we have demonstrated how to use the assembly kit [18] at the concrete example of the multi-threaded programming language MWL. Using the assembly kit has turned out to be very helpful in the identification of an appropriate security property. This application is also interesting because it shows how time-sensitive security can be specified in the assembly kit. For a different technique to address timing channels by explicit *tick*-events we refer to [11].

Bisimulation vs Trace-based Equivalence. The reader familiar with transition-system-based semantics might be surprised by the fact that the article relates a bisimulation-based property of programs with a trace-based one. It is well-known that small-step bisimulation makes more distinctions than trace-based equivalence. It is also well-known that trace-based properties are usually not compositional whereas bisimulation-based ones often are. Nevertheless, we have been able to prove correctness and completeness results for our translation of the security property.

What made these developments possible in the presence of the two major differences between the bisimulation-based and trace-based models? The crucial property is the deterministic nature of strong low-bisimulation. Indeed, bisimulation is defined on deterministic transitions ensuring that two bisimilar thread pools have the same branching behavior.[2] This property is necessary for guaranteeing *scheduler-independent security*. Two programs have to have identical branching behavior in order to be indistinguishable for the attacker under a scheduler-independent low-bisimulation. Otherwise, the two programs in the then and else branches, respectively, of an if statement with a

[2] Technically one can view the identifier *pid* of a thread that is chosen by the scheduler for the next transition as a label that is distinguished by the strong low-bisimulation.

secret condition could be used to leak the secret condition through observing the branching behavior ([27] shows how to implement this attack using the properties of a particular scheduler).

Although the determinism of bisimulation is the key feature to relating bisimulation-based and trace-based models, it is not crucial for the actual security definition of MWL (certain security properties can be defined through low determinism, as in [25, 24]). For example, if MWL had a non-deterministic choice operator $[]$ then the non-deterministic program $l := 0 \; [] \; l := 1$ would be considered secure under Definition 7. However, the two security definitions (Definition 7 and Definition 2 for MWL thread pools) would be no longer equivalent. Indeed, at no surprise, the completeness theorem (Theorem 4) would not hold. A counterexample is the program $\mathsf{if}\ h = 0\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ where $C_1 = l := 0; (l := 1 \; [] \; l := 2)$ and $C_2 = (l := 0; l := 1) \; [] \; (l := 0; l := 2)$. This program is considered secure under the trace-based model (Definition 2) but not secure according to the bisimulation-based Definition 7. Note that, whether one intuitively considers this program as secure or not depends very much on the model of computation one has in mind. For a detailed investigation of this close relation between notions of information flow and models of computation (notions of equivalence) we refer to [26].

Future Work. Plans for future work are centered around exploiting the connection between the two types of security that we have established in the present article. Promising directions include the adaption of intransitive security policies[3] for MWL basing on solutions that were proposed in the context of the assembly kit [20] and progress towards a development method that allows for the stepwise development starting from abstract specifications and ending with concrete programs (cf. [21] for recent progress on the refinement of information flow properties).

Another potentially interesting direction of research is to apply the reduction techniques of [27] combined with the results of this paper to reasoning about probabilistic security properties for event-based systems. Moreover, the integration of synchronization primitives for communication between multiple thread pools together with an according strengthening of the security predicate is another important goal for future work because it is a prerequisite for applying the results of this paper to truly distributed systems.

Acknowledgments

Thanks are due to Fabio Martinelli for encouraging discussions. We also wish to thank David Sands and Koen Claessen for useful feedback on bisimulation proofs. For useful comments on the presentation, we would like to thank Alexandra Heidger and Axel Schairer.

[3] Intransitive flow policies would provide a way to represent downgrading (and thus, e.g., secure encryption) in the multi-threaded while-language.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, January 1999.

[2] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 40–53, January 2000.

[3] J. Agat and D. Sands. On Confidentiality and Algorithms. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, May 2001. To appear.

[4] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.

[5] E. S. Cohen. Information Transmission in Computational Systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[6] E. S. Cohen. Information Transmission in Sequential Programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[7] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[8] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[9] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[10] R. Focardi and R. Gorrieri. A Classification of Security Properties for Process Algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[11] R. Focardi, R. Gorrieri, and F. Martinelli. Information Flow Analysis in a Discrete-Time Process Algebra. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 170–184, July 3–5 2000.

[12] S. N. Foley. A Universal Theory of Information Flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 116–122, Oakland, CA, April 27–29 1987.

[13] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 26–28 1982.

[14] J. Guttman and M. Nadel. "What Needs Securing?". In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 34–57, June 12-15 1988.

[15] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[16] D. M. Johnson and F. J. Thayer. Security and the Composition of Machines. In *Proceedings of the Computer Security Foundations Workshop*, pages 72–89, Franconia, NH, June 1988.

[17] K. R. M. Leino and R. Joshi. A Semantic Approach to Secure Information Flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[18] H. Mantel. Possibilistic Definitions of Security – An Assembly Kit –. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 185–199, Cambridge, UK, July 3–5 2000.

[19] H. Mantel. Unwinding Possibilistic Security Properties. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, LNCS 1895, pages 238–254, Toulouse, France, October 4-6 2000. Springer.

[20] H. Mantel. Information Flow Control and Applications – Bridging a Gap –. In J. N. Olivera and P. Zave, editors, *Proceedings of FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, LNCS 2021, pages 153–172, Berlin, Germany, March 12-16 2001. Springer.

[21] H. Mantel. Preserving Information Flow Properties under Refinement. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 13-16 2001. To appear.

[22] D. McCullough. Specifications for Multi-Level Security and a Hook-Up Property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 27–29 1987.

[23] J. McLean. A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 79–93, Oakland, CA, May 16–18 1994.

[24] A. Roscoe. CSP and Determinism in Security Modelling. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 114–127, Oakland, CA, May 1995.

[25] A. Roscoe, J. Woodcock, and L. Wulf. Non-interference through Determinism. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, LNCS 875, pages 33–53, Brighton, UK, November 7–9 1994. Springer.

[26] P. Ryan and S. Schneider. Process Algebra and Non-interference. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 214–227, Mordano, Italy, June 28–30 1999.

[27] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 200–215, Cambridge, UK, July 3–5 2000.

[28] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[29] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

[30] D. Sutherland. A Model of Information. In *9th National Computer Security Conference*, September 1986.

[31] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[32] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):1–21, 1996.

[33] J. T. Wittbold and D. M. Johnson. Information Flow in Nondeterministic Systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 144–161, Oakland, CA, May 1990.

[34] A. Zakinthinos and E. Lee. A General Theory of Security Properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 94–102, Oakland, CA, May 4–7 1997.