Scheduler-Independent Declassification

Alexander Lux, Heiko Mantel, and Matthias Perner

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

Abstract The controlled declassification of secrets has received much attention in research on information-flow security, though mostly for sequential programming languages. In this article, we aim at guaranteeing the security of concurrent programs. We propose the novel security property WHAT&WHERE that allows one to limit what information may be declassified where in a program. We show that our property provides adequate security guarantees independent of the scheduling algorithm (which is non-trivial due to the refinement paradox) and present a security type system that reliably enforces the property. In a second scheduler-independence result, we show that an earlier proposed security condition is adequate for the same range of schedulers. These are the first scheduler-independence results in the presence of declassification.

1 Introduction

When giving a program access to secrets, one would like to know that the program does not leak them to untrusted sinks. Such a confidentiality requirement can be formalized by information-flow properties like, e.g., *noninterference* [12].

Noninterference-like properties require that a program's output to untrusted sinks is independent of secrets. Such a lack of dependence obviously ensures that public outputs do not reveal any secrets. While being an adequate characterization of confidentiality, the requirement is often too restrictive. The desired functionality of a program might inherently require some correlation between secrets and public output. Examples are password-based authentication mechanisms (a response to an authentication attempt depends on the secret password), encryption algorithms (a cipher-text depends on the secret plain-text), and online stores (electronic goods shall be kept secret until they have been ordered).

Hence, it is necessary to relax noninterference-like properties such that a deliberate release of some secret information becomes possible. While this desire has existed since the early days of research on information-flow control (e.g. in the Bell/La Padula Model secrets can be released by so called trusted processes [8]), solutions for controlling declassification are just about to achieve a satisfactory level of maturity (see [33] for an overview). However, research on declassification has mostly focused on sequential programs so far, while controlling declassification in multi-threaded programs is not yet equally well understood.

Generalizing definitions of information-flow security for sequential programs to security properties that are suitable for concurrent systems is known to be non-trivial. Already in the eighties, Sutherland [34] and McCullough [23] proposed

J. Gibbons and P. Nogueira, MPC 2012, LNCS 7342, pp. 25–47, 2012.

noninterference-like properties for distributed systems. These were first steps in a still ongoing exploration of sensible definitions of information-flow security [19]. The information-flow security of multi-threaded programs, on which we focus in this article, is also non-trivial. Due to the refinement paradox [14], the scheduling of threads requires special attention. In particular, it does not suffice to simply assume a possibilistic scheduler, because a program might have secure information-flow if executed with the fictitious possibilistic scheduler, but be insecure if executed, e.g., with a Round-Robin or uniform scheduler.

Our first main contribution is the formal definition of two schemas for noninterference-like properties for multi-threaded programs. Our schemas $\text{WHAT}^{\mathfrak{s}}$ and $\text{WHAT\&WHERE}^{\mathfrak{s}}$ are parametric in a scheduler model $\mathfrak{s}$. Both schemas can be used to capture confidentiality requirements, but they differ in how declassification is controlled. If the scheduler is known then $\mathfrak{s}$ can be specified concretely and, after instantiating one of our schemas with $\mathfrak{s}$, one obtains a property that adequately captures information-flow security for this scheduler.

However, often the concrete scheduler is not known in advance. While, in principle, one could leave the scheduler parametric and use, e.g., $\forall\mathfrak{s}.\text{WHAT}^{\mathfrak{s}}$ as security condition, such a universal quantification over all possible schedulers is rather inconvenient, in program analysis as well as in program construction. Fortunately, an explicit universal quantification over schedulers can be avoided.

Our second main contribution is the definition of a novel security condition WHAT&WHERE and a scheduler-independence result, which shows that WHAT&WHERE implies $\text{WHAT\&WHERE}^{\mathfrak{s}}$ for all possible scheduler models $\mathfrak{s}$. A compositionality result shows that our novel property is compatible with compositional reasoning about security. Based on this result, we derive a security type system for verifying our novel security property efficiently.

Our third main contribution is a scheduler-independence result showing that our previously proposed property WHAT_1 [20] implies $\text{WHAT}^{\mathfrak{s}}$ for all $\mathfrak{s}$.

Previous scheduler-independence results were limited to information-flow properties that forbid declassification (e.g. [31,36,22]). With this article, we close this gap by developing the first scheduler-independence results for information-flow properties that support controlled declassification. Scheduler independence provides the basis for verifying security without knowing the scheduler under which a program will be run. Our scheduler-independence results also reduce the conceptual complexity of constructing secure programs. They free the developer from having to consider concrete schedulers when reasoning about security.

Proofs of all theorems in this article are available on the authors' web-pages.

2 Preliminaries

2.1 Multi-threaded Programs

Multi-threaded programs perform computations in concurrent threads that can communicate with each other, e.g. via shared memory. When the number of threads exceeds the number of available processing units, scheduling becomes necessary. Usually, the schedule for running threads is determined dynamically

at run-time based on previous scheduling decisions and on observations about the current configuration, such as the number of currently active threads.

In this article, we focus on multi-threaded programs that run on a single-core CPU with a shared memory for inter-thread communication. In this section, we present our model of program execution (a small-step operational semantics), our model of scheduler decisions (a labeled transition system), and an integration of these two models. The resulting system model is similar to the one in [22].

Semantics of Commands and Expressions. We assume a set of *commands* $\mathcal{C}$, a set of *expressions* $\mathcal{E}$, a set of *program variables* $\mathcal{V}\mathit{ar}$, and a set of *values* $\mathcal{V}\mathit{al}$. We leave these sets underspecified, but give example instantiations in Section 2.2.

We define the set of *memory states* by the function space $\mathcal{M}\mathit{em} = \mathcal{V}\mathit{ar} \to \mathcal{V}\mathit{al}$. A function $m \in \mathcal{M}\mathit{em}$ models which values are currently stored in the program variables. We define the set of *program states* by $\mathcal{C}_\epsilon = \mathcal{C} \cup \{\epsilon\}$. A program state from $\mathcal{C}$ models which part of the program remains to be executed while the special symbol ϵ models termination. We define the set of *thread pools* by $\mathcal{C}^*$ (i.e. the set of finite lists of commands). Each command in a thread pool is the program state of an individual thread in a multi-threaded program. We refer to threads by their position $k \in \mathbb{N}_0$ in a thread pool $\mathit{thr} \in \mathcal{C}^*$. If a thread is uniquely determined by $\mathit{thr}[k]$, i.e. the command at position k, then we sometimes refer to the thread by this command. We define $\#(\mathit{thr})$ to equal the number of threads in the thread pool $\mathit{thr} \in \mathcal{C}^*$. The list $\langle c_0, c_1, \ldots, c_{n-1}\rangle$ with $c_0, c_1, \ldots, c_{n-1} \in \mathcal{C}$ models a thread pool with n threads. The list $\langle\rangle$ models the empty thread pool. Note that the symbol ϵ does not appear in thread pools.

We model *evaluation of expressions* by the function $\mathit{eval} : \mathcal{E} \times \mathcal{M}\mathit{em} \to \mathcal{V}\mathit{al}$, where $\mathit{eval}(e, m)$ equals the value to which $e \in \mathcal{E}$ evaluates in $m \in \mathcal{M}\mathit{em}$.

We model *execution steps* by judgments of the form $\langle c_1, m_1\rangle \xrightarrow{\alpha} \langle c_2, m_2\rangle$ where $c_1 \in \mathcal{C}$, $c_2 \in \mathcal{C}_\epsilon$, $m_1, m_2 \in \mathcal{M}\mathit{em}$, and $\alpha \in \mathcal{C}^*$. Intuitively, this judgment models that a command c_1 is executed in a memory state m_1 resulting in a program state c_2 and a memory state m_2. The label $\alpha \in \mathcal{C}^*$ carries information about threads spawned by the execution step. If the execution step does not spawn new threads then $\alpha = \langle\rangle$ holds, otherwise we have $\alpha = \langle c_0, c_1, \ldots, c_{n-1}\rangle$ where $c_0, c_1, \ldots, c_{n-1} \in \mathcal{C}$ are the threads spawned in this order.

We assume deterministic commands, i.e. for each $c_1 \in \mathcal{C}$ and $m_1 \in \mathcal{M}\mathit{em}$, there exists exactly one tuple $(\alpha, c_2, m_2) \in \mathcal{C}^* \times \mathcal{C}_\epsilon \times \mathcal{M}\mathit{em}$ such that $\langle c_1, m_1\rangle \xrightarrow{\alpha} \langle c_2, m_2\rangle$ is derivable. As an alternative notation for the effect of a command on the memory, we define the function $[\![\bullet]\!] : \mathcal{C} \to (\mathcal{M}\mathit{em} \to \mathcal{M}\mathit{em})$ by $[\![c_1]\!](m_1) = m_2$ iff $\exists c_2 \in \mathcal{C}_\epsilon.\ \exists \alpha \in \mathcal{C}^*.\ \langle c_1, m_1\rangle \xrightarrow{\alpha} \langle c_2, m_2\rangle$.

As a notational convention, we use $v \in \mathcal{V}\mathit{al}$ to denote values, $x \in \mathcal{V}\mathit{ar}$ to denote variables, $m \in \mathcal{M}\mathit{em}$ to denote memory states, $c \in \mathcal{C}_\epsilon$ to denote program states, $e \in \mathcal{E}$ to denote expressions, $\mathit{thr} \in \mathcal{C}^*$ to denote thread pools, and $k \in \mathbb{N}_0$ to denote positions of threads.

Scheduler Model. We present a parametric scheduler model that can be instantiated for a wide range of schedulers. For modeling the behavior of schedulers, we use labeled transition systems as described below.

We assume a set of *scheduler states* $\mathcal{S}$ and a set of possible *scheduler inputs* In. Scheduler states model the memory of a scheduler and scheduler inputs model the input to the scheduler by the environment. We leave the set In underspecified, but require that any $in \in \mathit{In}$ reveals at least the number of active threads in the current thread pool and denote this number by $\#(in)$.

We define the set of *scheduler decisions* by $\mathcal{Dec} = \mathit{In} \times \mathbb{N}_0 \times [0;1]$. Intuitively, a scheduler decision $(in, k, p) \in \mathcal{Dec}$ models that the scheduler selects the k^{th} thread with the probability p given the scheduler input in. The special case $p = 1$ models a deterministic decision.

