Controlling the What and Where of Declassification in Language-Based Security

Heiko Mantel and Alexander Reinhard

Security Engineering Group, RWTH Aachen University, Germany
mantel@cs.rwth-aachen.de, reinhard@i4.informatik.rwth-aachen.de

Abstract. While a rigorous information flow analysis is a key step in obtaining meaningful end-to-end confidentiality guarantees, one must also permit possibilities for declassification. Sabelfeld and Sands categorized the existing approaches to controlling declassification in their overview along four dimensions and according to four prudent principles [16]. In this article, we propose three novel security conditions for controlling the dimensions *where* and *what*, and we explain why these conditions constitute improvements over prior approaches. Moreover, we present a type-based security analysis and, as another novelty, prove a soundness result that considers more than one dimension of declassification.

1 Introduction

Research on information flow security aims at finding better ways to characterizing and analyzing security requirements concerning aspects of confidentiality and integrity. Regarding confidentiality, the aim of an information flow analysis is to answer: "Can a given program be trusted to operate in an environment where it has read access to secret data and write access to untrusted information sinks?" There is a variety of approaches to information flow security on the level of concrete programs (see [12] for an overview). In the simplest case, one has a two-level policy demanding that information cannot flow from *high* to *low*. Secure information flow can then be characterized using the idea underlying *noninterference* [6]: If *low* outputs of the program do not depend on *high* inputs then there is no danger that secret data is leaked to untrusted sinks.

Noninterference provides an intuitively convincing, declarative characterization of information flow security. However, there are security mechanisms and application scenarios that need some information to flow from *high* to *low*. For instance, a password-based authentication mechanism necessarily reveals some information about the secret password, decryption relies on a dependence between a cipher-text and the secret plain-text that it encodes, and electronic commerce requires secret data to be released after it has been paid for. For making information flow security compatible with such requirements, one must permit exceptions in the security policy. But, this raises the question how to control that one does not introduce possibilities for unintended information leakage.

Appeared in R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 141–156, 2007

For clarifying the intentions underlying the various approaches to controlling information release, three dimensions were introduced in [9]: *what* information is declassified, *who* can control whether declassification occurs, and *where* can declassification happen. In [16], Sabelfeld and Sands develop a taxonomy that categorizes the existing approaches along these dimensions[1] and propose four prudent principles of controlling declassification. The taxonomy clarified the relationship between the various approaches, and it revealed some anomalies and misconceptions that had previously gone unnoticed. Another interesting outcome is that each approach mainly aims at a single dimension and does not provide adequate control for any of the respective other dimensions.

In this article, our scope is controlling the *what* and *where* of declassification in a type-based security analysis. In summary, our research contributions are:

- A novel security characterization for controlling *where* declassification occurs. Our property WHERE is similar to *intransitive noninterference* [9], but WHERE satisfies the prudent principles of declassification from [16], including monotonicity, which is not satisfied by intransitive noninterference.
- Two novel security characterizations for controlling *what* is declassified. Our properties WHAT_1 and WHAT_2 are similar to selective dependency [3] and its descendants (e.g., [13]), but, unlike these properties, WHAT_1 and WHAT_2 are applicable to concurrent programs. Lifting a security characterization from a sequential to a concurrent setting is often not straightforward, in particular, one must address the danger of internal timing leaks [15].
- A security type system for analyzing the information flow in concurrent programs under policies that permit controlled exceptions. Our type system localizes *where* declassification occurs and controls *what* is declassified. We prove soundness results with respect to each of our properties WHERE, WHAT_1, and WHAT_2. To our knowledge, the only other formal soundness result for an information flow type system that considers *where* and *what* is the one by Li and Zdancewic [7]. However, they aim at sequential programs and mainly at controlling the *what* dimension [16].

