Flexible Scheduler-Independent Security

Heiko Mantel and Henning Sudbrock

Computer Science, TU Darmstadt, Germany,
{mantel,sudbrock}@cs.tu-darmstadt.de

Abstract We propose an approach to certify the information flow security of multi-threaded programs independently from the scheduling algorithm. A scheduler-independent verification is desirable because the scheduler is part of the runtime environment and, hence, usually not known when a program is analyzed. Unlike for other system properties, it is not straightforward to achieve scheduler independence when verifying information flow security, and the existing independence results are very restrictive. In this article, we show how some of these restrictions can be overcome. The key insight in our development of a novel scheduler-independent information flow property was the identification of a suitable class of schedulers that covers the most relevant schedulers. The contributions of this article include a novel security property, a scheduler independence result, and a provably sound program analysis.

1 Introduction

Whether a program can be entrusted secrets depends on the flow of information caused by running this program. *Noninterference* is a security property that characterizes secure information flow by the requirement that a program's output to untrusted sinks does not depend on secrets [1]. This requirement ensures that an attacker cannot conclude any information about secrets from the output that he can possibly observe, even if he has access to the full code of the program.

In order to obtain reliable analysis results, a noninterference analysis needs to properly respect the semantics of the given language. This raises the question of how to deal with aspects that influence a program's behavior, but that are outside the definition of the programming language's semantics. Examples are elements of the language whose behavior is not specified in the language's definition (e.g. native methods in Java) or elements of the runtime environment.

In this article, we focus on how to deal with a particular element of the runtime environment, namely the scheduler. Unlike for other properties, it is not sufficient to assume a possibilistic scheduler in a noninterference analysis, i.e. the scheduler that admits all possible scheduling choices. Secure information flow under a possibilistic scheduler need not imply that a program is secure for other schedulers because refining some part of a secure system's specification (such as the scheduler) may result in a system that violates security [2].

Many information flow analyses are scheduler dependent in the sense that they assume a particular scheduler, such that the analysis results are only valid if

Appeared in:
D. Gritzalis, B. Preneel, and M. Theoharidou (Eds.): ESORICS 2010, LNCS 6345, pp. 116–133, 2010.

the program is executed under this scheduler. For instance, a uniform scheduler is assumed in [3], a Round-Robin scheduler in [4], and a possibilistic scheduler in [5].

There are also a few approaches that support a scheduler-independent analysis. So far, there are two main approaches to defining information flow properties that are scheduler independent, firstly, requiring that a program's public output is deterministically determined by the program's public input and, secondly, requiring that a program's possible behaviors for any two inputs, which comprise identical public inputs, are stepwise indistinguishable to an observer of the program's public outputs. The first approach was introduced by Zdancewic and Myers, adapting the idea of defining secure information flow based on observational determinism to language-based security [6]. The second approach was used by Sabelfeld and Sands to define the so-called strong security property [7]. While both approaches provide a semantic basis for program analyses that are sound independently of the scheduling algorithm, they are far from satisfactory. The resulting security properties are very restrictive because they are violated by many intuitively secure programs. The main deficiency of security properties based on observational determinism is that they forbid nondeterminism in the publicly observable behavior of a program, albeit intuitively secure programs can have nondeterministic public output. Strong security suffers from a different problem. It requires a restrictive lock-step indistinguishability, which implies, for instance, that a program's execution time must not depend on secrets, even if such differences in the timing do not cause differences in the public output.

In this article, we propose a scheduler-independent security property that permits nondeterminism in a program's publicly observable behavior without requiring a restrictive lock-step indistinguishability. Our solution does not require non-standard modifications to the interface of schedulers (as in other approaches, e.g., [8,9]). In fact our approach is the first that is suitable for programs with nondeterministic publicly observable behavior whose runtime depends on secrets, while providing scheduler independence for common schedulers like Round-Robin and uniform schedulers (see Section 6 for a more detailed comparison). The existence of a scheduler-independent security property with these features is somewhat surprising given that Sabelfeld proved in [10] that strong security is the weakest property that implies information flow security for a natural class of schedulers. The key step in our development was the identification of a different class of schedulers, the *robust schedulers*, that also contains the most relevant schedulers.

In summary, our contributions include (1) the definition of a novel security property for multi-threaded programs, (2) the novel notion of robust schedulers, (3) a theorem showing that our security property is scheduler independent for robust schedulers, and (4) a provably sound, scheduler-independent program analysis for enforcing our security property. We illustrate the progress made by the security analysis of a small, but realistic example program. The proofs of all theorems in this article are made available on the authors' website. We expect that our improvements constitute a significant step towards more widely applicable information flow analyses for concurrent programs.

2 Preliminaries

In this article, we consider multi-threaded programs that communicate via shared memory. In this section, we leave the set *Com* of commands unspecified. It will be instantiated by a multi-threaded imperative programming language in Section 5.

2.1 Execution Model

A multi-threaded program executes threads from a pool, that we represent by a finite list of threads. The thread pool's size has no upper bound and varies during program execution as threads are removed upon termination and new threads may be spawned. Active threads are implicitly numbered consecutively by their position in the thread pool. The program memory is shared between all threads, and thereby provides a means for inter-thread communication.

A *thread configuration* is a pair $\langle com, mem\rangle \in (Com \cup \{\mathsf{stop}\}) \times (Var \to Val)$ that models a snapshot during the execution of a single thread. If $com = \mathsf{stop}$ then the thread has terminated, while if $com \in Com$ then this is the command that remains to be executed by the thread. The second element of a thread configuration, *mem*, models the current state of the program memory by assigning a value to each program variable, where *Var* is the set of variables and *Val* is a not further specified set of values. We denote the set $(Var \to Val)$ with *Mem*.

A *program configuration* is a pair $\langle thr, mem\rangle$, consisting of a *thread pool* $thr : \mathbb{N}_0 \to (Com \cup \{\mathsf{stop}\})$ and a *shared memory* $mem \in Mem$, that models a snapshot during the execution of a multi-threaded program. If $thr(k) = \mathsf{stop}$ then there is no thread at position k, while if $thr(k) \in Com$ then this is the command that remains to be executed by the kth thread. We define the size of a thread pool *thr* by $\sharp(thr) = |\{k \in \mathbb{N}_0 \mid thr(k) \neq \mathsf{stop}\}|$. We furthermore require that $thr(k) \neq \mathsf{stop}$ implies $thr(l) \neq \mathsf{stop}$ for all $l < k$, i.e. the thread pool has no gaps. We denote the set of all thread pools satisfying these requirements with *Progs*. Note that $\sharp(thr) = \min\{k \in \mathbb{N}_0 \mid thr(k) = \mathsf{stop}\}$ holds for all $thr \in Progs$.