Definition 1. *A* scheduler model $\mathfrak{s}$ *is a labeled transition system* $(\mathcal{S}, s_0, \mathcal{Dec}, \rightarrow)$*, where* $\mathcal{S}$ *is a set of scheduler states,* $s_0 \in \mathcal{S}$ *is an initial state,* $\mathcal{Dec}$ *is the set of scheduler decisions, and* $\rightarrow \subseteq \mathcal{S} \times \mathcal{Dec} \times \mathcal{S}$ *is a transition relation such that:*

1. $\forall (s_1, (in, k, p), s_2) \in \rightarrow \,.\, (k < \#(in) \wedge p \neq 0)$
2. $\forall s_1 \in \mathcal{S}.\, \forall in \in \mathit{In}.\, \left(\#(in) > 0 \Longrightarrow \left(\sum_{(s_1,(in,k,p),s_2)\in\rightarrow} p\right) = 1\right)$
3. $\forall s_1, s_2, s_2' \in \mathcal{S}.\, \forall in \in \mathit{In}.\, \forall k \in \mathbb{N}_0.\, \forall p, p' \in \,]0;1].$
   $(((s_1, (in, k, p), s_2) \in \rightarrow) \wedge ((s_1, (in, k, p'), s_2') \in \rightarrow) \Longrightarrow p = p' \wedge s_2 = s_2')$

For a scheduler model $\mathfrak{s}$*, we write* $(s_1, in) \overset{k}{\underset{p}{\rightsquigarrow}}_{\mathfrak{s}} s_2$ *iff* $(s_1, (in, k, p), s_2) \in \rightarrow$.

Conditions 1 and 2 ensure that a scheduler model definitely selects some thread from the current thread pool. Condition 3 ensures that the probability of a scheduler decision and the resulting scheduler state are uniquely determined by the original scheduler state, the scheduler input, and the selected thread.

Our notion of scheduler models is suitable for expressing a wide range of schedulers, including Round-Robin schedulers as well as uniform schedulers.

For simplicity of presentation we consider only scheduler models without redundant states. Formally, we define the bisimilarity of scheduler states coinductively by a symmetric relation $\sim\, = \mathcal{S} \times \mathcal{S}$ that is the largest relation such that for all $dec \in \mathcal{Dec}$ and for all $s_1, s_1', s_2 \in \mathcal{S}$, if $s_1 \sim s_1'$ and $(s_1, dec, s_2) \in \rightarrow$ then there exists a scheduler state $s_2' \in \mathcal{S}$ with $(s_1', dec, s_2') \in \rightarrow$ and $s_2 \sim s_2'$. We require that the equivalence classes of $\sim$ are singleton sets, i.e. $\forall s, s' \in \mathcal{S}.\, (s \sim s' \Longrightarrow s = s')$, which means that there are no redundant states. Note that any given scheduler model can be transformed into one that satisfies this constraint by using the equivalence classes of $\sim$ as scheduler states.

As a notational convention, we use $in \in \mathit{In}$ to denote scheduler inputs, $p \in [0;1]$ to denote probabilities, and $s \in \mathcal{S}$ to denote scheduler states. For brevity, we often write *scheduler* instead of scheduler model.

Integration into a System Model. We now present the system model which defines the interaction between threads and a scheduler.

We define the set of *observation functions* by the function space $\mathcal{Obs} = (\mathcal{C}^* \times \mathcal{Mem}) \rightarrow \mathit{In}$. A function $obs \in \mathcal{Obs}$ models the input to a scheduler for a given thread pool and memory state. We define the set of *system configurations* by $\mathcal{Cnf} = \mathcal{C}^* \times \mathcal{Mem} \times \mathcal{S}$. Intuitively, a system configuration $\langle\!| thr, m, s |\!\rangle \in \mathcal{Cnf}$ models the current state of a multi-threaded program in a run-time environment.

We model *system steps* by judgments of the form $cnf_1 \Rightarrow^{\mathfrak{s}}_{k,p} cnf_2$, where $cnf_1, cnf_2 \in \mathcal{Cnf}$ and $(k,p) \in \mathbb{N}_0 \times]0;1]$. Intuitively, this judgment models that, in system configuration cnf_1, the scheduler selects the k^{th} thread with probability p and that this results in cnf_2. We define the rule for deriving this judgment by:

$$[\text{SysStep}]\frac{\begin{array}{ll}(s_1, in) \overset{k}{\rightsquigarrow}^{\mathfrak{s}}_{p} s_2 & \langle\!| thr_1[k], m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c_2, m_2 |\!\rangle \\ in = obs(thr_1, m_1) & thr_2 = update_k(thr_1, c_2, \alpha)\end{array}}{\langle\!| thr_1, m_1, s_1 |\!\rangle \Rightarrow^{\mathfrak{s}}_{k,p} \langle\!| thr_2, m_2, s_2 |\!\rangle}$$

The two premises on the left hand side require the selection of the k^{th} thread with probability p by scheduler $\mathfrak{s}$ given the scheduler input $obs(thr_1, m_1)$. The third premise requires that the execution step of thread $thr_1[k]$ spawns new threads α and results in program state c_2 and memory state m_2. The fourth premise requires that the resulting thread pool thr_2 is obtained by $update_k(thr_1, c_2, \alpha)$.

Intuitively, $update_k$ replaces the program state at a position k by a program state c_2 and inserts newly created threads (i.e. α) after c_2. Formally, we define $update_k(thr, c, \alpha)$ by $sub(thr, 0, k-1) :: \langle c \rangle :: \alpha :: sub(thr, k+1, \#(thr)-1)$ if $c \neq \epsilon$, and otherwise by $sub(thr, 0, k-1) :: \alpha :: sub(thr, k+1, \#(thr)-1)$, where $::$ is the append operator that has the empty list $\langle\rangle$ as neutral element and $sub(thr, i, j)$ equals the list of threads i to j, i.e. $sub(thr, i, j) = \langle thr[i] \rangle :: sub(thr, i+1, j)$ if $i \leq j < \#(thr)$, and $sub(thr, i, j) = \langle\rangle$ otherwise.

We define the auxiliary function $stepsTo^{\mathfrak{s}} : (\mathcal{Cnf} \times \mathfrak{P}(\mathcal{Cnf})) \to \mathfrak{P}(\mathbb{N}_0 \times]0;1])$ by $stepsTo^{\mathfrak{s}}(cnf_1, Cnf) = \{(k,p) \mid \exists cnf_2 \in Cnf.\ cnf_1 \Rightarrow^{\mathfrak{s}}_{k,p} cnf_2\}$.

That is, applying the function $stepsTo^{\mathfrak{s}}$ to cnf_1 and Cnf returns the labels of all possible system steps from $cnf_1 \in \mathcal{Cnf}$ to some configuration in Cnf.

We call a property $P : \mathcal{Cnf} \to Bool$ an *invariant* under $\mathfrak{s}$ if $P(cnf_1)$ and $cnf_1 \Rightarrow^{\mathfrak{s}}_{k,p} cnf_2$ imply $P(cnf_2)$ for all $cnf_1, cnf_2 \in \mathcal{Cnf}$ and $(k,p) \in \mathbb{N}_0 \times]0;1]$.

As a notational convention, we use $cnf \in \mathcal{Cnf}$ to denote system configurations. Moreover, we introduce the selectors $pool(cnf) = thr$, $mem(cnf) = m$, and $sst(cnf) = s$ for decomposing a system configuration $cnf = \langle\!| thr, m, s |\!\rangle$.

2.2 Exemplary Programming Language

We define security on a semantic level. However, to give concrete examples we introduce a simple multi-threaded while language with dynamic thread creation. We define $\mathcal{E}$ and $\mathcal{C}$ of our example language by:

$$\begin{array}{l} e ::= v \mid x \mid op(e, \ldots, e) \\ c ::= \mathsf{skip}_\iota \mid x :=_\iota e \mid c;c \\ \qquad \mid \mathsf{spawn}_\iota(c, \ldots, c) \mid \mathsf{if}_\iota\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \mathsf{while}_\iota\ e\ \mathsf{do}\ c\ \mathsf{od} \end{array}$$

Some commands carry a label $\iota \in \mathbb{N}_0$ that we will use to identify program points.

The operational semantics for our language defines which instances of the judgment $\langle\!| c_1, m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c_2, m_2 |\!\rangle$ are derivable. The only notable aspect of the semantics is the label α. If the top-level command is $\mathsf{spawn}_\iota(c_0, \ldots, c_{n-1})$, then we have $\alpha = \langle c_0, \ldots, c_{n-1} \rangle$ while, otherwise, $\alpha = \langle\rangle$ holds.

For readability, we also use infix instead of prefix notation for expressions.

2.3 Attacker Model and Security Policies

A security policy describes what information a user is allowed to know based on a classification of information according to its confidentiality. We use sets of *security domains* to model different degrees of confidentiality. *Domain assignments* associate each program variable with a security domain.

Definition 2. *A* multi-level security policy *(brief: mls-policy) is a triple* $(\mathcal{D}, \leq, dom)$*, where* $\mathcal{D}$ *is a finite set of security domains,* $\leq$ *is a partial order on* $\mathcal{D}$*, and* $dom : \mathcal{Var} \to \mathcal{D}$ *is a domain assignment.*

Intuitively, $d \not\leq d'$ with $d, d' \in \mathcal{D}$ models that no information must flow from the security domain d to the security domain d'.

A *d-observer* is a user who is allowed to observe a variable $x \in \mathcal{Var}$, only if $dom(x) \leq d$. Hence, he can distinguish two memory states only if they differ in the value of at least one variable x with $dom(x) \leq d$. Dual to the ability to distinguish memory states is the following *d-indistinguishability.*

Definition 3. *Two memory states* $m \in \mathcal{Mem}$ *and* $m' \in \mathcal{Mem}$ *are* d-equal *for* $d \in \mathcal{D}$ *(denoted:* $m =_d m'$*), iff* $\forall x \in \mathcal{Var}.\ (dom(x) \leq d \Longrightarrow m(x) = m'(x))$.