In our project, we gained some further insights on controlling declassification. For instance, our property WHAT_1 is compositional but does not satisfy the monotonicity principle, while our property WHAT_2 is not compositional but satisfies monotonicity. We found that, when controlling the *what* dimension of declassification, one faces a fundamental difficulty when attempting to satisfy compositionality as well as monotonicity (see Sect. 3.2). While using the prudent principles of declassification as a sanity check for our security characterizations, we found that formalizing the informal descriptions of the principles from [16] is not always completely straightforward, and in some cases more than one formalization is sensible. As an example, we provide two alternative formalizations of the conservativity principle for WHERE (see Theorem 2).

[1] The taxonomy distinguishes localization of declassification with respect to aspects of time during program execution (*when*) from other aspects of localization (*where*) and categorizes according to the four dimensions: *what*, *who*, *where*, and *when*.

2 Controlling Declassification in Dimension *where*

We propose a novel characterization of information flow security that controls *where* declassification can occur. It is ensured that declassification is localized to specific parts of the security policy as well as to specific parts of the computation.

Definition 1. *A* multi-level security policy *(brief:* MLS policy*) is a pair* $(\mathcal{D}, \leq)$, *where* $\mathcal{D}$ *is a set of security domains and* $\leq \subseteq \mathcal{D} \times \mathcal{D}$ *is a partial order. The triple* $(\mathcal{D}, \leq, \rightsquigarrow)$ *is an* MLS policy with exceptions *where* $\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$. *The minimal and the maximal domain in* $(\mathcal{D}, \leq)$ *are called low and high, respectively, if they exist.*

Computation steps are modeled by labeled transitions between configurations of the form $\langle\!\langle C_1 \dots C_n \rangle, s \rangle\!\rangle$. Here, the state s is a mapping from program variables to values, and the vector models a pool of n threads that concurrently execute the commands $C_1, \dots, C_n \in Com$, respectively. For simplicity, we do not distinguish between commands and command vectors of length one in the notation and use the term *program* for referring to commands as well as to command vectors.

We distinguish ordinary computation steps, which are modeled by a transition relation $\twoheadrightarrow_o$, from declassification steps, which are modeled by a family of relations $(\twoheadrightarrow_d^{\mathcal{D}_1 \to D_2})_{\mathcal{D}_1, \{D_2\} \subseteq \mathcal{D}}$. Given a policy $(\mathcal{D}, \leq, \rightsquigarrow)$, the intuition is that an ordinary transition must strictly obey the ordering $\leq$ (which means that information may only flow upwards according to $\leq$), while declassification steps may violate this ordering by downgrading information from the domains in $\mathcal{D}_1$ to the domain D_2. However, such violations must comply with the relation $\rightsquigarrow$.

2.1 Preliminaries

Given a set Var of program variables, a *domain assignment* is a function $dom : Var \to \mathcal{D}$. By assigning a security domain $dom(Id)$ to each variable, it creates a connection between the configurations in a computation and the security policy. Taking the perspective of an observer in a security domain D, two states s, t are indistinguishable if all variables at or below this domain have the same value.

Definition 2. *For a given domain* $D \in \mathcal{D}$, *two states* s *and* t *are* D-equal *(denoted by* $s =_D t$*) if* $\forall Id \in Var : dom(Id) \leq D \implies s(Id) = t(Id)$.

In the following, let $(\mathcal{D}, \leq, \rightsquigarrow)$ be a policy and dom be a domain assignment. We adopt the naming conventions used above: D denotes a security domain, s and t denote states, C denotes a command, and V and W denote command vectors.

The PER approach [14] characterizes information flow security based on indistinguishability relations on programs. Two programs are indistinguishable for a security domain D if running them in two D-equal states reveals no secrets to an observer in D, unless this is explicitly permitted by the given security policy. The D-indistinguishability relation is not reflexive. It only relates programs to themselves if they have secure information flow.