To make scheduling explicit, we introduce system configurations. Formally, a *system configuration* is a triple $\langle thr, mem, sst\rangle$ such that $\langle thr, mem\rangle$ is a program configuration and $sst \in sSt$ is a *scheduler state*. Scheduler states and other aspects of scheduling will be introduced in Section 3.1. We use *Conf* to denote the set of all system configurations and introduce selector functions *getT*, *getM*, and *getS* to retrieve the elements from a system configuration, i.e., $getT(\langle thr, mem, sst\rangle) = thr$, $getM(\langle thr, mem, sst\rangle) = mem$, and $getS(\langle thr, mem, sst\rangle) = sst$.

To model execution steps, we introduce the judgment

$$conf \Rightarrow_{k,p} conf'$$

where $conf, conf' \in Conf$, $k \in \mathbb{N}_0$, and $0 < p \leq 1$. Intuitively, this judgment models that a transition from the system configuration *conf* to the system configuration $conf'$ is possible. The index k identifies the thread performing a computation step by its position in the thread pool $getT(conf)$. The probability of the transition is specified by the index p. Note that purely deterministic behavior can be modeled by restricting p to the singleton set $\{1\}$.

Derivability of the judgment for system configurations is defined based on two further judgments. The judgment $(sst, sin) \overset{k}{\rightsquigarrow}_p sst'$ models that the scheduler selects the kth thread with probability p. This decision may be based on sst, the state of the scheduler, and further input sin. The resulting scheduler state is sst'. The judgment $\langle com, mem\rangle \overset{\alpha}{\rightarrowtriangle} \langle com', mem'\rangle$ models that executing the command com in the memory mem results in the thread configuration $\langle com', mem'\rangle$. The label $\alpha \in Lab$ is an event from the set $Lab = \{new(coms) \mid coms \in Com^*\}$ that captures information about the creation of threads in the computation step. An event $new(coms)$ models that new threads are spawned to execute the commands in the list $coms$. We omit the label if no new threads were spawned.

Based on the above two judgments we can now model the stepwise execution of a multi-threaded program under a given scheduler by the following rule:

$$\frac{\begin{array}{cc}(sst, sin) \overset{k}{\rightsquigarrow}_p sst' & \langle thr(k), mem\rangle \overset{\alpha}{\rightarrowtriangle} \langle com', mem'\rangle \\ sin = obs(thr, mem) & thr' = update_k(thr, com', \alpha)\end{array}}{\langle\, thr, mem, sst\,\rangle \Rightarrow_{k,p} \langle\, thr', mem', sst'\,\rangle} \tag{1}$$

The third premise of the rule indicates that inputs to schedulers result from an observation of the program configuration (Section 3.1 will refine scheduler inputs). The function $update_k$ in the fourth premise updates the thread pool thr. In this article, we assume that spawned threads are inserted in the list of threads after the thread that executed the spawn operation. Moreover, if a thread terminates then it is removed from the list. For $\alpha = new(coms)$ and $k < \sharp(thr)$, the thread pool $update_k(thr, com, \alpha)$ is defined by $replace_k(thr, coms)$ if $com = \mathsf{stop}$ and by $replace_k(thr, [com]{::}coms)$ otherwise (where $::$ denotes list concatenation).[1]

2.2 Traces

A *trace* models a possible run of a program under some scheduler by a pair (str, dtr). The *system trace* $str : \mathbb{N}_0 \to Conf$ models the run of the program, where $str(0)$ is a snapshot of the system before starting its execution, and $str(k)$ is a snapshot of the system after k execution steps. The *decision trace* $dtr : \mathbb{N}_0 \to \mathbb{N}_0$ models the scheduler's decisions during the run, where $dtr(k)$ is the position of the thread selected for execution in the kth execution step.

Definition 1. *A* trace *is a pair* $tr = (str, dtr)$ *consisting of a* system trace $str : \mathbb{N}_0 \to Conf$ *and a* decision trace $dtr : \mathbb{N}_0 \to \mathbb{N}_0$.

We model the termination of a run by designated final configurations, namely those configurations in which the thread pool is empty.

Definition 2. *We call a trace* $tr = (str, dtr)$ terminating *if it reaches a final configuration, i.e. if* $\exists j \in \mathbb{N}_0 : \sharp(getT(str(j))) = 0$ *holds. The* length of a terminating trace *is* $\sharp(tr) = \min\{j \in \mathbb{N}_0 \mid \sharp(getT(str(j))) = 0\}$.

[1] For $k \in \mathbb{N}_0$, the partial function $replace_k : (Progs \times Com^*) \rightharpoonup Progs$ is defined by $replace_k(thr, [com_0, \ldots, com_{n-1}])(j) = com$ for $k < \sharp(thr)$ where $com = thr(j)$ if $j < k$, $com = com_{j-k}$ if $k \leq j < k + n$, and $com = thr(j - n)$ if $k + n \leq j$.

Given a set of traces Tr, we define the subset of terminating traces *by*

$$Tr{\Downarrow} = \{(str, dtr) \in Tr \mid \exists j \in \mathbb{N}_0 : \sharp(getT(str(j))) = 0\}.$$

The subset of traces terminating with memory *mem is defined by*

$$Tr{\Downarrow_{mem}} = \{tr \in Tr{\Downarrow} \mid getM(str(\sharp(tr))) = mem\}.$$

3 Scheduling and Scheduler-Specific Security

We capture scheduler-dependent information flow security by a noninterference-like information flow property that is parametric in the choice of the scheduler.

3.1 An Explicit Scheduler Model

During the execution of a multi-threaded program, the scheduler repeatedly decides which thread shall next proceed with the computation. Scheduling algorithms differ not only in how they make this decision, but also in the information on which they base their decision. For instance, a uniform scheduler needs to know the current number of threads in order to randomly choose among the available threads with equal probability. A Round-Robin scheduler iterates over the list of available threads in a cyclic fashion. Beyond knowing the number of threads, this requires that the scheduler remembers its scheduling choice from the previous step. A scheduler might even need to know part of the program's memory, for instance, in priority-based scheduling if priorities are first-class values that may be read and modified by the program itself. To cover these and various other possibilities, we assume that schedulers base their decision on their current internal state and their observation of the program configuration.

We leave the set of scheduler states sSt and the set of scheduler inputs sIn unspecified. We only assume that it is at least possible to retrieve the current number of threads from any given input $sin \in sIn$ and denote this number by $\sharp(sin)$. The scheduler's output is modeled by a natural number, indicating the position of the next thread to be run. The behaviour of a scheduler is modeled by a labeled transition relation, where a label $(sin, k, p) \in sIn \times \mathbb{N}_0 \times]0, 1]$ indicates that sin is the input to the scheduler, k is the position of the chosen thread, and p is the probability of this choice. We use probabilistic transitions in order to account for schedulers that are not deterministic like, e.g., uniform schedulers.