An *attacker* is a d-observer who tries to get information that he must not know. In terms of d-indistinguishability, this means that an attacker tries to distinguish initially d-equal memory states by running programs. Conversely, a program is intuitively secure, if running this program does not enable a d-observer to distinguish any two initial memory states that are d-equal. This intuition will be formalized by security properties in Section 3.

For the rest of the article, we assume that $(\mathcal{D}, \leq, dom)$ is an mls-policy.

2.4 Auxiliary Concepts for Relations

For any relation $R \subseteq A \times A$, there is at least one subset A' of A (namely $A' = \emptyset$) such that the restricted relation $R_{|A'} = R \cap (A' \times A')$ is an equivalence relation on A'. We characterize the subsets $A' \subseteq A$ for which $R_{|A'}$ constitutes an equivalence relation by a predicate $EquivOn_A \subseteq \mathfrak{P}(A \times A) \times \mathfrak{P}(A)$ that we define by $EquivOn_A(R, A')$ if and only if $R_{|A'}$ is an equivalence relation on A'.

In our definitions of security, we will use *partial equivalence relations* (brief: *pers*), i.e. binary relations that are symmetric and transitive but that need not be reflexive (see Sections 3 and 4.2). For each per $R \subseteq A \times A$, there is a unique maximal set $A' \subseteq A$ such that $EquivOn_A(R_{|A'}, A')$ holds. This maximal set is the set $A_{R,refl} = \{e \in A \mid e\ R\ e\}$, i.e. the subset of A on which R is reflexive.

Theorem 1. *If* $R \subseteq A \times A$ *is a per on a set* A *then* $EquivOn_A(R_{|A_{R,refl}}, A_{R,refl})$ *holds and* $\forall A' \subseteq A.\ (EquivOn_A(R_{|A'}, A') \Longrightarrow A' \subseteq A_{R,refl})$.

For brevity, we will use the symbol R instead of $R_{|A'}$ when this does not lead to ambiguities. In particular, we will write $EquivOn_A(R, A')$ meaning that $EquivOn_A(R_{|A'}, A')$ holds. Moreover, if $R \subseteq A \times A$ is a per, we will use $[e]_R$ to refer to the equivalence classes of an element $e \in A_{R,refl}$ under $R_{|A_{R,refl}}$.

Finally, we define a partial function $classes_A : \mathfrak{P}(A \times A) \rightharpoonup \mathfrak{P}(\mathfrak{P}(A))$ by $classes_A(R) = \{[e]_R \mid e \in A_{R,refl}\}$ if R is a per, while $classes_A(R)$ is undefined if R is not a per. That is, if R is a per, then $classes_A(R)$ equals the set of all equivalence classes of R (meaning the equivalence classes of $R_{|A_{R,refl}}$).

If the set A is clear from the context we write *classes* instead of $classes_A$.

3 Declassification in the Presence of Scheduling

A declassification is the deliberate release of secrets or, in other words, an intentional violation of an mls-policy. Naturally, such a release of secrets must be rigorously constrained to prevent unintended information leakage.

Example 1. Online music shops rely on not giving out songs for free. Hence, songs are only delivered to a user after he has paid. However, often downsampled previews are offered without payment to any user for promotion. The following example program shall implement this functionality.

$$P_1 = \mathsf{if}_1 \text{ paid then out:=}_2\text{song else out:=}_3\text{downsample(song, bitrate) fi}$$

Consider an mls-policy with two domains *low* and *high*, and the total order $\leq$ with $high \not\leq low$. The domain assignment *dom* is defined such that $dom(\mathsf{song}) = high$ and $dom(\mathsf{out}) = low$ hold. Intuitively, this mls-policy means that song is confidential with respect to out. The program P_1 intuitively satisfies the requirement that any user may receive a downsampled preview, while only a user who has paid may receive the full song. Note that some information about the confidential song is released in both branches of P_1, i.e. a declassification occurs. However, what information is released differs for the two branches. ◊

As this example shows, an adequate control of declassification needs to respect what information (the full song or the preview) is released and where this release occurs (e.g., after payment has been checked by the program). This corresponds to the W-aspects *What* and *Where* that we address in this article. The W-aspects of declassification were first introduced in [21] and form the basis for a taxonomy of approaches to controlling declassification [33].

Before presenting our schema WHAT&WHERE$^{\mathsf{s}}$ for scheduler-specific security properties that control what is declassified where (see Section 3.3), we introduce the simpler schema WHAT$^{\mathsf{s}}$ (see Section 3.2) for controlling what is declassified. We show in Section 3.4 that WHAT&WHERE$^{\mathsf{s}}$ implies WHAT$^{\mathsf{s}}$ and also satisfies the so called prudent principles of declassification from [33].

3.1 Escape Hatches and Immediate Declassification Steps

As usual, we use pairs $(d, e) \in \mathcal{D} \times \mathcal{E}$, so called *escape hatches* [29], to specify what information may be declassified. Intuitively, (d, e) allows a d-observer to peek at the value of e, even if in e occurs a variable x with $dom(x) \not\leq d$. Hence, an escape hatch might enable a d-observer to distinguish memory states although they are d-equal. Dual to this ability is the following notion of (d, H)-*equality*.

Definition 4. *Two memory states m and m' are* (d, H)-equal *for $d \in \mathcal{D}$ and a set of escape hatches $H \subseteq \mathcal{D} \times \mathcal{E}$ (denoted: $m \sim_d^H m'$), iff $m =_d m'$ and $\forall (d', e) \in H.\ (d' \leq d \Longrightarrow (eval(e, m) = eval(e, m')))$ hold.*

We employ program points to restrict where declassification may occur. For each program, we assume a set of *program points* $\mathcal{PP} \subseteq \mathbb{N}_0$ and a function $pp : \mathcal{C} \to \mathcal{PP}$ that returns a program point for each sub-command of the program. Moreover, we assume that program points are unique within a program.

For our example language, we use the labels ι to define the function pp. For instance, $pp(\mathsf{out}{:=_2}\mathsf{song}) = 2$ and $pp(\mathsf{if}_1\ \mathsf{paid\ then}\ \ldots\ \mathsf{else}\ \ldots\ \mathsf{fi}) = 1$ hold. As sequential composition does not carry a label ι, we define $pp(c_1;c_2) = pp(c_1)$. Note that, after unwinding a loop, multiple sub-commands in a program state might be associated with the same program point. This results from copying the body of a while loop in the operational semantics if the guard evaluates to true.

We augment escape hatches with program points from $\mathcal{PP}$ and call the resulting triples *local escape hatches*. Like an escape hatch $(d, e) \in \mathcal{D} \times \mathcal{E}$, a local escape hatch $(d, e, \iota) \in \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ intuitively allows a d-observer to peek at the value of e. However, (d, e, ι) allows this only while the command at program point ι is executed. We use a set $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ to specify at which program points a d-observer may peek at which values. For Example 1, a natural set of local escape hatches would be $\{(low, \mathsf{downsample(song,\ bitrate)}, 3), (low, \mathsf{song}, 2)\}$.

Definition 5. *A* local escape hatch *is a triple $(d, e, \iota) \in \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$. We call a set of local escape hatches $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$* global *(denoted: Global(lH)) if $(d, e, \iota) \in lH$ implies $(d, e, \iota') \in lH$ for all $d \in \mathcal{D}$, $e \in \mathcal{E}$, and $\iota, \iota' \in \mathcal{PP}$.*

To aggregate the information that may be declassified at a given program point, we define the filter function $htchLoc : \mathfrak{P}(\mathcal{D} \times \mathcal{E} \times \mathcal{PP}) \times \mathcal{PP} \to \mathfrak{P}(\mathcal{D} \times \mathcal{E})$ by $htchLoc(lH, \iota) = \{(d, e) \in \mathcal{D} \times \mathcal{E} \mid (d, e, \iota) \in lH\}$. Given a set of points $PP \subseteq \mathcal{PP}$, we use $htchLoc(lH, PP)$ as a shorthand notation for $\bigcup\{htchLoc(lH, \iota) \mid \iota \in PP\}$. Note that if lH is global then $\forall \iota, \iota' \in \mathcal{PP}.\ (htchLoc(lH, \iota) = htchLoc(lH, \iota'))$.

We call a command an *immediate d-declassification command* for a set of escape hatches $H \subseteq \mathcal{D} \times \mathcal{E}$ if its next execution step might reveal information to a d-observer that he should not learn according to the mls-policy, but that may permissibly be released to him due to some escape hatch in H.

Definition 6. *The predicate IDC_d on $\mathcal{C} \times \mathfrak{P}(\mathcal{D} \times \mathcal{E})$ is defined by*

$$IDC_d(c, H) \Longleftrightarrow \left[\begin{array}{c} (\exists m, m' \in \mathcal{M}\!\mathit{em}.\ m =_d m' \wedge [\![c]\!](m) \neq_d [\![c]\!](m')) \\ \wedge\ (\forall m, m' \in \mathcal{M}\!\mathit{em}.\ m \sim_d^H m' \Longrightarrow [\![c]\!](m) =_d [\![c]\!](m')) \end{array}\right]$$

The predicate IDC_d characterizes the immediate d-declassification commands for each set of escape hatches H. The predicate requires, firstly, that a release of secrets could, in principle, occur (i.e. for some pair of d-equal memories, the next step results in memories that are not d-equal) and, secondly, that no more information is released than allowed by the escape hatches (i.e. for all pairs of (d, H)-equal memories, the next step must result in d-equal memories).

Remark 1. If $IDC_d(c, htchLoc(lH, \iota))$ and $c \in \mathcal{C}$ is the command at program point $\iota \in \mathcal{PP}$ then c either has the form $x{:=_\iota}e$ or the form $x{:=_\iota}e; c'$. $\Diamond$

All concepts defined in this section are monotonic in the set of escape hatches, and the empty set of escape hatches is equivalent to forbidding declassification.

Theorem 2. *For all $d \in \mathcal{D}$ and $H, H' \subseteq \mathcal{D} \times \mathcal{E}$ the following propositions hold:*

1. $\forall m, m' \in \mathcal{Mem}.\ ((\neg(m \sim_d^{H'} m') \wedge H' \subseteq H) \Longrightarrow \neg(m \sim_d^{H} m'))$;
2. $\forall m, m' \in \mathcal{Mem}.\ (m \sim_d^{\emptyset} m' \Longleftrightarrow m =_d m')$;
3. $\forall c \in \mathcal{C}.\ ((IDC_d(c, H') \wedge H' \subseteq H) \Longrightarrow IDC_d(c, H))$; *and*
4. $\forall c \in \mathcal{C}.\ \neg(IDC_d(c, \emptyset))$.