Definition 3 ([15]). *A* strong D-bisimulation *is a symmetric relation* R *on command vectors of equal size that satisfies the formula in Fig. 1 where the part with dark-gray background is deleted. The relation* $\cong_D$ *is the union of all strong* D*-bisimulations. A program* V *is* strongly secure *if* $V \cong_D V$ *holds for all* $D \in \mathcal{D}$.

$$
\begin{array}{l}
\forall s, s', t : \forall i \in \{1 \ldots n\} : \forall W : \\
(V \; R \; V' \wedge \langle C_i, s \rangle \rightharpoonup \langle W, t \rangle \wedge s =_D s') \\
\Rightarrow \exists W', t' : W \; R \; W' \wedge \langle C'_i, s' \rangle \rightharpoonup \langle W', t' \rangle \\
\qquad \wedge \left[t =_D t' \vee \left[\begin{array}{l} \exists \mathcal{D}_1, \{D_2\} \subseteq \mathcal{D} : \\ \left[\begin{array}{l} \quad \langle C_i, s \rangle \rightharpoonup_d^{\mathcal{D}_1 \to D_2} \langle W, t \rangle \\ \wedge \forall D' \in \mathcal{D}_1 : (D' \rightsquigarrow D_2 \vee D' \leq D_2) \end{array} \right] \\ \wedge D_2 \leq D \wedge \exists D' \in \mathcal{D}_1 : s \neq_{D'} s' \end{array} \right] \right]
\end{array}
$$

Fig. 1. Characterization of Strong $(D, \rightsquigarrow)$-Bisimulation Relations (see Definition 4) where $V = \langle C_1, \ldots, C_n \rangle$, $V' = \langle C'_1, \ldots, C'_n \rangle$, and $\rightharpoonup \; = \; \rightharpoonup_o \cup (\bigcup_{\mathcal{D}_1, \{D_2\} \subseteq \mathcal{D}} \rightharpoonup_d^{\mathcal{D}_1 \to D_2})$

For two commands $C, C' \in Com$, being strongly D-bisimilar ($C \approxeq_D C'$) means that each computation step that is possible for C in a state s can be simulated in each D-equal state s' by a computation step of C', where the resulting programs W and W' are strongly D-bisimilar and the resulting states t and t' are D-equal. As a consequence, strong security enforces the flow of information to comply with the ordering $\leq$ without permitting any exceptions. The strong security condition is the weakest security definition that is scheduler independent and is preserved under parallel and sequential composition [11]. Technically, the former is a consequence of requiring strongly D-bisimilar programs to execute in lock-step.

2.2 A Novel Characterization of Flow Security

In this article, we propose several characterizations of information flow security that permit declassification while controlling it in a particular dimension. Our security conditions are derived using the PER approach, and each of them is presented as a variant of the strong security condition. We use the terms *what-security* and *where-security* to indicate in which dimension declassification is controlled and distinguish different variants for the same dimension with indices.

Definition 4 (WHERE). *A* strong $(D, \rightsquigarrow)$-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the entire formula in Fig. 1. The relation $\approxeq_D^{\rightsquigarrow}$ is the union of all strong $(D, \rightsquigarrow)$-bisimulations. A program V has secure information flow while complying with the restrictions where declassification can occur if $V \approxeq_D^{\rightsquigarrow} V$ holds for all $D \in \mathcal{D}$ (brief: V is* where-secure *or $V \in$ WHERE).*

Declassification is possible as t and t' in Fig. 1 need not be D-equal. However, such exceptions are constrained by the formula with dark-gray background:

- steps causing declassification must be declassification transitions $\rightharpoonup_d^{\mathcal{D}_1 \to D_2}$;
- information flow must be permitted from each $D' \in \mathcal{D}_1$ to D_2 (by $\rightsquigarrow$ or $\leq$);
- declassification may only affect D if D_2 is observable, and it may only reveal differences between s and s' that can be observed from domains in $\mathcal{D}_1$.