Definition 3. *A* scheduler *is a labeled transition system $(sSt, sst_0, sLab, \rightarrow)$ with initial state $sst_0 \in sSt$, label set $sLab = sIn \times \mathbb{N}_0 \times]0, 1]$, and transition relation $\rightarrow \subseteq sSt \times sLab \times sSt$ that satisfies the following properties:*

1. *if $(sst, (sin, k, p), sst') \in \rightarrow$, then $k < \sharp(sin)$;*
2. *for each triple (sst, sin, k) there is at most one probability $p \in]0, 1]$ and one scheduler state $sst' \in sSt$ such that $(sst, (sin, k, p), sst') \in \rightarrow$; and*
3. *the equality $\sum\{\!|p \mid \exists k, sst' : \ (sst, (sin, k, p), sst') \in \rightarrow|\!\} = 1$ holds for each pair (sst, sin) (where $\{\!| \ldots |\!\}$ denotes a multiset).*

The first property in Definition 3 ensures that a scheduler selects among the available threads, the second property ensures that a scheduler cannot choose

a single thread position with multiple probabilities or modify its internal state nondeterministically, and the third property ensures that in each state the probabilities of the transitions for a given input form a probability distribution.

Definition 4. *The judgment* $(sst, sin) \overset{k}{\rightsquigarrow}_p sst'$ *(from Section 2.1) is* derivable *for a scheduler* $(sSt, sst_0, sLab, \rightarrow)$ *if and only if* $(sst, (sin, k, p), sst') \in \rightarrow$.

Our model for schedulers is sufficiently expressive for common scheduling algorithms such as uniform, Round-Robin, and priority-based scheduling.

Example 1. A uniform scheduler can be modeled by the labeled transition system $UNI = (\{s\}, s, sLab, \rightarrow_{UNI})$, where $\rightarrow_{UNI}$ is defined by $(sst, (sin, k, p), sst') \in \rightarrow_{UNI}$ if and only if $k < \sharp(sin)$, $p = 1/\sharp(sin)$, and $sst' = sst = s$.

Example 2. A Round-Robin scheduler can be modeled by the labeled transition system $RR = (sSt_{RR}, sst_{RR,0}, sLab, \rightarrow_{RR})$, with $sSt_{RR} : \{choice, size\} \rightarrow \mathbb{N}_0$, $sst_{RR,0}(choice) = 0$, and $sst_{RR,0}(size) = 1$. The scheduler variables *choice* and *size* store from the previous step which thread position was selected and what size the thread pool had. The transition relation is defined by $(sst, (sin, k, p), sst') \in \rightarrow_{RR}$ if and only if $p = 1$, $k = (sst(choice) + 1 + (\sharp(sin) - sst(size))) \mod \sharp(sin)$, $sst'(choice) = k$, and $sst'(size) = \sharp(sin)$.[2]

We specify the traces modeling possible program runs under a given scheduler.

Definition 5. *For a scheduler* $\mathcal{S} = (sSt, sst_0, sLab, \rightarrow)$*, the* set of possible traces starting in a system configuration *conf is defined by* $(str, dtr) \in Tr_{\mathcal{S}}(conf)$ *if and only if*

$$str(0) = conf \wedge \forall j \in \mathbb{N}_0 : \left(\begin{array}{l} (\exists p \in]0,1] : \ str(j) \Rightarrow_{dtr(j),p} str(j+1)) \\ \vee \left(\begin{array}{l} \sharp(getT(str(j))) = 0 \\ \wedge\, str(j+1) = str(j) \wedge dtr(j) = 0 \end{array} \right) \end{array} \right)$$

Note that if a final configuration is reached, the program performs no further computation steps. We model this by requiring that from that point on the system trace infinitely often repeats the final configuration and the decision trace infinitely often repeats the value 0. We say that a *system configuration conf is terminating under a scheduler* $\mathcal{S}$ if $Tr_{\mathcal{S}}(conf) = Tr_{\mathcal{S}}(conf)\Downarrow$. *A thread pool thr is terminating* if for all $mem \in Mem$ the system configuration $\langle thr, mem, sst_0 \rangle$ is terminating under all schedulers $\mathcal{S}$ with initial scheduler state sst_0.

We use the probabilities of single execution steps to compute the probability that a terminating program run with a given sequence of scheduler decisions occurs when executing a program in a given configuration under a given scheduler:

Definition 6. *For a trace* $tr = (str, dtr) \in Tr_{\mathcal{S}}(conf)\Downarrow$ *we define the* probability of tr under the scheduler $\mathcal{S}$ *by* $\rho_{\mathcal{S}}(tr) = p_0 * \ldots * p_{\sharp(tr)-1}$*, where the* p_j *are the unique probabilities with* $str(j) \Rightarrow_{dtr(j),p_j} str(j+1)$ *for* $0 \leq j < \sharp(tr)$.

[2] Note that our condition on k ensures, firstly, that no thread is skipped if the current thread terminates (in this case $\sharp(sin) - sst(size)$ equals -1) and, secondly, that newly created threads obtain their term only after all other threads have been scheduled (in this case $\sharp(sin) - sst(size)$ equals the number of newly created threads).

3.2 Scheduler-Specific Security Property

We consider a security lattice with two security domains, *low* and *high*, where the requirement is that no information flows from *high* to *low*. This is the simplest policy capturing information flow security. A *domain assignment* is a function $dom : Var \to \{low, high\}$ that associates a security domain with each program variable. The resulting security requirement is that no information may flow from variables with domain *high* to variables with domain *low*.

We assume that attackers cannot directly access the values of high variables (i.e., access control works correctly). The following indistinguishability relation captures this upper bound on the observational capabilities of attackers.

Definition 7. *Two memories* $mem, mem' \in Mem$ *are* low-equal*, denoted by* $mem =_L mem'$*, if and only if* $mem(var) = mem'(var)$ *for all* $var \in Var$ *with* $dom(var) = low$*. We use* $[mem]_{=_L}$ *to denote the equivalence class* $\{mem' \in Mem \mid mem =_L mem'\}$*.*

Definition 8. A thread pool *thr* is $\mathcal{S}$-secure *for* $\mathcal{S} = (sSt, sst_0, sLab, \to)$ *if*

$$\sum_{mem' \in [mem]_{=_L}} \rho_{\mathcal{S}}(\langle thr, mem_1, sst_0 \rangle, mem') = \sum_{mem' \in [mem]_{=_L}} \rho_{\mathcal{S}}(\langle thr, mem_2, sst_0 \rangle, mem')$$

holds for all mem_1*,* mem_2*,* $mem \in Mem$ *with* $mem_1 =_L mem_2$*, where the value* $\rho_{\mathcal{S}}(conf, mem)$ *is the probability that a program run under the scheduler* $\mathcal{S}$ *that starts in the system configuration conf terminates with memory mem. It is defined by* $\rho_{\mathcal{S}}(conf, mem) = \sum_{tr \in Tr_{\mathcal{S}}(conf) \Downarrow_{mem}} \rho_{\mathcal{S}}(tr)$*.*

A command *com* is $\mathcal{S}$-secure *if the thread pool* thr_{com} *containing the single thread com is* $\mathcal{S}$*-secure (i.e.* $thr_{com}(0) = com$ *and* $thr_{com}(j) = \mathsf{stop}$ *for all* $j > 0$*).*

Our notion of $\mathcal{S}$-security guarantees that the probability that an $\mathcal{S}$-secure program terminates with given values of low variables is independent from the initial values of secrets. This means, $\mathcal{S}$-security implies that an attacker who can observe the initial and final values of low variables cannot conclude anything about high inputs. This implication still holds if the attacker knows the code of the program and the scheduling algorithm. Moreover, it also holds if the attacker can observe multiple runs of the program. Note that the number of execution steps of an $\mathcal{S}$-secure program may depend on the values of low variables (in contrast to, e.g., the bisimulation-based scheduler-specific security condition in [7]).