A command is not a *d-declassification command* if its next execution step does not reveal any information to a d-observer that he cannot observe directly.

Definition 7. *The predicate NDC_d on $\mathcal{C}$ is defined by*

$$NDC_d(c) \Longleftrightarrow (\forall m, m' \in \mathcal{Mem}.\ m =_d m' \implies [\![c]\!](m) =_d [\![c]\!](m'))$$

Note that $NDC_d(c)$ cannot hold if $IDC_d(c, H)$ holds for some $H \subseteq \mathcal{D} \times \mathcal{E}$. If c leaks beyond what H permits then neither $IDC_d(c, H)$ nor $NDC_d(c)$ holds.

We use $\iota \in \mathcal{PP}$ to denote program points, $H \subseteq \mathcal{D} \times \mathcal{E}$ to denote sets of escape hatches, and $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ to denote sets of local escape hatches.

3.2 The Security Conditions WHAT[s]

Security can be characterized based on pers (brief for partial equivalence relations, see Section 2.4). Following this approach, one defines a program to be secure if it is related to itself by a suitable per [30]. Consequently, the set of secure programs for a per $R \subseteq A \times A$ is $\bigcup classes_A(R)$. We will characterize confidentiality by pers that relate two thread pools only if they yield indistinguishable observations for any two initial configurations that must remain indistinguishable. Which configurations must remain indistinguishable depends on the observer's security domain d and on the set H of available escape hatches. We make this explicit by annotating pers with d and H (as, e.g., in $R_{d,H}$).

Definition 8. *Let $d \in \mathcal{D}$ and $H \subseteq \mathcal{D} \times \mathcal{E}$. The* lifting of a relation $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *to a relation $R^{\uparrow}_{d,H} \subseteq \mathcal{Cnf} \times \mathcal{Cnf}$ is $R^{\uparrow}_{d,H} = (R_{d,H} \times \sim_d^H \times \sim)$.*

Note that, if two configurations cnf and cnf' are related by $R^{\uparrow}_{d,H}$ then they look the same to a d-observer because $mem(cnf) \sim_d^H mem(cnf')$ implies $mem(cnf) =_d mem(cnf')$. Moreover, the lifting of a per to the set $\mathcal{Cnf}$ results, again, in a per.

Proposition 1. *If $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ is a per, then $R^{\uparrow}_{d,H} \subseteq \mathcal{Cnf} \times \mathcal{Cnf}$ is a per.*

Towards a Scheduler-specific Security Condition. Even if two configurations cnf and cnf' look the same to a d-observer, he might be able to infer in which of the configurations a program run must have started based on the observations that he makes during the run. For instance, he can exclude the possibility that the run started in cnf' if he makes an observation that is incompatible with all configurations that are reachable from cnf'. In this case, he obtains information about the actual initial configuration from the fact that certain observations are impossible if the program is run under a given scheduler.

In addition, an attacker might obtain information about the initial configuration from the probability of observations. For instance, if he makes certain observations quite often, when running the program in some initial configuration (which remains fixed and is initially unknown to the attacker), but the likelihood of this observation would be rather low if cnf' were the initial configuration, then the attacker can infer that cnf' is probably not the unknown initial configuration.[1]

We aim at defining a security property that rules out deductions of information about secrets based on the possibility as well as the probability of observations. We will focus on the latter aspect in the following because deductions based on possibilities are just a special case of deductions based on probabilities.

The probability of moving from a configuration cnf to some configuration in a set Cnf depends not only on the program, but also on the scheduler $\mathfrak{s}$.

Definition 9. *The function* $prob^{\mathfrak{s}} : \mathcal{C}nf \times \mathfrak{P}(\mathcal{C}nf) \to [0;1]$ *is defined by:*

$$prob^{\mathfrak{s}}(cnf, Cnf) = \sum_{(k,p) \in stepsTo^{\mathfrak{s}}(cnf, Cnf)} p \ .$$

We will use the function $prob^{\mathfrak{s}}$ in our definition of WHAT$^{\mathfrak{s}}$ to capture that the likelihood of certain observations is the same in two given configurations.

If strict multi-level security were our goal then we could define security based on a per that relates two thread pools thr and thr' only if any two configurations $\langle thr, m, s\rangle$ and $\langle thr', m', s'\rangle$ with $m =_d m'$ and $s \sim s'$ cause indistinguishable observations. As we aim at permitting declassification, the situation is more involved. After a declassification occurred, a d-observer might be allowed to obtain information about the initial configuration that he cannot infer without running the program. However, such inferences should be strictly limited by the exceptions to multi-level security specified by a given set of escape hatches.

WHAT$^{\mathfrak{s}}$. We are now ready to define information-flow security. For each scheduler model $\mathfrak{s}$, we propose a security condition WHAT$^{\mathfrak{s}}$ that restricts declassification according to the constraints specified by a set of escape hatches. Following the per-approach, we define a multi-threaded program as WHAT$^{\mathfrak{s}}$-secure if it is related to itself by some relation $R_{d,H}$ that satisfies the following property.

Definition 10. *Let* $d \in \mathcal{D}$ *be a security domain and* $H \subseteq \mathcal{D} \times \mathcal{E}$ *be a set of escape hatches. An* $\mathfrak{s}$-specific strong (d, H)-bisimulation *is a per* $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *that fulfills the following two conditions:*

1. $\forall (cnf, cnf') \in R^{\uparrow}_{d,H}.\ \forall Cls \in classes(R^{\uparrow}_{d,H}).$
   $prob^{\mathfrak{s}}(cnf, Cls) = prob^{\mathfrak{s}}(cnf', Cls)$
2. *the property* $\lambda cnf \in \mathcal{C}nf.\ (cnf \in \bigcup classes(R^{\uparrow}_{d,H}))$ *is an invariant under* $\mathfrak{s}$.

Condition 1 in Definition 10 ensures that if a single computation step is performed in two related configurations cnf and cnf' under a scheduler $\mathfrak{s}$ then each equivalence class of $R^{\uparrow}_{d,H}$ is reached with the same probability from the two

[1] By increasing the number of runs such inferences are possible with high confidence, even if the difference between observed frequency and expected frequency is small.

configurations. Condition 2 ensures that all configurations that can result after a computation step are again contained in some equivalence class of $R^{\uparrow}_{d,H}$. This lifts Condition 1 from individual steps to entire runs. The two conditions ensure that if two configurations are related by $R^{\uparrow}_{d,H}$ (which means they must remain indistinguishable for a d-observer who may use the escape hatches in H) then they, indeed, remain indistinguishable when the program is run.

Definition 11. *A thread pool* $thr \in \mathcal{C}^*$ has secure information flow *for* $(\mathcal{D}, \leq, dom)$ *and* $H \subseteq \mathcal{D} \times \mathcal{E}$ *under* $\mathfrak{s}$ *(brief:* $thr \in \mathrm{WHAT}^{\mathfrak{s}}$*) iff for each* $d \in \mathcal{D}$ *there is a set* $H' \subseteq H$ *and a relation* $R_{d,H'} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *such that* $(thr\ R_{d,H'}\ thr)$ *holds, and such that* $R_{d,H'}$ *is an* $\mathfrak{s}$-specific strong (d, H')-bisimulation.

Definition 11 ensures that if $thr \in \mathrm{WHAT}^{\mathfrak{s}}$ and $m \sim^{H}_{d} m'$ and $s \sim s'$ then the configurations $\langle thr, m, s\rangle$ and $\langle thr, m', s'\rangle$ yield indistinguishable observations for d while the multi-threaded program thr is executed under $\mathfrak{s}$.

$\mathrm{WHAT}^{\mathfrak{s}}$ will serve as the basis of our first scheduler-independence result in Section 4. More concretely, we will show that our previously proposed security condition WHAT_1 [20] implies $\mathrm{WHAT}^{\mathfrak{s}}$ for a wide range of schedulers. Moreover, we will use $\mathrm{WHAT}^{\mathfrak{s}}$ when arguing that our second security condition $\mathrm{WHAT\&WHERE}^{\mathfrak{s}}$ adequately controls what is declassified (see Section 3.4).

3.3 The Security Conditions WHAT&WHERE$^{\mathfrak{s}}$

We employ local escape hatches to specify where a particular secret may be declassified. The annotations of pers are adapted accordingly by replacing H with a set lH of local escape hatches. Moreover a set of program points $PP \subseteq \mathcal{PP}$ is added as third annotation (resulting in $R_{d,lH,PP}$). The set PP will be used to constrain local escape hatches in the definition of $\mathrm{WHAT\&WHERE}^{\mathfrak{s}}$.

Definition 12. *Let* $d \in \mathcal{D}$, $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$, *and* $PP \subseteq \mathcal{PP}$. *The lifting of a relation* $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *to a relation* $R^{\uparrow}_{d,lH,PP} \subseteq \mathcal{Cnf} \times \mathcal{Cnf}$ *is defined by* $R^{\uparrow}_{d,lH,PP} = (R_{d,lH,PP} \times \sim^{H}_{d} \times \sim)$, *where* $H = htchLoc(lH, PP)$.

Proposition 2. *If* $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *is a per then* $R^{\uparrow}_{d,lH,PP} \subseteq \mathcal{Cnf} \times \mathcal{Cnf}$ *also is a per.*

Note that $\langle thr, m, s\rangle\ R^{\uparrow}_{d,lH,PP}\ \langle thr', m', s'\rangle$ implies that $m \sim^{htchLoc(lH,PP)}_{d} m'$ holds. This means that each variable $x \in \mathcal{Var}$ has the same value in m as in m' if x is visible for a d-observer (i.e. $m =_d m'$). Moreover, an expression $e \in \mathcal{E}$ has the same value in m as in m' if it may be declassified to d according to lH for at least one of the program points in PP (i.e. if $\exists (d', e, \iota) \in lH.\ (d' \leq d \wedge \iota \in PP)$).