That is, *where*-security localizes exceptions, within a computation, to the declassification steps and, within an MLS policy, to where $\rightsquigarrow$ permits it. In this respect, our condition is similar to intransitive noninterference [9], but the two security conditions are not identical. Most importantly, *where*-security satisfies all prudent principles of declassification (see Sect. 2.3), unlike intransitive noninterference [16]. Technically, the differences become apparent in the definition of the respectively underlying notion of a strong D-bisimulation. In [9], firstly, declassification steps downgrade information from a single domain D_1 (rather than from a set of domains $\mathcal{D}_1$), secondly, declassification steps may only make information flow according to the relation $\rightsquigarrow$ (rather than according to $\rightsquigarrow \cup \leq$), and thirdly, each transition must be simulated by a transition with the identical annotation (while Fig. 1 requires nothing about the labels of the transition $\langle\!| C_i', s' |\!\rangle \rightarrowtriangle \langle\!| W', t' |\!\rangle$). The first two relaxations are helpful for a flexible combination with a control of *what* is downgraded. The third relaxation is crucial for satisfying the principle *monotonicity of release* (see Sect. 2.3).

2.3 Prudent Principles and Compositionality

To investigate our security definition more concretely, we augment the multi-threaded while language MWL from [15] with a declassifying assignment:

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid \mathit{Id}{:=}\mathit{Exp} \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \mid \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od} \\ & \mid \mathsf{fork}(C V) \mid [\mathit{Id}{:=}\mathit{Exp}] \end{aligned}$$

We use B and Exp for denoting Boolean-valued and integer-valued expressions, respectively. The language $\mathcal{E}$ for expressions shall not be specified here. We only assume that the evaluation of expressions is atomic and deterministic. That expression Exp evaluates to value n in state s is denoted by $\langle\!| \mathit{Exp}, s |\!\rangle \downarrow n$. We assume a function *sources* that returns for an expression the set of security domains on which the value of the expression possibly depends or, more formally, $\forall s, t : (((\forall D \in \mathit{sources}(\mathit{Exp}) : s =_D t) \wedge \langle\!| \mathit{Exp}, s |\!\rangle \downarrow n \wedge \langle\!| \mathit{Exp}, t |\!\rangle \downarrow m) \implies n = m)$.

The semantics of MWL instantiate the transition relations $\rightarrowtriangle_o$ and $\rightarrowtriangle_d^{\mathcal{D}_1 \to D_2}$. A command $[\mathit{Id}{:=}\mathit{Exp}]$ causes a $\rightarrowtriangle_d^{\mathcal{D}_1 \to D_2}$ transition where $\mathcal{D}_1 = \mathit{sources}(\mathit{Exp})$ and $D_2 = \mathit{dom}(\mathit{Id})$. Assignments, skip, conditionals, loops, and fork cause ordinary transitions. The statement $\mathsf{fork}(C V)$ spawns the threads $\langle C \rangle V$ where C is the designated *main thread*. If threads are created within the sub-command C_1 of a sequential composition $C_1; C_2$ then C_2 is executed after the main thread has terminated. A formal definition of the semantics is provided in Appendix A.

Sabelfeld and Sands propose the following principles of declassification [16]:

Semantic consistency: The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms.

Conservativity: The security of a program with no declassifications is equivalent to noninterference.

Monotonicity of release: Adding further declassifications to a secure program cannot render it insecure.

Non-occlusion: The presence of a declassification operation cannot mask other covert information leaks.

We now validate our security characterization against these prudent principles.

As suggested in [16], we define semantic equivalence between programs by $\approxeq = \approxeq_{high}$, where $\approxeq_{high}$ is the strong *high*-bisimulation for the single-domain policy $(\{high\}, \{(high, high)\})$. A *context* $\mathcal{C}$ is a program where the hole $\bullet$ may occur as an atomic sub-command. We use $\mathcal{C}[C]$ to denote the program that one obtains by replacing each occurrence of $\bullet$ with C. The proof of the following and all other theorems in this article will be provided in an extended version.

Theorem 1 (Semantic consistency). *Let C, C' be programs without declassification commands. Then $C' \approxeq C$ and $\mathcal{C}[C] \in WHERE$ imply $\mathcal{C}[C'] \in WHERE$.*[2]

Strong security follows from *where*-security not only if there are no declassification operations in a program, but also if the policy does not permit any exceptions. In the other direction, *where*-security is a weakening of strong security.

Theorem 2 (Conservativity).