Remark 1. While we prefer to define $\mathcal{S}$-security using the probabilities of traces, an equivalent property could be defined using Markov chains, as, e.g., in [11].

4 Scheduler-Independent Information Flow Security

In this section, we present the novel information flow property that is the main contribution of this article. Our security property is scheduler independent in the sense that it implies $\mathcal{S}$-security for a large class of schedulers. We characterize this

class by the novel notion of robust schedulers and show that this class covers the most common scheduling algorithms. As we will illustrate, our security property is suitable for programs with nondeterministic behavior and whose runtime may depend on secrets. That is, it overcomes restrictions of the existing scheduler-independent security properties.

4.1 A Novel Security Property

We partition the set *Com* into *high commands* that definitely do not modify low variables and into *low commands* that potentially modify values of low variables.

Definition 9. *The* set of high commands $HCom \subset Com$ *is the largest set of commands such that if* $com \in HCom$ *then the following holds:*

$$\forall com' \in Com \cup \{\mathsf{stop}\} : \forall mem, mem' \in Mem : \forall com_0, \ldots, com_{n-1} \in Com :$$
$$\langle com, mem\rangle \xrightarrow{new([com_0,\ldots,com_{n-1}])} \langle com', mem'\rangle \implies$$
$$\big(mem =_L mem' \wedge com' \in HCom \cup \{\mathsf{stop}\} \wedge \forall j \in \{0,\ldots,n-1\}.\ com_j \in HCom\big)$$

The set of low commands $LCom \subset Com$ *is defined as* $Com \setminus HCom$.

We refer to threads executing high commands as *high threads* and to threads executing low commands as *low threads*. Note that a low thread becomes high after an execution step if the command that remains to be executed is high. High threads, by definition, cannot become low during a program's execution.

Low matches link the positions of corresponding low threads in thread pools.

Definition 10. *A* low match *of two thread pools* thr_1 *and* thr_2 *with the same number of low threads (i.e.* $|\{k_1 \in \mathbb{N}_0 \mid thr_1(k_1) \in LCom\}| = |\{k_2 \in \mathbb{N}_0 \mid thr_2(k_2) \in LCom\}|$*) is an order-preserving bijection with the domain* $\{k_1 \in \mathbb{N}_0 \mid thr_1(k_1) \in LCom\}$ *and the range* $\{k_2 \in \mathbb{N}_0 \mid thr_2(k_2) \in LCom\}$.

That is, a low match maps the position of the nth low thread in one thread pool to the position of the nth low thread in the other thread pool:

Theorem 1. *The low match of* thr_1 *and* thr_2 *is unique and given by the function*

$$l\text{-}match_{thr_1,thr_2}(k_1) =$$
$$\min\Big\{k_2 \in \mathbb{N}_0 \mid \big|\{l_1 \le k_1 \mid thr_1(l_1) \in LCom\}\big| = \big|\{l_2 \le k_2 \mid thr_2(l_2) \in LCom\}\big|\Big\}.$$

Due to space restrictions, the proof of the above theorem as well as the proofs of all other theorems in this article are provided on the authors' website.

We use the PER-approach [12] to define the novel security property, i.e., we define an indistinguishability relation on thread pools that is not reflexive, as it only relates thread pools to themselves that have secure information flow.

Definition 11. *A symmetric relation* R *on thread pools with an equal number of low threads is a* low bisimulation modulo low matching*, if whenever* $thr_1\ R\ thr_2$*,* $mem_1 =_L mem_2$*, and* $\langle thr_1(k_1), mem_1\rangle \xrightarrow{\alpha_1} \langle com_1, mem_1'\rangle$*, then*

1. *if* $thr_1(k_1) \in LCom$*, then there exist* com_2*,* mem_2'*, and* α_2 *with*
   (a) $\langle thr_2(k_2), mem_2\rangle \xrightarrow{\alpha_2} \langle com_2, mem_2'\rangle$*,*
   (b) $mem_1' =_L mem_2'$*, and*
   (c) $update_{k_1}(thr_1, com_1, \alpha_1) \;\; R \;\; update_{k_2}(thr_2, com_2, \alpha_2)$
   where $k_2 = l\text{-}match_{thr_1, thr_2}(k_1)$*; and*
2. *if* $thr_1(k_1) \in HCom$*, then* $update_{k_1}(thr_1, com_1, \alpha_1) \;\; R \;\; thr_2$*.*

The relation $\sim$ *is the union of all low bisimulations modulo low matching.*

Definition 12. A thread pool *thr* is FSI-secure *if* $thr \sim thr$*.* A command *com* is FSI-secure *if the thread pool* thr_{com} *(see Definition 8) is FSI-secure.*

We will show in Section 4.3 that all terminating FSI-secure programs are also $\mathcal{S}$-secure for any robust scheduler $\mathcal{S}$. This scheduler independence result motivates the expansion of the acronym FSI-security, which is *flexible scheduler-independent security*.

The following theorem shows that FSI-security is compositional.

Theorem 2. *Let* thr_1 *and* thr_2 *be FSI-secure thread pools. Then their* parallel composition $par(thr_1, thr_2)$ *is also FSI-secure, where*

$$par(thr_1, thr_2)(k) = \begin{cases} thr_1(k) & , \text{ if } k < \sharp(thr_1) \\ thr_2(k - \sharp(thr_1)) & , \text{ otherwise.} \end{cases}$$

The compositionality result is not only crucial for a modular analysis, but also illustrates that FSI-security is suitable for multi-threaded programs containing races: As it suffices that each individual thread of a program is FSI-secure, FSI-security imposes no restrictions on the relationships between variables occurring in concurrent threads. This constitutes a significant improvement over security properties based on observational determinism.

Moreover, FSI-security is suitable for programs whose runtime depends on confidential information. While FSI-security requires stepwise indistinguishability for low threads (Item *1* in Definition 11), no such requirement is imposed on a thread once it is high (Item *2* in Definition 11). This constitutes a major improvement over the strong security condition. In particular, unlike strong security, FSI-security is satisfied by every high command.