Towards Controlling Where Declassification Occurs. If $NDC_d(c)$ holds then the next step of the command c respects strict multi-level security (i.e. no declassification to security domain d occurs in this step). If $IDC_d(c, H)$ holds then the next step of c might declassify information to d, and any such declassification is authorized by the escape hatches in H. However, if neither $NDC_d(c)$

nor $IDC_d(c, H)$ is true then there are memory states $m, m' \in \mathcal{Mem}$ such that $m \sim_d^H m'$ holds while $[\![c]\!](m) =_d [\![c]\!](m')$ does not hold. This means that information might be leaked whose declassification is not permitted by H.

In our definition of the security condition, we need to rule out this third possibility, i.e. $\neg IDC_d(c, H) \wedge \neg NDC_d(c)$ where H is the set of escape hatches that are enabled. Which escape hatches are enabled in a given computation step depends on the set of local escape hatches and on the set of program points that might cause the computation step.

The set of program points that might cause a transition from a configuration cnf to some configuration in a set Cnf depends on the scheduler.

Definition 13. *The function* $pps^{\mathfrak{s}} : (\mathcal{Cnf} \times \mathfrak{P}(\mathcal{Cnf})) \to \mathfrak{P}(\mathcal{PP})$ *is defined by:*

$$pps^{\mathfrak{s}}(cnf, Cnf) = \{pp(cnf[k]) \mid (k, p) \in stepsTo^{\mathfrak{s}}(cnf, Cnf)\} \quad .$$

Using $pps^{\mathfrak{s}}$, we define which hatches might be relevant for a computation step.

Definition 14. *The function* $htchs^{\mathfrak{s}} : (\mathfrak{P}(\mathcal{D} \times \mathcal{E} \times \mathcal{PP}) \times \mathcal{Cnf} \times \mathfrak{P}(\mathcal{Cnf})) \to \mathfrak{P}(\mathcal{D} \times \mathcal{E})$ *is defined by* $htchs^{\mathfrak{s}}(lH, cnf, Cnf) = htchLoc(lH, pps^{\mathfrak{s}}(cnf, Cnf))$.

WHAT&WHERE$^{\mathfrak{s}}$. We are now ready to introduce our second schema for scheduler-specific security conditions. Unlike WHAT$^{\mathfrak{s}}$, WHAT&WHERE$^{\mathfrak{s}}$ allows one to control where a particular declassification can occur. This combined control of the W-aspects *What* and *Where* is needed, for instance, in Example 1.

Like in Section 3.2, we define a class of pers on thread pools to characterize indistinguishability from the perspective of a d-observer. A program is then defined to be secure under a scheduler $\mathfrak{s}$ if it is related to itself. Which configurations must remain indistinguishable differs from Section 3.2 because information may only be declassified in a computation step if this is permitted by the set of local escape hatches that are enabled at this step. That is, declassification is more constrained than in Section 3.2.

Definition 15. *Let* $d \in \mathcal{D}$ *be a security domain,* $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ *be a set of local escape hatches, and* $PP \subseteq \mathcal{PP}$ *be a set of program points. An* $\mathfrak{s}$-specific strong (d, lH, PP)-bisimulation *is a per* $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *that fulfills the following three conditions:*

1. $\forall (thr, thr') \in R_{d,lH,PP} \,.\, \forall k \in \mathbb{N}_0.$
   $k < \#(thr) \implies (NDC_d(thr[k]) \vee IDC_d(thr[k], htchLoc(lH, pp(thr[k]))))$
2. $\forall (cnf, cnf') \in R^{\uparrow}_{d,lH,PP}. \forall Cls \in classes(R^{\uparrow}_{d,lH,PP}).$
   $(htchs^{\mathfrak{s}}(lH, cnf, Cls) \cup htchs^{\mathfrak{s}}(lH, cnf', Cls)) \subseteq htchLoc(lH, PP)$
   $\implies prob^{\mathfrak{s}}(cnf, Cls) = prob^{\mathfrak{s}}(cnf', Cls)$
3. $\lambda cnf \in \mathcal{Cnf}.\ (cnf \in \bigcup classes(R^{\uparrow}_{d,lH,PP}))$ *is an invariant under* $\mathfrak{s}$

Condition 1 in Definition 15 ensures that each thread $thr[k]$ either causes no declassification to the security domain d or is an immediate declassification command for the set of locally available escape hatches. Condition 2 ensures that

if a single computation step is performed in two related configurations cnf and cnf' then each equivalence class of $R^{\uparrow}_{d,lH,PP}$ is reached with the same probability from the two configurations. In contrast to Condition 1 in Definition 10, this is only required under the condition that each escape hatch (d', e) with $d' \leq d$, that is available at some program point ι that might cause the next computation step, is also contained in $htchLoc(lH, PP)$. Note that this precondition (i.e. $(htchs^{\mathfrak{s}}(lH, cnf, Cls) \cup htchs^{\mathfrak{s}}(lH, cnf', Cls)) \subseteq htchLoc(lH, PP)$) is trivially fulfilled if $PP = \mathcal{PP}$ holds. However, if PP is a proper subset of $\mathcal{PP}$ then the precondition might be violated. That is, choosing a set PP that is too small might lead to missing possibilities for information laundering. We will avoid this pitfall by universally quantifying over all subsets $PP \subseteq \mathcal{PP}$ in the definition of WHAT&WHERE$^{\mathfrak{s}}$. Finally, Condition 3 ensures that all configurations that can result after a computation step are again contained in some equivalence class of $R^{\uparrow}_{d,lH,PP}$. This lifts Condition 1 and 2 from individual steps to entire runs.

Definition 16. *A thread pool $thr \in \mathcal{C}^*$* has secure information flow *for $(\mathcal{D}, \leq, dom)$ and $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ under $\mathfrak{s}$ (brief: $thr \in$ WHAT&WHERE$^{\mathfrak{s}}$) iff for each $d \in \mathcal{D}$ and for each $PP \subseteq \mathcal{PP}$ there are a set $lH' \subseteq lH$ and a relation $R_{d,lH',PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ such that $(thr\ R_{d,lH',PP}\ thr)$ holds, and such that $R_{d,lH',PP}$ is an $\mathfrak{s}$-specific strong (d, lH', PP)-bisimulation.*

The structure of Definition 16 is similar to the one of Definition 11. The main differences are, firstly, that a set lH of local escape hatches is used instead of a set H of escape hatches and, secondly, that the escape hatches, that are available to a d-observer, are further constrained by a set $PP \subseteq \mathcal{PP}$. The universal quantification over all subsets PP of $\mathcal{PP}$ is crucial for achieving the desired control of where a declassification can occur. It were not enough to require Condition 2 in Definition 15 just for $PP = \mathcal{PP}$ because the resulting security guarantee would control what is declassified without restricting where declassification can occur.

Example 2. Let $P_2 = \mathsf{if}_1\ \mathsf{h}\ \mathsf{then}\ \mathsf{spawn}_2(\mathsf{l}:=_3 0, \mathsf{l}:=_4 1)\ \mathsf{else}\ \mathsf{spawn}_5(\mathsf{l}:=_6 1, \mathsf{l}:=_7 0)\ \mathsf{fi}$ and $lH = \emptyset$. We consider a biased scheduler $\mathfrak{s}$ that selects the second of two threads with lower, but non-zero probability. Independent of the value of h, P_2 might terminate with a memory state in which $\mathsf{l} = 0$ holds as well as with a memory state in which $\mathsf{l} = 1$ holds. Nevertheless, a good guess about the initial value of h is possible after observing several runs with the same initial memory. If $\mathsf{l} = 0$ is observed significantly more often than $\mathsf{l} = 1$, then it is likely that $\mathsf{h} = \mathsf{False}$ holds in the initial state. Hence, the program is intuitively insecure.

Running P_2 with two memories that differ in h deterministically results in two different thread pools, namely in $\langle \mathsf{l}:=_3 0, \mathsf{l}:=_4 1\rangle$ and $\langle \mathsf{l}:=_6 1, \mathsf{l}:=_7 0\rangle$. These two thread pools must be related by $R_{low,lH,\mathcal{PP}}$ according to Condition 2 in Definition 15. However, the probability of moving from these two configurations into the same equivalence class differs as our biased scheduler chooses the first thread with a higher probability than the second. Therefore, Condition 2 is violated by the second computation step and, hence, $P_2 \notin$ WHAT&WHERE$^{\mathfrak{s}}$. $\Diamond$

Example 3. Let $P_3 = \mathsf{h2}:=_1 \mathsf{absolute}(\mathsf{h2}); \mathsf{if}_2\ \mathsf{h1}\ \mathsf{then}\ \mathsf{l1}:=_3 \mathsf{h2}\ \mathsf{else}\ \mathsf{l1}:=_4 \text{-}\mathsf{h2}\ \mathsf{fi}$ and $lH = \{(low, \mathsf{h2}, 3), (low, \mathsf{h2}, 4)\}$. The assignments in both branches do not reveal more

information than permitted by the respective local escape hatches. However, the sign of the value stored in l1 after a run reveals information about the initial value of h1 in addition. Hence, the program is intuitively insecure.

Two consecutive computation steps of P_3 in two memories that differ in h1 result in two different thread pools, namely in $\langle \mathsf{l1}{:=_3}\mathsf{h2} \rangle$ and $\langle \mathsf{l1}{:=_4}\mathsf{h2} \rangle$. According to Condition 2 in Definition 15, these two thread pools must be related by $R_{low,lH,\mathcal{PP}}$. However, a third computation step in each of them results in two memories that are *low*-distinguishable and, hence, $P_3 \notin \text{WHAT\&WHERE}^{\mathsf{s}}$. $\Diamond$

3.4 Meta-properties of the Scheduler-Specific Security Properties

The security conditions WHAT&WHERE$^{\mathsf{s}}$ restrict declassification according to a set of local escape hatches. This allows one a more fine-grained control of declassification by restricting what information can be declassified where. In comparison to WHAT$^{\mathsf{s}}$, declassification shall be controlled more rigorously, and WHAT&WHERE$^{\mathsf{s}}$ is indeed at least as restrictive as WHAT$^{\mathsf{s}}$.

Theorem 3. *Let* $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ *and* $thr \in \mathcal{C}^*$. *If* $thr \in \text{WHAT\&WHERE}^{\mathsf{s}}$ *with* lH *then* $thr \in \text{WHAT}^{\mathsf{s}}$ *with* $H = htchLoc(lH, \mathcal{PP})$.