1. *If $\rightsquigarrow = \emptyset$ and $V \in WHERE$ then V is strongly secure.*
2. *If no declassification occurs in V and $V \in WHERE$ then V is strongly secure.*
3. *If V is strongly secure then $V \in WHERE$.*

Monotonicity holds with respect to the exceptions permitted by the policy and also with respect to the declassification operations in the program.

Theorem 3 (Monotonicity). *Let $\rightsquigarrow \subseteq \rightsquigarrow'$.*

1. *If $V \in WHERE$ for $(\mathcal{D}, \leq, \rightsquigarrow)$ then $V \in WHERE$ for $(\mathcal{D}, \leq, \rightsquigarrow')$.*
2. *If $\mathcal{C}[Id{:=}Exp] \in WHERE$ then $\mathcal{C}[\,[Id{:=}Exp]\,] \in WHERE$.*

Theorems 1–3 demonstrate that our novel security characterization satisfies the first three principles of declassification from [16]. A formal proof of the fourth prudent principle is impossible. Such a proof would require a formal characterization of secure information flow as a reference point, which we do not have a priori as Definition 4 *defines* a characterization based on an intuitive understanding.

The following compositionality results hold for WHERE. We define expressions Exp, Exp' to be *D-indistinguishable* (denoted by $Exp \equiv_D Exp'$) if $\forall s, t : ((s =_D t \wedge \langle\!| Exp, s |\!\rangle \downarrow n \wedge \langle\!| Exp', t |\!\rangle \downarrow m) \Rightarrow n = m)$.

Theorem 4. *If $C_1 \approxeq_D^{\rightsquigarrow} C_1'$, $C_2 \approxeq_D^{\rightsquigarrow} C_2'$ and $V \approxeq_D^{\rightsquigarrow} V'$ then*

1. $C_1;\ C_2\ \approxeq_D^{\rightsquigarrow}\ C_1';\ C_2'$;
2. $\mathsf{fork}(C_1 V)\ \approxeq_D^{\rightsquigarrow}\ \mathsf{fork}(C_1' V')$;
3. $B \equiv_D B' \Rightarrow (\mathsf{while}\ B\ \mathsf{do}\ C_1\ \mathsf{od}\ \approxeq_D^{\rightsquigarrow}\ \mathsf{while}\ B'\ \mathsf{do}\ C_1'\ \mathsf{od})$;
4. $(B \equiv_D B' \vee C_1 \approxeq_D^{\rightsquigarrow} C_2) \Rightarrow (\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}\ \approxeq_D^{\rightsquigarrow}\ \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'\ \mathsf{fi})$.

3 Controlling Declassification in the Dimension *what*

We propose two characterizations of information flow security that control what is declassified. Each of them is a natural adaptation of the idea underlying Cohen's *selective dependency* [3] (and its descendants like, e.g., *delimited release* [13] or *abstract noninterference* [5]) to a multi-threaded language.

[2] As usual, the proposition does not hold if one replaces sub-commands with declassification commands. For instance, consider $\mathcal{C} = \bullet$, $C = [l{:=}h]$, and $C' = l{:=}h$ for the two-domain policy where $dom(h) = high$, $dom(l) = low$, and $high \rightsquigarrow low$.

Definition 5. *An* MLS policy with escape hatches *is a triple* $(\mathcal{D}, \leq, \mathcal{H})$*, where* $(\mathcal{D}, \leq)$ *is an MLS policy, and* $\mathcal{H} \subseteq \mathcal{D} \times \mathcal{E}$ *is a set of escape hatches.*

From now, we assume that $(\mathcal{D}, \leq, \mathcal{H})$ denotes an MLS policy with escape hatches. Given a policy $(\mathcal{D}, \leq, \mathcal{H})$ the intuition is that, for any D, the visible behavior of secure programs may depend on the initial value of identifiers visible to D and also on the initial values of expressions Exp if $(D', \mathit{Exp}) \in \mathcal{H}$ and $D' \leq D$. Formally, an observer in a domain D may be able to determine which equivalence class of the relation $=_D^{\mathcal{H}}$ contains the initial state, but no further information.