Theorem 3. *Let* $com \in HCom$*. Then com is FSI-secure.*

In summary, FSI-security overcomes serious deficiencies of the two main approaches to defining scheduler-independent security mentioned in Section 1.

4.2 The Class of Robust Schedulers

The essential idea of robust schedulers is that the scheduling order of low threads does not depend on the high threads in a thread pool. We formalize the class of robust schedulers in Definition 15 based on the auxiliary notions of thread purge functions (Definition 13) and of $\mathcal{S}$-simulations (Definition 14).

Definition 13. *The thread pool th-purge(thr) is defined by*

$$\textit{th-purge}(thr)(k_1) = thr\Big(\min\big\{k_2 \in \mathbb{N}_0 \mid k_1 = \big|\{l < k_2 \mid thr(l) \in LCom \cup \{\mathsf{stop}\}\}\big|\big\}\Big)$$

for all $k_1 \in \mathbb{N}_0$. We denote with th-purge(conf) the system configuration obtained from conf by replacing $getT(conf)$ with th-purge($getT(conf)$).

Intuitively, *th-purge*(*thr*) is obtained from *thr* by removing all high threads and leaving the order of low threads unchanged:

Theorem 4. *For a thread pool thr, th-purge(thr) contains no high threads and as many low threads as thr. Moreover, if $k \in \mathbb{N}_0$ with $thr(k) \in LCom$ then*

$$thr(k) = \textit{th-purge}(thr)(\textit{l-match}_{thr,\textit{th-purge}(thr)}(k)).$$

Definition 14. *Let $\mathcal{S} = (sSt, sst_0, sLab, \rightarrow)$ be a scheduler. An* $\mathcal{S}$-simulation *is a relation $<$ that relates arbitrary configurations with configurations that do not contain high threads, such that $conf_1 < conf_2$ and $conf_1 \Rightarrow_{k_1,p_1} conf_1'$ imply*

1. *if $getT(conf_1)(k_1) \in LCom$, then there exists $conf_2'$ with*
   (a) *$conf_2 \Rightarrow_{k_2,p_2} conf_2'$, where $k_2 = \textit{l-match}_{getT(conf_1),getT(conf_2)}(k_1)$ and $p_2 = p_1/\textit{l-prob}_{\mathcal{S}}(conf_1)$, and $\textit{l-prob}_{\mathcal{S}}(conf)$ denotes the probability that a low thread is selected by the scheduler $\mathcal{S}$ in the system configuration conf that is defined by $\textit{l-prob}_{\mathcal{S}}(\langle thr, mem, sst\rangle) = \sum\{\!|p \mid \exists k, sst' : thr(k) \in LCom \wedge (sst, obs(thr, mem)) \overset{k}{\rightsquigarrow}_p sst'|\!\}$, as well as*
   (b) *$conf_1' < \textit{th-purge}(conf_2')$; and*
2. *if $getT(conf_1)(k_1) \in HCom$, then $conf_1' < conf_2$.*

The relation $<_{\mathcal{S}}$ is the union of all $\mathcal{S}$-simulations.

Definition 15. *The scheduler $\mathcal{S} = (sSt, sst_0, sLab, \rightarrow)$ is* robust *if*

$$\langle thr, mem, sst_0\rangle <_{\mathcal{S}} \textit{th-purge}(\langle thr, mem, sst_0\rangle)$$

holds for each FSI-secure thread pool thr and each memory mem.

Intuitively, a scheduler is robust if the scheduling of low threads during a run of an FSI-secure thread pool remains unchanged when one removes all high threads from the thread pool. That is, the probability that the scheduler selects a low thread among all low threads in a configuration equals the probability to select the matching low thread if all high threads were removed (i.e., $p_2 = p_1/\textit{l-prob}_{\mathcal{S}}(conf_1)$). This is, in particular, the case for uniform and Round-Robin schedulers:

Theorem 5. *The uniform scheduler (see Example 1) is robust.*

Theorem 6. *The Round-Robin scheduler (see Example 2) is robust.*

Robust schedulers will only be employed in combination with observation functions that properly confine the interface between programs and schedulers:

Definition 16. *We call the observation function obs (introduced in Section 2.1)* confined*, if it satisfies the following property for all thread pools thr_1, thr_2 and for all memories mem_1, mem_2:*

$$(\sharp(thr_1) = \sharp(thr_2) \wedge mem_1 =_L mem_2) \implies obs(thr_1, mem_1) = obs(thr_2, mem_2)$$

Confined observation functions only provide information about the current number of threads and the values of public variables. This is sufficient for common schedulers like Round-Robin or priority-based schedulers.

Note that Definition 15 quantifies over all FSI-secure thread pools. This is essential. Quantifying over all thread pools (i.e., including ones that are not FSI-secure) would result in a significantly smaller class of robust schedulers, that, for instance, does not include Round-Robin and uniform schedulers.

4.3 Scheduler Independence Result

We are now ready to present the scheduler independence result:

Theorem 7. *Let thr be a terminating thread pool that is FSI-secure and let $\mathcal{S}$ be a robust scheduler under a confined observation function. Then the thread pool thr is $\mathcal{S}$-secure.*

Theorems 5 and 6 show that we obtain scheduler-independent information flow security for a practically relevant class of schedulers. As FSI-security overcomes restrictions of the two main approaches to scheduler-independent security (see Section 4.1), we expect that our results will contribute to more widely applicable information flow analyses for concurrent programs.

5 Security Analysis for a Multi-threaded Language

We use a simple multi-threaded imperative programming language supporting the dynamic creation of new threads for illustrating how to analyze concrete programs with respect to FSI-security. We define the set *Com* by the following grammar (using a set *Exp* of expressions that we do not specify further):

$$\begin{aligned} com ::= {} & \mathsf{skip} \mid var{:=}exp \mid com; com \mid \mathsf{if}\ (exp)\ \mathsf{then}\ com\ \mathsf{else}\ com\ \mathsf{fi} \\ & \mid \mathsf{while}\ (exp)\ \mathsf{do}\ com\ \mathsf{od} \mid \mathsf{spawn}(com, \ldots, com), \end{aligned}$$

where $var \in Var$ and $exp \in Exp$. The operational semantics for commands is formalized by a calculus for the judgment $\langle com, mem\rangle \xrightarrow{\alpha} \langle com', mem'\rangle$ introduced in Section 2.1. The derivation rules are as usual, we refrain from stating their definition due to space restrictions.

5.1 Security Type System

We present a security type system for our example language. This type system provides the basis for an automated scheduler-independent security analysis.

We type commands with types of the form (ass, stp), where $ass, stp \in \{low, high\}$. The intuition of the typing judgment $\vdash com : (ass, stp)$ is as follows: If $ass = high$, then neither the thread executing *com* nor the threads that are created due to spawn-commands within *com* assign to low variables, i.e., *com* is a high command. If $stp = low$, then the number of execution steps made by a thread executing *com* cannot depend on the values of high variables. However, the execution time of threads spawned by this thread may depend on high values.