In [32], various so called prudent principles were proposed as sanity checks for definitions of information-flow security that are compatible with declassification. In order to convince ourselves about the adequacy of our novel security condition, we have checked WHAT&WHERE$^{\mathsf{s}}$ against these principles, and we have shown that it satisfies the following prudent principles (based on the formalization of a slightly augmented set of prudent principles in [16]):

Semantic consistency [32] The (in)security of a program is invariant under semantic-preserving transformations of declassification-free subprograms.

Monotonicity of release [32] Allowing further declassifications for a program that is WHAT&WHERE$^{\mathsf{s}}$-secure cannot render it insecure.

Persistence [16] For every program that satisfies WHAT&WHERE$^{\mathsf{s}}$, all programs that are reachable also satisfy this security condition.

Relaxation [16] Every program that satisfies noninterference also satisfies WHAT&WHERE$^{\mathsf{s}}$.

Noninterference up-to [16] Every WHAT&WHERE$^{\mathsf{s}}$-secure program also satisfies noninterference if it were executed in an environment that terminates the program when it is about to perform a declassification.

Another prudent principle proposed in [32] is Non-occlusion. This principle requires that the presence of a declassifying operation cannot mask other covert information leaks. Unfortunately, a bootstrapping problem occurs. Any adequate formal characterization of non-occlusion itself is an adequate definition of information-flow security with controlled declassification. If such an adequate characterization existed then there would be no need to propose a definition of information-flow security.

4 Secure Declassification for Multi-threaded Programs

When developing a multi-threaded program, usually a specification of the scheduler's interface is available, but the concrete scheduler is not known. An interface might reveal to a scheduler information about the current configuration such as the number of active threads and the values of special program variables (e.g., for setting scheduling priorities). However, the scheduler should not have direct access to secrets via the interface because the scheduling of threads might have an effect on the probability of an attacker's observations. Hence, one should treat all elements of the scheduler's interface like public sinks in a security analysis.

We specify interfaces to schedulers by observation functions (see Section 2.1) and assume that interfaces do not give a scheduler access to the value of program counters as well as of variables that might contain secrets. This is captured by the following restriction on observation functions.

Definition 17. *An observation function* $obs \in \mathcal{Obs}$ *is* confined *wrt. an mls-policy* $(\mathcal{D}, \leq, dom)$*, iff for all* thr_1*,* $thr_1' \in \mathcal{C}^*$ *and all* $m_1, m_1' \in \mathcal{Mem}$*:*

$$(\sharp(thr_1) = \sharp(thr_1') \wedge \exists d \in \mathcal{D}.\ m_1 =_d m_1') \implies obs(thr_1, m_1) = obs(thr_1', m_1') \ .$$

If the interface to the scheduler is confined, then the scheduling behavior is identical for any two configurations that have the same number of active threads and assign the same value to each variable that is visible for all security domains.

Remark 2. Note that our restriction to confined observation functions does not eliminate the refinement problem for schedulers. As already pointed out in [35], a program might have secure information flow if executed with the fictitious possibilistic scheduler, but be insecure if executed with a uniform scheduler. Since a uniform scheduler bases its decisions only on the number of active threads, its interface can be captured by a confined observation function. Another example of a scheduler with a confined observation function is the biased scheduler described in Example 2. The program P_2 in this example is insecure if run with the biased scheduler, but it would be secure if run with the possibilistic scheduler. ◊

As the concrete scheduler is usually not known when developing a program, properties are needed that allow one to reason about security independently of the concrete scheduler. In this section, we recall the security property WHAT_1 from [20] and propose the novel security property WHAT&WHERE. We show that these properties imply $\text{WHAT}^{\mathfrak{s}}$ and $\text{WHAT\&WHERE}^{\mathfrak{s}}$, respectively, for all schedulers $\mathfrak{s}$ and confined observation functions. These scheduler-independence results provide the theoretical basis for reasoning in a sound way about the security of multi-threaded programs without knowing the concrete scheduler.

4.1 Scheduler-independent WHAT-Security

The following definition of strong (d, H)-bisimulations is an adaptation of the corresponding notion from [20] to the formal exposition used in this article.

$$\forall thr, thr' \in \mathcal{C}^*.\ \forall m_1, m_1' \in \mathcal{M}em.\ \forall k \in \mathbb{N}_0.\ \forall \alpha \in \mathcal{C}^*.\ \forall c \in \mathcal{C}_\epsilon.\ \forall m_2 \in \mathcal{M}em.$$
$$\Big[\ thr\ R_{d,H}\ thr' \wedge m_1 \sim_d^H m_1' \wedge \langle\!| thr[k], m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c, m_2 |\!\rangle$$
$$\Longrightarrow \exists \alpha' \in \mathcal{C}^*.\ \exists c' \in \mathcal{C}_\epsilon.\ \exists m_2' \in \mathcal{M}em.$$
$$\Big[\ \langle\!| thr'[k], m_1' |\!\rangle \xrightarrow{\alpha'} \langle\!| c', m_2' |\!\rangle \wedge \langle c \rangle\ R_{d,H}\ \langle c' \rangle \wedge \alpha\ R_{d,H}\ \alpha' \wedge m_2 \sim_d^H m_2'\ \Big]\ \Big]$$

Figure 1. Condition 2 in the definition of strong (d, H)-bisimulations

Definition 18. *Let $d \in \mathcal{D}$ be a security domain and $H \subseteq \mathcal{D} \times \mathcal{E}$ be a set of escape hatches. A* strong (d, H)-bisimulation *is a per $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ that fulfills the following two conditions:*

1. $\forall (thr, thr') \in R_{d,H}\ .\ \#(thr) = \#(thr')$ *and*
2. $R_{d,H}$ *satisfies the formula in Figure 1.*

If two thread pools $thr, thr' \in \mathcal{C}^*$ are strongly (d, H)-bisimilar, and the scheduler chooses in some memory state m the k'th thread of the first thread pool thr for a step, then the thread at position k in the second thread pool thr' can also perform a computation step in any memory state m' that is (d, H)-equal to m (see dark-gray boxes in Figure 1). Moreover, the program states as well as the lists of spawned threads resulting after these two steps are, again, strongly (d, H)-bisimilar (see medium-gray box in Figure 1). Finally, the resulting memory states are, again (d, H)-equal (see light-gray box in Figure 1).

Definition 19. *A thread pool thr* has secure information flow *for $(\mathcal{D}, \leq, dom)$ and $H \subseteq \mathcal{D} \times \mathcal{E}$ (brief: $thr \in WHAT_1$) iff for each $d \in \mathcal{D}$ there is a strong (d, H)-bisimulation $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ such that $(thr\ R_{d,H}\ thr)$ holds.*

We are now ready to present our scheduler-independence result for WHAT-security. The theorem states that WHAT_1 implies $\text{WHAT}^{\mathfrak{s}}$ for each scheduler model $\mathfrak{s}$. Hence, WHAT_1 is suitable for reasoning about WHAT-security in a sound manner without having to explicitly consider scheduling.

Theorem 4. *Let $(\mathcal{D}, \leq, dom)$ be an mls-policy, $H \subseteq \mathcal{D} \times \mathcal{E}$ be a set of escape hatches, $obs \in \mathcal{O}bs$ be an observation function that is confined wrt. $(\mathcal{D}, \leq, dom)$, and $thr \in \mathcal{C}^*$ be a thread pool. If $thr \in WHAT_1$ holds, then $thr \in \text{WHAT}^{\mathfrak{s}}$ holds for each scheduler model $\mathfrak{s}$.*

4.2 Scheduler-independent WHAT&WHERE-Security

Like in Section 3.3, we use pers that are annotated with a security domain d, a set lH of local escape hatches, and a set PP of program points. Unlike in Section 3.3, we constrain pers without referring to system steps, because system steps depend on the concrete scheduler's behavior. Our novel security property WHAT&WHERE shall provide adequate control over what information is declassified where, independently of the scheduler under that a program is run.

$$\forall thr, thr' \in \mathcal{C}^*.\ \forall m_1, m_1' \in \mathcal{Mem}.\ \forall k \in \mathbb{N}_0.\ \forall \alpha \in \mathcal{C}^*.\ \forall c \in \mathcal{C}_\epsilon.\ \forall m_2 \in \mathcal{Mem}.$$
$$\left[\begin{array}{l} thr\ R_{d,lH,PP}\ thr' \wedge m_1 \sim_d^{htchLoc(lH,PP)} m_1' \wedge \langle\!\langle thr[k], m_1 \rangle\!\rangle \xrightarrow{\alpha} \langle\!\langle c, m_2 \rangle\!\rangle \\ \Longrightarrow \exists \alpha' \in \mathcal{C}^*.\ \exists c \in \mathcal{C}_\epsilon.\ \exists m_2' \in \mathcal{Mem}. \\ \quad \left[\begin{array}{l} \langle\!\langle thr'[k], m_1' \rangle\!\rangle \xrightarrow{\alpha'} \langle\!\langle c', m_2' \rangle\!\rangle \wedge \langle c \rangle\ R_{d,lH,PP}\ \langle c' \rangle \wedge \alpha\ R_{d,lH,PP}\ \alpha' \\ \wedge \Big(m_2 \sim_d^{htchLoc(lH,PP)} m_2' \vee htchLoc(lH, pp(thr[k])) \not\subseteq htchLoc(lH, PP) \Big) \end{array}\right] \end{array}\right]$$

Figure 2. Condition 3 in the definition of strong (d, lH, PP)-bisimulations

Definition 20. *Let $d \in \mathcal{D}$ be a security domain, $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ be a set of local escape hatches, and $PP \subseteq \mathcal{PP}$ be a set of program points. A* strong (d, lH, PP)-bisimulation *is a per $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ that fulfills the following three conditions:*

1. $\forall (thr, thr') \in R_{d,lH,PP}\ .\ \#(thr) = \#(thr')$,
2. $\forall (thr, thr') \in R_{d,lH,PP}\ .\ \forall k \in \mathbb{N}_0.$
   $k < \#(thr) \Longrightarrow (NDC_d(thr[k]) \vee IDC_d(thr[k], htchLoc(lH, pp(thr[k]))))$,
3. *$R_{d,lH,PP}$ satisfies the formula in Figure 2.*

Condition 1 in Definition 20 ensures that related thread pools have equal size (like Condition 1 in Definition 18). Condition 2 ensures that each thread either causes no declassification to d or is an immediate declassification command for the set of locally available escape hatches (like Condition 1 in Definition 15).