Definition 6. *Two states s and t are* $(D, \mathcal{H})$-equal *($s =_D^{\mathcal{H}} t$) if*
1. *$s =_D t$ and*
2. *$\forall (D', \mathit{Exp}) \in \mathcal{H} : (D' \leq D \implies ((\langle\!|\mathit{Exp}, s|\!\rangle \downarrow n \wedge \langle\!|\mathit{Exp}, t|\!\rangle \downarrow m) \Rightarrow n = m))$*

That is, an escape hatch $(D', \mathit{Exp}) \in \mathcal{H}$ indicates that observers in domain $D \geq D'$ may learn the initial value of expression Exp during a program's execution. The following lemma shows that $(D, \mathcal{H})$-equality is a subset of D-equality.

Lemma 1. $\forall D : \forall s, t : [(\forall \mathcal{H} : (s =_D^{\mathcal{H}} t \implies s =_D t)) \wedge (s =_D t \implies s =_D^{\emptyset} t)]$

3.1 Two Novel Characterizations of Flow Security

Our conditions WHAT$_1$ and WHAT$_2$ constitute adaptations of strong security (Definition 3) that permit declassification while controlling *what* is declassified.

Definition 7 (WHAT$_1$). *A* strong $(D, \mathcal{H})$-bisimulation *is a symmetric relation R on command vectors of equal size that satisfies the formula in Fig. 2. The relation $\approxeq_D^{\mathcal{H}}$ is the union of all strong $(D, \mathcal{H})$-bisimulations. A program V has secure information flow while complying with the restrictions what can be declassified if $\forall D : V \approxeq_D^{\mathcal{H}} V$ (brief: V is* what$_1$-secure *or $V \in$ WHAT$_1$).*

The difference between Definition 7 and the definition of strong D-bisimulations (see Definition 3) is that $=_D^{\mathcal{H}}$ occurs instead of $=_D$ on both sides of the implication. In the premise, $s =_D^{\mathcal{H}} s'$ occurs instead of $s =_D s'$. This modification leads to a *relaxation* of the security condition (see Lemma 1): differences in the values of an expression Exp that occurs in an escape hatch (D', Exp) may be revealed to an observer in domain D if $D' \leq D$. In the consequence, using $t =_D^{\mathcal{H}} t'$ instead of $t =_D t'$ leads to a *strengthening* of the security condition: the states t and t' must not differ in the values of expressions Exp that occur in an escape hatch $(D', \mathit{Exp}) \in \mathcal{H}$ with $D' \leq D$. The intention is to prevent unintended information leakage via subsequent declassifications that involve escape hatches.

Example 1. In this and the following examples we assume the two-level policy.

For illustrating the first modification, let $\mathcal{H} = \{(\mathit{low}, \mathsf{h1{+}h2})\}$, $C_1 = \mathsf{l{:=}h1{+}h2}$, and $C_2 = [\mathsf{l{:=}h1{+}h2}]$. Neither C_1 nor C_2 is strongly secure (take *low*-equal states that differ in the value of h1+h2), but both are *what*$_1$-secure. Recall that *what*$_1$-security does not aim at localizing *where* declassification occurs and, hence, declassifying assignments are treated like usual assignments (unlike in Sect. 2).

For illustrating the second modification, let $C_3 = \mathsf{h1{:=}0}; [\mathsf{l{:=}h1{+}h2}]$. This program leaks the initial value of h2 and, hence, does not comply with the security policy. In fact, this program is not *what*$_1$-secure due to the requirement $t =_D^{\mathcal{H}} t'$.

$$\begin{array}{l}\forall s, s', t : \forall i \in \{1 \ldots n\} : \forall W : \\ (V\ R\ V' \wedge \langle\!| C_i, s |\!\rangle \rightarrowtriangle \langle\!| W, t |\!\rangle \wedge s =_D^{\mathcal{H}} s') \\ \Rightarrow \exists W', t' : \langle\!| C_i', s' |\!\rangle \rightarrowtriangle \langle\!| W', t' |\!\rangle \wedge t =_D^{\mathcal{H}} t' \wedge W\ R\ W'\end{array}$$