The typing rules are displayed in Figure 1. We denote with $dom(exp)$ the security domain of an expression, where $dom(exp) = low$ if all variables occurring

$$[\mathsf{SKIP}]\frac{}{\vdash \mathsf{skip} : (high, low)} \qquad [\mathsf{ASS}]\frac{dom(exp) \sqsubseteq dom(var)}{\vdash var{:=}exp : (dom(var), low)}$$

$$[\mathsf{IF}]\frac{\vdash com_1 : (ass, stp) \quad \vdash com_2 : (ass, stp) \quad dom(exp) \sqsubseteq ass}{\vdash \mathsf{if}\ (exp)\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi} : (ass, stp \sqcup dom(exp))}$$

$$[\mathsf{WHILE}]\frac{\vdash com : (ass, stp) \quad stp \sqcup dom(exp) \sqsubseteq ass}{\vdash \mathsf{while}\ (exp)\ \mathsf{do}\ com\ \mathsf{od} : (ass, stp \sqcup dom(exp))}$$

$$[\mathsf{SPAWN}]\frac{\forall i \in \{0, \ldots, k-1\}.\ \vdash com_i : (ass, stp_i)}{\vdash \mathsf{spawn}(com_0, \ldots, com_{k-1}) : (ass, low)}$$

$$[\mathsf{SEQ}]\frac{\vdash com_1 : (ass_1, stp_1) \quad \vdash com_2 : (ass_2, stp_2) \quad stp_1 \sqsubseteq ass_2}{\vdash com_1; com_2 : (ass_1 \sqcap ass_2, stp_1 \sqcup stp_2)}$$

$$[\mathrm{SUB}]\frac{\vdash com : (ass', stp') \quad ass \sqsubseteq ass' \quad stp' \sqsubseteq stp}{\vdash com : (ass, stp)}$$

Figure 1. Security type system

in *exp* have domain *low*, and $dom(exp) = high$ otherwise. As usual for a two-level policy, we assume $low \sqsubseteq high$ and denote the greatest lower bound respectively least upper bound operator on security domains with $\sqcap$ and $\sqcup$, respectively. Note that subtyping is covariant in the first component of a type and contravariant in its second component (compare rule [SUB] in Figure 1).

Rule [ASS] forbids assignments from high to low variables, and rules [IF] and [WHILE] forbid assignments to low variables under high guards of conditionals and loops (compare, e.g., [13]). Furthermore, rules [IF] and [WHILE] ensure that commands containing high guards (and whose runtime might hence depend on the values of high variables) can only be typed if $stp = high$. Rule [SPAWN] allows to type programs that dynamically spawn threads: A spawn-command is typable with $stp = low$, as it is executed in a single execution step. Moreover, the command $\mathsf{spawn}(com_0, \ldots, com_{k-1})$ is only typable with $ass = high$ if each com_i is typable with $ass = high$. Rules [SEQ] and [WHILE] ensure that if a typable command assigns to low variables, then its runtime before such an assignment only depends on the values of low variables. This is essential for the soundness of the type system, as it ensures that lock-step execution is possible for low threads that correspond to each other under the low matching.[3]

Theorem 8. *If the judgment* $\vdash com : (ass, stp)$ *is derivable in the type system for some* $com \in Com$ *and* $ass, stp \in \{low, high\}$ *then com is FSI-secure.*

[3] Note that our security type system does not contain a rule for typing conditionals with high guards whose branches may contain assignments to low variables if the branches are related by an indistinguishability relation. Such a rule is, e.g., provided in [14], and could be soundly integrated into our type system. We refrain from such a rule here to ensure that there are no choice points when generating a type derivation.

Initial thread :
networkOut$_l$:="getStockPrices";
stockPrices$_l$:=networkIn$_l$;
spawn(*writeStockPricesToDatabase*);
networkOut$_l$:="getFundsPrices";
fundsPrices$_l$:=networkIn$_l$;
spawn(*writeFundsPricesToDatabase*);
spawn(*computeAccountOverview*)

writeStockPricesToDatabase:
i$_l$:=0;
while (i$_l$ < getSize(stockPrices$_l$)) do
 database$_l$:= database$_l$
 + getTitleAt(stockPrices$_l$, i$_l$)
 + getPriceAt(stockPrices$_l$, i$_l$);
 i$_l$:= i$_l$+1 od

writeFundsPricesToDatabase:
j$_l$:=0;
while (j$_l$ < getSize(fundsPrices$_l$)) do
 database$_l$:= database$_l$
 + getTitleAt(fundsPrices$_l$, j$_l$)
 + getPriceAt(fundsPrices$_l$, j$_l$);
 j$_l$:= j$_l$+1 od

computeAccountOverview:
k$_h$:= 0; overview$_h$:= "";
while (k$_h$ < getSize(userPortfolio$_h$)) do
 title$_h$:= getTitleAt(userPortfolio$_h$, k$_h$);
 if (isStock(title$_h$)) then
 price$_h$:= getPriceFor(stockPrices$_l$, title$_h$)
 ∗ getQuantityAt(userPortfolio$_h$, k$_h$)
 else
 price$_h$:= getPriceFor(fundsPrices$_l$, title$_h$)
 ∗ getQuantityAt(userPortfolio$_h$, k$_h$)
 fi;
 oldPrice$_h$:=getLastPrice(database$_l$, title$_h$);
 if (oldPrice$_h$ ≤ price$_h$)
 then tendency$_h$:= "up"
 else tendency$_h$:= "down"
 fi;
 overview$_h$:= overview$_h$ + title$_h$
 + price$_h$ + tendency$_h$;
 k$_h$:= k$_h$+1
od

Figure 2. Exemplary security analysis: implementation

Thus, due to the compositionality of FSI-security (Theorem 2), a thread pool *thr* is FSI-secure if $thr(k)$ is typable for each $k < \sharp(thr)$.

Note that the type systems proposed in [15,8,11,16] are similar to our type system as they restrict the assignments a program may perform after executing a conditional or a loop with a high guard. However, note that [15,11,16] only guarantee soundness for one scheduler-specific security property. Note also that [8] targets a language that allows dynamic thread creation (a typical feature of multi-threaded programming languages) only in a very limited form (no threads may be created inside loops), and the article assumes that threads idle after their termination instead of being removed from the thread pool. Our type system and its soundness result do not share these limitations. The scheduler independence result from [8] will be further compared to the result in this article in Section 6.