Condition 3 bears similarities with Condition 2 in Definition 18 (see Figure 1). If two thread pools $thr, thr' \in \mathcal{C}^*$ are strongly (d, lH, PP)-bisimilar, and the scheduler chooses in some memory state m the k'th thread of thr for a step, then the k'th thread of thr' can also perform a computation step in any memory state m' that is (d, H)-equal to m (where $H = htchLoc(lH, PP)$), and the resulting program states as well as lists of spawned threads are, again, strongly (d, lH, PP)-bisimilar (see dark-gray boxes in Figure 2). Note that an expression e that occurs in a local escape hatch $(d', e, \iota) \in lH$ need not have the same value in m and m' if $\iota \notin PP$. Consequently, Condition 3 only requires the resulting memory states to be (d, H)-equal (see medium-gray box in Figure 2), if no such local escape hatch might affect the computation step under consideration (see light-gray box in Figure 2). Like in Section 3.3, choosing a set PP that is too small might lead to missing possibilities for information laundering and, again, we will avoid this pitfall by universally quantifying over all subsets $PP \subseteq \mathcal{PP}$.

Definition 21. *A thread pool $thr \in \mathcal{C}^*$* has secure information flow *for an mls-policy $(\mathcal{D}, \leq, dom)$ and a set of local escape hatches $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ (brief: $thr \in$* WHAT&WHERE*) iff for each $d \in \mathcal{D}$ and for each $PP \subseteq \mathcal{PP}$ there is a strong (d, lH, PP)-bisimulation $R_{d,lH,PP}$ such that $(thr\ R_{d,lH,PP}\ thr)$ holds.*

We are now ready to present our second scheduler-independence result.

$$[\text{tconstd}]\frac{}{H \vdash v : d} \qquad [\text{tvard}]\frac{dom(x) = d}{H \vdash x : d} \qquad [\text{thatchd}]\frac{(d, e) \in H}{H \vdash e : d}$$

$$[\text{topd}]\frac{H \vdash e_1 : d_1 \ \ldots \ H \vdash e_m : d_m \qquad \forall i \in \{1, \ldots, m\}.\ d_i \leq d}{H \vdash op(e_1, \ldots, e_m) : d}$$

Figure 3. Security type system for expressions

Theorem 5. *Let $(\mathcal{D}, \leq, dom)$ be an mls-policy, $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ be a set of local escape hatches, $obs \in \mathcal{Obs}$ be an observation function that is confined wrt. $(\mathcal{D}, \leq, dom)$, and $thr \in \mathcal{C}^*$ be a thread pool. If $thr \in$ WHAT&WHERE holds, then $thr \in$ WHAT&WHERE$^{\mathfrak{s}}$ holds for each scheduler model $\mathfrak{s}$.*

The scheduler-independence theorem shows that WHAT&WHERE provides as much control of what information is declassified where as WHAT&WHERE$^{\mathfrak{s}}$, but without referring to specific schedulers. Hence, WHAT&WHERE is adequate for reasoning about the security of programs when the scheduler is unknown.

5 Security Type System

Our security property WHAT&WHERE is compositional in the following sense:

Theorem 6. *Let $c_0, \ldots, c_{n-1} \in \mathcal{C}$ be commands and $e \in \mathcal{E}$ be an expression. If $\langle c_0 \rangle, \ldots, \langle c_{n-1} \rangle \in$ WHAT&WHERE and if $(m =_d m' \implies eval(e, m) = eval(e, m'))$ holds for all $m, m' \in \mathcal{Mem}$ and all $d \in \mathcal{D}$, then we have:*

1. $\langle c_0; c_1 \rangle \in$ WHAT&WHERE,
2. $\langle \mathsf{spawn}_\iota(c_0, \ldots, c_{n-1}) \rangle \in$ WHAT&WHERE,
3. $\langle \mathsf{while}_\iota\ e\ \mathsf{do}\ c_0\ \mathsf{od} \rangle \in$ WHAT&WHERE, *and*
4. $\langle \mathsf{if}_\iota\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \rangle \in$ WHAT&WHERE.

We will now define a syntactic approximation of WHAT&WHERE for our example language in Section 2.2 in the form of a type system. Before we present the typing rules for the commands, we present typing rules for expressions. The judgment $H \vdash e : d$ (where $H \subseteq \mathcal{D} \times \mathcal{E}$, $e \in \mathcal{E}$ and $d \in \mathcal{D}$) can be derived with the typing rules in Figure 3. Intuitively, the judgment $H \vdash e : d$ shall model that the value of e only depends on information that a d-observer is permitted to obtain (for a given mls-policy and the set H of escape hatches). That the typing rules capture this intuition is ensured by the following theorem:

Theorem 7. *Let $H \subseteq \mathcal{D} \times \mathcal{E}$, $e \in \mathcal{E}$, and $d \in \mathcal{D}$. If $H \vdash e : d$ is derivable then*

$$\forall m, m' \in \mathcal{Mem}.\ \left[m \sim_d^H m' \implies eval(e, m) = eval(e, m')\right] .$$

For verifying the security of programs we use judgments of the form $\vdash c$ (where $c \in \mathcal{C}$). Intuitively, $\vdash c$ shall express that c satisfies our novel security condition WHAT&WHERE from Section 4.2. The typing rules for this judgment are presented in Figure 4. The typing rules tseq, tspawn, twhile and tif correspond

$$[\text{tassign}]\frac{htchLoc(lH,\iota) \vdash e : d \quad d \leq dom(x) \quad SubstClosure(lH,x,e)}{\vdash x{:=}_\iota e}$$

$$[\text{tseq}]\frac{\vdash c_1 \quad \vdash c_2}{\vdash c_1\,;\,c_2} \qquad [\text{tif}]\frac{\emptyset \vdash e : d' \qquad \forall d''.\ d' \leq d'' \quad \vdash c_1 \quad \vdash c_2}{\vdash \mathsf{if}_\iota\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}} \qquad [\text{tskip}]\frac{}{\vdash \mathsf{skip}_\iota}$$

$$[\text{tspawn}]\frac{\vdash c_0\ \ldots\ \vdash c_{n-1}}{\vdash \mathsf{spawn}_\iota(c_0,\ldots,c_{n-1})} \qquad [\text{twhile}]\frac{\emptyset \vdash e : d' \quad \forall d''.\ d' \leq d'' \quad \vdash c}{\vdash \mathsf{while}_\iota\ e\ \mathsf{do}\ c\ \mathsf{od}}$$

Figure 4. Security type system for commands

to the four cases of the compositionality theorem (i.e., Theorem 6). Note that the first two preconditions of twhile and tif indeed ensure that $(m =_d m' \Longrightarrow eval(e,m) = eval(e,m'))$ holds for all $m, m' \in \mathcal{Mem}$ and all $d \in \mathcal{D}$. The first two preconditions of the rule for assignments (i.e., tassign) ensure that information only flows into a variable $x \in \mathcal{Var}$ if this is permissible according to the mls-policy and to the set of locally available escape hatches. The third precondition of rule tassign prevents information laundering like in the following example.

Example 4. Let $P_4 = \mathsf{h2}{:=_1}\mathsf{0};\mathsf{l}{:=_2}\mathsf{h1+h2}$ and $lH = \{(low, \mathsf{h1+h2}, \iota) \mid \iota \in \mathcal{PP}\}$. If the third precondition of rule tassign were not present, then P_4 would be accepted by the type system. However, the program reveals the value of $\mathsf{h1}$ to a *low*-observer, which is not permitted by lH under the two-level mls-policy. ◊

In order to avoid such possibilities for information laundering via escape hatches, we use the predicate *SubstClosure* in the third precondition of rule tassign:

Definition 22. *We define* $SubstClosure \subseteq \mathfrak{P}(\mathcal{D} \times \mathcal{E} \times \mathcal{PP}) \times \mathcal{Var} \times \mathcal{E}$ *by*

$$SubstClosure(lH,x,e) \Longleftrightarrow \forall (d',e',\iota') \in lH.\ (d', e'[x\backslash e], \iota') \in lH$$

where $e'[x\backslash e]$ *is the expression that results from substituting all occurrences of variable* x *in expression* e' *by the expression* e*.*

The third precondition of rule tassign (i.e., $SubstClosure(lH,x,e)$) requires that, if the target x of an assignment occurs in the expression e' of some $(d',e',\iota') \in lH$ then $(d', e'[x\backslash e], \iota') \in lH$ must also hold. This ensures that the local escape hatch $(d',e',\iota') \in lH$ may still be used legitimately, after assigning e to x.

The following soundness theorem shows that the judgment $\vdash c$ indeed captures WHAT&WHERE:

Theorem 8. *Let* $c \in \mathcal{C}$*. If* $\vdash c$ *is derivable then* $c \in$ WHAT&WHERE *holds.*

If a program is typable with our security type system, then it adequately controls what information is declassified where, no matter under which scheduler the program is run. This follows from the soundness theorem above in combination with our scheduler-independence result for WHAT&WHERE (i.e., Theorem 5).

Example 5. We reconsider the program P_1 from Example 1 and the set $lH = \{(low, \mathsf{downsample(song,\ bitrate)}, 3), (low, \mathsf{song}, 2)\}$. The judgment $\vdash P_1$ can be derived by applying the rules tif, tvard (for paid and $d = low$), tassign, thatchd

(for song), tassign, thatchd (for downsample(song, bitrate)). From Theorem 8 and Theorem 5 we obtain $P_1 \in \text{WHAT\&WHERE}^{\mathfrak{s}}$ regardless of the scheduler $\mathfrak{s}$. ◊

Remark 3. The type system presented in this section is suitable for verifying WHAT&WHERE-security in a sound way. In the definition of the typing rules, we aimed for conceptual simplicity rather than for maximizing the precision of the analysis. For instance, a more fine-grained treatment of conditionals could be developed by using safe approximation relations (like in [21]). ◊

6 Related Work

Research on information-flow security has addressed scheduler independence as well as declassification, but not yet the combination of these two aspects.