Fig. 2. Characterization of Strong $(D, \mathcal{H})$-Bisimulation Relations (see Definition 7) where $V = \langle C_1, \ldots, C_n \rangle$, $V' = \langle C_1', \ldots, C_n' \rangle$, and $\rightarrowtriangle\ =\ \rightarrowtriangle_o \cup (\bigcup_{\mathcal{D}_1, \{D_2\} \subseteq \mathcal{D}} \rightarrowtriangle_d^{\mathcal{D}_1 \rightarrow D_2})$

Unfortunately, $what_1$-security does not satisfy the monotonicity principle (see Sect. 3.2). As a solution, we propose another security characterization.

Definition 8 (WHAT$_2$). *A program V has secure information flow while complying with the restrictions what can be declassified if $\forall D : \exists \mathcal{H}' \subseteq \mathcal{H} : V \approxeq_D^{\mathcal{H}'} V$ (brief: V is $what_2$-*secure *or $V \in WHAT_2$).*

Note that Definition 8 is also based on the notion of a strong $(D, \mathcal{H})$-bisimulation. The difference from Definition 7 is the existential quantification over $\mathcal{H}'$. This relaxation could be exploited in a security analysis by treating expressions in escape hatches like usual expressions if they are not used for declassification. Another effect of the relaxation is that the monotonicity principle is satisfied.

3.2 Prudent Principles and Compositionality

We now validate the security characterizations of this section against the prudent principles (see Sect. 2.3) and use the results to compare the characterizations.

Interestingly, WHAT$_1$ and WHAT$_2$ are preserved even if one replaces arbitrary sub-programs with semantically equivalent ones.

Theorem 5 (Strong semantic consistency). *Let C, C' be programs (possibly containing declassification commands).*

1. *If $C' \approxeq C$ and $\mathcal{C}[C] \in WHAT_1$ then $\mathcal{C}[C'] \in WHAT_1$.*
2. *If $C' \approxeq C$ and $\mathcal{C}[C] \in WHAT_2$ then $\mathcal{C}[C'] \in WHAT_2$.*

Both security conditions satisfy the conservativity principle. Additionally, $what_2$-security is a relaxation of strong security. Due to the strict handling of variables in escape hatches, $what_1$-security is not a relaxation of strong security if $\mathcal{H} \neq \emptyset$.

Theorem 6 (Conservativity).

1. *(a) If $\mathcal{H} = \emptyset$ and $V \in WHAT_1$ then V is strongly secure.*
   (b) If $\mathcal{H} = \emptyset$ and $V \in WHAT_2$ then V is strongly secure.
2. *(a) If $\mathcal{H} = \emptyset$ and V is strongly secure then $V \in WHAT_1$.*
   (b) If V is strongly secure, then $V \in WHAT_2$.

Theorem 7 (Monotonicity of Release).
Let $\mathcal{H} \subseteq \mathcal{H}'$. If $V \in WHAT_2$ for $(\mathcal{D}, \leq, \mathcal{H})$ then $V \in WHAT_2$ for $(\mathcal{D}, \leq, \mathcal{H}')$.

Example 2. Consider C_4 = `h1:=0`. Intuitively, this program has secure information flow for the two-domain policy (where $dom($`h1`$) = high$), and it also satisfies the strong security condition. For any set $\mathcal{H}$, we obtain $C_4 \in$ WHAT$_2$ from $C_4 \approxeq_{low}^{\emptyset} C_4$ (take $\mathcal{H}' = \emptyset$). However, C_4 is not $what_1$-secure for $\mathcal{H} = \{(low,$ `h1+h2`$)\}$ as it updates the variable `h1`, which occurs in the escape hatch.