5.2 Exemplary Security Analysis

Consider the code fragment in Figure 2, which is part of an application managing personal finances. The program contacts two network-based services that provide up-to-date pricing information for stocks respectively funds (by writing to respectively reading from the variables networkOut$_l$ and networkIn$_l$). The program appends the retrieved information to the information in the variable database$_l$ that contains historical pricing information for future reference. Moreover, using the novel data and historical data already present in database$_l$, the program

generates an overview of the user's custody account. Those three activities are spawned in new threads (*writeStockPricesToDatabase*, *writeFundsPricesToDatabase*, and *computeAccountOverview*) to improve the interactivity of the overall program and not block computations following this code fragment. In our example, we model the network requests, the data retrieved from the network, the data stored in the database, and the generated overview with string values. The data encoded in those string values is accessed by the program using selector expressions like, for instance, getLastPrice.

The subscripts of variables indicate whether a variable is classified as low (l) or as high (h). Information in the database and information retrieved from the network services is public and classified as *low*, while the user's portfolio and the report created based on the portfolio are confidential and classified as *high*.

Applying the type system to the program proves that the program is FSI-secure: The initial thread as well as the threads *writeStockPricesToDatabase* and *writeFundsPricesToDatabase* are typable with the type (low, low), while the thread *computeAccountOverview* is typable with the type $(high, high)$.

Note that the program is rejected by existing analyses that guarantee security for common schedulers. Observational determinism [6] is violated, as the order in which entries are written to the database depends on the order in which the threads *writeStockPricesToDatabase* and *writeFundsPricesToDatabase* are scheduled. Strong security [7] is violated as the runtime of the loop in the thread *computeAccountOverview* depends on confidential information. The soundness of the type system together with the scheduler independence result guarantee that the order of the database entries never depends on confidential information when using a robust scheduler.

6 Related Work

An overview on information flow security in a multi-threaded setting is provided in [17]. Here, we focus on approaches that cover the problem of scheduling.

Most approaches assume a particular scheduling algorithm. In consequence, their results do not necessarily generalize to other schedulers. Several approaches consider a scheduler that selects threads purely nondeterministically (for instance, [5,18,19,20,21]). Uniform schedulers are assumed in [3,11], and a Round-Robin scheduler is assumed in [4,22].

There are only a few approaches to scheduler-independent information flow security. In the following, we discuss those approaches in more detail.

The idea of *observational determinism* goes back to McLean [23] and Roscoe et al. [24,25], who proposed security properties not at the level of a programming language, but more abstractly for specifications. The idea was adapted to a language-based setting in [6,26]. Observational determinism requires that public observations of program executions are deterministic regardless of the interleaving of threads and the values of secret variables. If this requirement is satisfied, restricting the possible interleavings by assuming a concrete scheduler cannot result in a dependency of public observations on secrets. Observational determinism has the drawback that it forbids useful nondeterminism which oc-

curs, for instance, when multiple threads append data to the same public variable (as in the example program from Section 5.2).

Sabelfeld and Sands [7] introduce *strong security*, which is scheduler-independent for a natural class of schedulers. Strong security is quite restrictive, as it requires that the runtime of a program must not depend on secret data. This drawback appeared unavoidable because [10] proved that the strong security condition is the weakest compositional property that implies information flow security for the natural class of schedulers. Hence, the strong security condition was used, despite its restrictiveness, as the foundation of many later developments (e.g., [27,28,29]) and has been generalized in various ways, e.g., for distributed systems [27] or to control declassification [30]. While [7] proposes a type system that can transform some insecure programs into strongly secure programs, only a subset of the intuitively secure programs is amenable to this approach (for instance, the type system does not transform the example program from Section 5.2 into a strongly secure program).

The combining calculus [20] is a first step towards combining approaches based on observational determinism and strong security as it allows the combination of different analysis techniques in a security analysis. However, a scheduler-independence result has not yet been established for the combining calculus.

Boudol and Castellani [8] propose a security type system for *controlled thread systems* that consist of a thread pool and a scheduler. If a controlled thread system is typable, then the thread pool is secure under the scheduler. In contrast to this article, the approach requires the size of a thread pool to remain fixed during a program run: dynamic thread creation is not supported, and threads remain in the thread pool upon termination (and may still be selected by the scheduler). Boudol and Castellani argue that if the termination of certain threads would be signaled to the scheduler, then controlled thread systems writing public variables cannot be typed. This is a non-standard restriction, as schedulers typically use the number of live threads when choosing the next thread.

As a different approach to relax the security property while remaining scheduler-independent, [9,31,32] propose to use non-standard schedulers that provide a customized interface to the scheduled threads. Via two special commands, programs can *hide* (and at a later point *unhide*) a thread; the scheduler guarantees that during the execution of hidden threads no other thread is scheduled. The approach allows to securely execute programs containing threads that assign to low variables after performing computations whose runtime depends on high data (hiding the thread during those computations), but at the cost that a scheduler with a non-standard interface must be used. Such threads are rejected by our security property, as they may cause information leakage when being executed under currently available schedulers.

Another approach that prevents scheduling during computations whose runtime depends on secrets is followed by [22]. It provides a program transformation that introduces yield-statements into a program instructing the scheduler to select another thread, such that no yield-statements occur during computations depending on secrets, and rescheduling only occurs after a yield-statement. The

approach is implemented for a Round-Robin scheduler, but the article argues that it is applicable for a wide class of schedulers. The transformation entails that computations on secrets block all remaining threads. This is particularly critical when these computations are time-consuming. In contrast, our approach allows any computations to be interleaved with the executions of other threads.

In the following we discuss two approaches that investigate scheduler-independence on the level of system specifications. Van der Meyden and Zhang [33] adapt security conditions for asynchronous systems to scheduled synchronous systems. They consider schedulers whose decisions do not depend on secret actions and show that the security properties are *scheduler-implementation independent* in the sense that a system satisfies a property under one implementation of a scheduler if and only if it satisfies the property under all of the scheduler's possible implementations. Note that this differs from requiring that security holds under different schedulers. Moreover, [33] prove that if the security definitions are satisfied for all deterministic schedulers, then they are also satisfied for all nondeterministic schedulers. Probabilistic schedulers are not considered.

Also when considering protocols the scheduling might impact security. In particular, the hidden random choice of a secret value in a security protocol could be revealed if the protocol's schedule depends on the choice's outcome. As a solution, [34] proposes to make random choices invisible to the scheduler by annotating protocol actions with labels and requiring that (a) the possible actions after a secret random choice obtain the same label and (b) the only input to the scheduler are the labels of the schedulable actions. The development is based on the probabilistic process algebra CCS_p. It differs from our approach as it requires program annotations that guide the possible choices of the scheduler.

7 Conclusion

Scheduler-independent information flow security is an important problem for concurrent programs, but previously existing solutions are far from being satisfactory: They are either very restrictive in the sense that they reject many intuitively secure programs, or in the sense that they require non-standard modifications of schedulers and their interfaces. Both restrictions limit the applicability of information flow security analyses for concurrent programs.