To achieve scheduler-independent information-flow security, three main directions have been explored. *Observational determinism* [36,13] requires that all observations of an attacker are deterministically determined by information that this attacker may obtain. This ensures that security is not affected by how non-determinism is resolved (including the selection of threads by a scheduler). An alternative approach to achieving scheduler independence requires a non-standard interface to schedulers. Schedulers can be asked to "hide" or "unhide" threads via this interface, where threads classified as "unhidden" may only be scheduled if no "hidden" threads are active [7,28]. *Strong security* [31] achieves scheduler independence by defining security based on stepwise bisimulation relations that match steps of threads at the same position, like in this article. *FSI-security* [22] is also a scheduler-independent security condition, although it is less restrictive than strong security. None of these approaches supports declassification.

Scheduler-independence results can be viewed as solutions to the refinement paradox [14] in a particular domain. In fact, the approach to define security based on observational determinism was originally developed as a general solution to avoid the refinement paradox [27]. Unfortunately, this approach also forbids intended non-determinism. An alternative is to identify notions of refinement that preserve information-flow security. For event-based specifications, such refinement operators are proposed in [18]. For sequential programs, refinements that preserve the property "ignorance of secrets" are characterized in [24].

The challenge of certifying information-flow security while permitting declassification is addressed in various publications (see [33] for an overview). In order to make differences in the goals of different approaches to controlling declassification explicit, three aspects of declassification were distinguished in [21]: *What* information may be declassified, *Where* information may be declassified, and *Who* may declassify information. Four dimensions of declassification, which are similar to these W-aspects, are used in [33] to classify existing approaches to declassification. Our novel security condition WHAT&WHERE for multi-threaded programs addresses the aspects *What* and *Where* in an integrated fashion.

For sequential programs, there are solutions addressing the aspects *What* (e.g., [29,15,16]), *Where* (e.g., [10,2,11]), and *Who* (e.g., [25,26,17]) in isolation.

There are also approaches that control *What* information is declassified *Where*. *Localized delimited release* [3] and the security conditions in [4] permit to specify from which program point on the value of a given expression may be declassified. *Delimited non-disclosure* [6] and *delimited gradual release* [5] permit to specify exactly at which position a given expression may be declassified. For the latter two, the value that may be declassified is the value to which the expression evaluates when the declassification is performed. In all other approaches (including the approach in this article), the value that may be declassified is the initial value of the expression. The relation between these two interpretations of escape hatches is clarified in [16]. All previously proposed approaches to control *What* is declassified *Where* were developed for sequential programs.

In a multi-threaded setting, several approaches adopt the ideas underlying *strong security* [31]. *Intransitive noninterference* [21] and WHERE [20] permit declassification by dedicated declassification commands that comply with a flow relation, which may be an intransitive relation. The properties WHAT_1 and WHAT_2 in [20] control that what is declassified complies with a given set of escape hatches. The conditions $SIMP^*_D$ [9] and *non-disclosure* [1] are also based on step-wise bisimulations. However, they do not require that matching steps are executed by threads at the same position, which seems necessary for achieving scheduler independence. While some of these approaches strive for scheduler independence, no scheduler-independence result has been published for them.

7 Conclusion

The scheduler-independence results presented in this article constitute the first two such results for definitions of information-flow security that are compatible with declassification. We showed that our previously proposed security condition WHAT_1 [20] provides adequate control of what can be declassified, for all schedulers that can be expressed in our scheduler model. When proposing WHAT_1, we had hoped that this condition is scheduler independent, but had no proof for this so far. Our novel security condition WHAT&WHERE provides adequate control of what can be declassified where, independent of the scheduler. Our two scheduler-independence results provide the theoretical basis for reasoning about the security of multi-threaded programs in a sound way, without having to explicitly consider the scheduler under which a program runs.

The security guarantees provided by WHAT&WHERE go far beyond a mere conjunction of the previously proposed conditions WHAT_1 and WHERE because a fine-grained, integrated control of what is declassified where is made possible.

The scheduler model (cf. Definition 1) that we used as basis in this article is sufficiently expressive to capture a wide range of schedulers, including uniform and Round-Robin schedulers. Moreover, to our knowledge, WHAT^{s} and $\text{WHAT\&WHERE}^{\text{s}}$ offer the first scheduler-specific definitions of information-flow security that are compatible with declassification. We used these schemas as reference points for our two scheduler-independence results, and they might serve as role models for other scheduler-specific security conditions in the future.

With this article, we hope to contribute foundations that lead to a better applicability and a more wide-spread use of information-flow analysis in practice.

Acknowledgments. We thank Carroll Morgan, Jeremy Gibbons and the anonymous reviewers for their helpful comments. This work was funded by the DFG under the project RSCP (MA 3326/4-1) in the priority program RS^3 (SPP 1496).

References

1. Almeida Matos, A., Boudol, G.: On Declassification and the Non-Disclosure Policy. Journal of Computer Security 17(5), 549–597 (2009)
2. Askarov, A., Sabelfeld, A.: Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In: IEEE Symposium on Security and Privacy. pp. 207–221 (2007)
3. Askarov, A., Sabelfeld, A.: Localized Delimited Release: Combining the What and Where Dimensions of Information Release. In: Workshop on Programming Languages and Analysis for Security. pp. 53–60 (2007)
4. Askarov, A., Sabelfeld, A.: Tight Enforcement of Information-Release Policies for Dynamic Languages. In: IEEE Computer Security Foundations Symposium. pp. 43–59 (2009)
5. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive Declassification Policies and Modular Static Enforcement. In: IEEE Symposium on Security and Privacy. pp. 339–353 (2008)
6. Barthe, G., Cavadini, S., Rezk, T.: Tractable Enforcement of Declassification Policies. In: IEEE Computer Security Foundations Symposium. pp. 83–97 (2008)
7. Barthe, G., Rezk, T., Russo, A., Sabelfeld, A.: Security of Multithreaded Programs by Compilation. In: ESORICS. pp. 2–18. LNCS 4734, Springer (2007)
8. Bell, D.E., LaPadula, L.: Secure Computer Systems: Unified Exposition and Multics Interpretation. Tech. Rep. MTR-2997, MITRE (1976)
9. Bossi, A., Piazza, C., Rossi, S.: Compositional Information Flow Security for Concurrent Programs. Journal of Computer Security 15(3), 373–416 (2007)
10. Broberg, N., Sands, D.: Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In: ESOP. pp. 180–196. LNCS 3924, Springer (2006)
11. Broberg, N., Sands, D.: Paralocks: Role-based Information Flow Control and Beyond. In: ACM Symposium on Principles of Programming Languages. pp. 431–444 (2010)
12. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: IEEE Symposium on Security and Privacy. pp. 11–20 (1982)
13. Huisman, M., Worah, P., Sunesen, K.: A Temporal Logic Characterisation of Observational Determinism. In: IEEE Computer Security Foundations Workshop. pp. 3–15 (2006)
14. Jacob, J.: On the Derivation of Secure Components. In: IEEE Symposium on Security and Privacy. pp. 242–247 (1989)
15. Li, P., Zdancewic, S.: Downgrading Policies and Relaxed Noninterference. In: ACM Symposium on Principles of Programming Languages. pp. 158–170 (2005)
16. Lux, A., Mantel, H.: Declassification with Explicit Reference Points. In: ESORICS. pp. 69–85. LNCS 5789, Springer (2009)
17. Lux, A., Mantel, H.: Who Can Declassify? In: FAST 2008. pp. 35–49. LNCS 5491, Springer (2009)

18. Mantel, H.: Preserving Information Flow Properties under Refinement. In: IEEE Symposium on Security and Privacy. pp. 78–91 (2001)
19. Mantel, H.: Information Flow and Noninterference. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security (2nd Ed.), pp. 605–607. Springer (2011)
20. Mantel, H., Reinhard, A.: Controlling the What and Where of Declassification in Language-Based Security. In: ESOP. pp. 141–156. LNCS 4421, Springer (2007)
21. Mantel, H., Sands, D.: Controlled Declassification based on Intransitive Noninterference. In: APLAS. pp. 129–145. LNCS 3302, Springer (2004)
22. Mantel, H., Sudbrock, H.: Flexible Scheduler-Independent Security. In: ESORICS. pp. 116–133. LNCS 6345, Springer (2010)
23. McCullough, D.: Specifications for Multi-Level Security and a Hook-Up Property. In: IEEE Symposium on Security and Privacy. pp. 161–166 (1987)
24. Morgan, C.: *The Shadow Knows:* Refinement of Ignorance in Sequential Programs. In: MPC. pp. 359–378. LNCS 4014, Springer (2006)
25. Myers, A.C., Liskov, B.: Protecting Privacy using the Decentralized Label Model. ACM Transactions on Software Engineering and Methodology 9(4), 410–442 (2000)
26. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing Robust Declassification and Qualified Robustness. Journal of Computer Security 14, 157–196 (2006)
27. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through Determinism. In: ESORICS. pp. 33–53. LNCS 875, Springer (1994)
28. 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)
29. Sabelfeld, A., Myers, A.C.: A Model for Delimited Information Release. In: ISSS 2003. pp. 174–191. LNCS 3233, Springer (2004)
30. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. In: ESOP. pp. 50–59. LNCS 1576, Springer (1999)
31. Sabelfeld, A., Sands, D.: Probabilistic Noninterference for Multi-threaded Programs. In: IEEE Computer Security Foundations Workshop. pp. 200–215 (2000)
32. Sabelfeld, A., Sands, D.: Dimensions and Principles of Declassification. In: IEEE Computer Security Foundations Workshop. pp. 255–269 (2005)
33. Sabelfeld, A., Sands, D.: Declassification: Dimensions and Principles. Journal of Computer Security 17(5), 517–548 (2009)
34. Sutherland, D.: A Model of Information. In: National Computer Security Conference (1986)
35. Volpano, D., Smith, G.: Probabilistic Noninterference in a Concurrent Language. In: IEEE Computer Security Foundations Workshop. pp. 34–43 (1998)
36. Zdancewic, S., Myers, A.C.: Observational Determinism for Concurrent Program Security. In: IEEE Computer Security Foundations Workshop. pp. 29–43 (2003)