Aiming at more widely applicable information flow analyses, we developed the novel security condition FSI-security. FSI-security overcomes deficiencies of the existing approaches to scheduler-independent security while still achieving scheduler independence for common schedulers. Our scheduler independence result is rather surprising in the light of the impossibility result from [10], which states that for a natural class of schedulers a compositional scheduler-independent security condition must be at least as restrictive as the strong security condition. The key insight for obtaining a security property that is less restrictive yet compositional and scheduler-independent for relevant schedulers was the identification of a different class of schedulers, the *robust schedulers*, which is also natural but smaller than the class in [10].

Acknowledgments. The authors thank Dave Sands for helpful comments in the early phase of this research project and the anonymous reviewers for their suggestions. This work was funded by the DFG (German Research Foundation) in the Computer Science Action Program. This article reflects only the authors' views, and the DFG and the authors are not liable for any use that may be made of the information contained therein.

References

1. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: 3rd IEEE Symposium on Security and Privacy. pp. 11–20. IEEE (1982)
2. Jacob, J.: On the Derivation of Secure Components. In: 10th IEEE Symposium on Security and Privacy. pp. 242–247. IEEE (1989)
3. Volpano, D., Smith, G.: Probabilistic Noninterference in a Concurrent Language. Journal of Computer Security 7(2,3), 231–253 (1999)
4. Russo, A., Hughes, J., Naumann, D.A., Sabelfeld, A.: Closing Internal Timing Channels by Transformation. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. pp. 120–135. LNCS, vol. 4435, Springer (2006)
5. Smith, G., Volpano, D.: Secure Information Flow in a Multi-threaded Imperative Language. In: 25th ACM Symposium on Principles of Programming Languages. pp. 355–364. ACM (1998)
6. Zdancewic, S., Myers, A.C.: Observational Determinism for Concurrent Program Security. In: 16th IEEE Computer Security Foundations Workshop. pp. 29–43. IEEE (2003)
7. Sabelfeld, A., Sands, D.: Probabilistic Noninterference for Multi-threaded Programs. In: 13th IEEE Computer Security Foundations Workshop. pp. 200–214. IEEE (2000)
8. Boudol, G., Castellani, I.: Noninterference for Concurrent Programs and Thread Systems. Theoretical Computer Science 281(1-2), 109–130 (2002)
9. Russo, A., Sabelfeld, A.: Securing Interaction between Threads and the Scheduler. In: 19th IEEE Computer Security Foundations Workshop. pp. 177–189. IEEE (2006)
10. Sabelfeld, A.: Confidentiality for Multithreaded Programs via Bisimulation. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. pp. 260–274. LNCS, vol. 2890, Springer (2003)
11. Smith, G.: Probabilistic Noninterference through Weak Probabilistic Bisimulation. In: 16th IEEE Computer Security Foundations Workshop. pp. 3–13. IEEE (2003)
12. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. In: Swierstra, S.D. (ed.) ESOP 1999. pp. 50–59. LNCS, vol. 1576, Springer (1999)
13. Volpano, D., Smith, G., Irvine, C.: A Sound Type System for Secure Flow Analysis. Journal of Computer Security 4(2,3), 167–188 (1996)
14. Mantel, H., Sands, D.: Controlled Declassification Based on Intransitive Noninterference. In: Chin, W. (ed.) APLAS 2004. pp. 129–145. LNCS, vol. 3302, Springer (2004)
15. Smith, G.: A New Type System for Secure Information Flow. In: 14th IEEE Computer Security Foundations Workshop. pp. 115–125. IEEE (2001)
16. Matos, A.A., Boudol, G., Castellani, I.: Typing Noninterference for Reactive Programs. Journal of Logic and Algebraic Programming 72(2), 124–156 (2007)

17. Sabelfeld, A., Myers, A.C.: Language-based Information-Flow Security. IEEE Journal on Selected Areas in Communication 21(1), 5–19 (2003)
18. Sabelfeld, A.: The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) PSI 2001. pp. 225–239. LNCS, vol. 2244, Springer (2001)
19. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure Information Flow by Self-Composition. In: 17th IEEE Computer Security Foundations Workshop. pp. 100–114. IEEE (2004)
20. Mantel, H., Sudbrock, H., Kraußer, T.: Combining Different Proof Techniques for Verifying Information Flow Security. In: Puebla, G. (ed.) LOPSTR 2006. pp. 94–110. LNCS, vol. 4407, Springer (2006)
21. Mantel, H., Reinhard, A.: Controlling the What and Where of Declassification in Language-Based Security. In: De Nicola, R. (ed.) ESOP 2007. pp. 141–156. LNCS, vol. 4421, Springer (2007)
22. Russo, A., Sabelfeld, A.: Security for Multithreaded Programs under Cooperative Scheduling. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. pp. 474–480. LNCS, vol. 4378, Springer (2006)
23. McLean, J.D.: Proving Noninterference and Functional Correctness using Traces. Journal of Computer Security 1(1), 37–57 (1992)
24. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through Determinism. In: Gollmann, D. (ed.) ESORICS 1994. pp. 33–53. LNCS, vol. 875, Springer (1994)
25. Roscoe, A.W.: CSP and Determinism in Security Modelling. In: 16th IEEE Symposium on Security and Privacy. pp. 114–127. IEEE (1995)
26. Huisman, M., Worah, P., Sunesen, K.: A Temporal Logic Characterisation of Observational Determinism. In: 19th IEEE Computer Security Foundations Workshop. pp. 3–15. IEEE (2006)
27. Mantel, H., Sabelfeld, A.: A Unifying Approach to the Security of Distributed and Multi-threaded Programs. Journal of Computer Security 11(4), 615–676 (2003)
28. Focardi, R., Rossi, S., Sabelfeld, A.: Bridging Language-Based and Process Calculi Security. In: Sassone, V. (ed.) FoSSaCS 2005. pp. 299–315. LNCS, vol. 3441, Springer (2005)
29. Köpf, B., Mantel, H.: Transformational Typing and Unification for Automatically Correcting Insecure Programs. International Journal of Information Security 6(2-3), 107–131 (2007)
30. Lux, A., Mantel, H.: Declassification with Explicit Reference Points. In: Backes, M., Ning, P. (eds.) ESORICS 2009. pp. 69–85. LNCS, vol. 5789, Springer (2009)
31. Barthe, G., Rezk, T., Russo, A., Sabelfeld, A.: Security of Multithreaded Programs by Compilation. In: Biskup, J., Lopez, J. (eds.) ESORICS 2007. pp. 2–18. LNCS, vol. 4734, Springer (2007)
32. Russo, A., Sabelfeld, A.: Securing Interaction between Threads and the Scheduler in the Presence of Synchronization. Journal of Logic and Algebraic Programming 78(7), 593–618 (2009)
33. van der Meyden, R., Zhang, C.: Information Flow in Systems with Schedulers. In: 21st IEEE Computer Security Foundations Symposium. pp. 301–312. IEEE (2008)
34. Chatzikokolakis, K., Palamidessi, C.: Making Random Choices Invisible to the Scheduler. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. pp. 42–58. Springer (2007)