Example 2 demonstrates that WHAT$_1$ does not satisfy monotonicity. The problem is that the condition $V \approxeq_D^{\mathcal{H}} V$ does not permit the updating of variables that occur in some escape hatch in $\mathcal{H}$. While such updates might lead to an information leak in subsequent assignments, they are harmless given that the variable only occurs in escape hatches that are never used for declassification. This problem does not arise with WHAT$_2$ as one can choose $\mathcal{H}'$ such that it only contains escape hatches that are used.

While we are confident that our characterizations WHAT$_1$ and WHAT$_2$ are adequate, a formal proof of the *non-occlusion* principle is not possible as we are defining what security means (as already explained for WHERE in Sect. 2.3).

However, we can analyze the compositionality of our security characterizations. We define expressions Exp, Exp' to be $(D, \mathcal{H})$*-indistinguishable* (denoted by $Exp \equiv_D^{\mathcal{H}} Exp'$) if $\forall s, t : ((s =_D^{\mathcal{H}} t \wedge \langle\!| Exp, s |\!\rangle \downarrow n \wedge \langle\!| Exp', t |\!\rangle \downarrow m) \Rightarrow n = m)$.

Theorem 8. *If $C_1 \approxeq_D^{\mathcal{H}} C_1'$, $C_2 \approxeq_D^{\mathcal{H}} C_2'$, and $V \approxeq_D^{\mathcal{H}} V'$ then*

1. $C_1;\ C_2\ \approxeq_D^{\mathcal{H}}\ C_1';\ C_2'$;
2. $\mathsf{fork}(C_1 V)\ \approxeq_D^{\mathcal{H}}\ \mathsf{fork}(C_1' V')$;
3. $B \equiv_D^{\mathcal{H}} B' \Rightarrow (\mathsf{while}\ B\ \mathsf{do}\ C_1\ \mathsf{od}\ \approxeq_D^{\mathcal{H}}\ \mathsf{while}\ B'\ \mathsf{do}\ C_1'\ \mathsf{od})$;
4. $(B \equiv_D^{\mathcal{H}} B' \vee C_1 \approxeq_D^{\mathcal{H}} C_2) \Rightarrow (\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}\ \approxeq_D^{\mathcal{H}}\ \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'\ \mathsf{fi})$.

Corollary 1. *If $C_1, C_2, V \in WHAT_1$ then*

1. $C_1;\ C_2 \in WHAT_1$;
2. $\mathsf{fork}(C_1 V) \in WHAT_1$;
3. *if the policy has a domain low and* $B \equiv_{low}^{\mathcal{H}} B$ *then* $\mathsf{while}\ B\ \mathsf{do}\ C_1\ \mathsf{od} \in WHAT_1$;
4. $[\forall D \in \mathcal{D} : (B \not\equiv_D^{\mathcal{H}} B \implies C_1 \approxeq_D^{\mathcal{H}} C_2)] \implies \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \in WHAT_1$.

Due to the existential quantification of $\mathcal{H}'$ in Definition 8, WHAT$_2$ is not compositional. This is illustrated by the following example.

Example 3. The programs $C_2 = [\mathsf{l{:}{=}h1{+}h2}]$ and $C_4 = \mathsf{h1{:}{=}0}$ (from Examples 1 and 2) are both *what*$_2$-secure for the set $\mathcal{H} = \{(low, \mathsf{h1{+}h2})\}$. However, neither $C_3 = C_4; C_2$ nor $C_5 = \mathsf{fork}(C_4\langle C_2\rangle)$ is *what*$_2$-secure.

In summary, none of our two characterizations WHAT$_1$ and WHAT$_2$ is superior to the respective other characterization. While WHAT$_1$ is compositional (see Corollary 1) but does not satisfy the monotonicity principle (see Example 2), WHAT$_2$ satisfies monotonicity (see Theorem 7) but is not compositional (see Example 3). It would be desirable to obtain a security characterization that is compositional and that satisfies the monotonicity principle. Unfortunately, one faces a fundamental difficulty when one also wants to control the *what* dimension of declassification. As discussed in Example 3, $C_3 = C_4; C_2$ and $C_5 = \mathsf{fork}(C_4\langle C_2\rangle)$ both violate the two-level policy for the set $\mathcal{H} = \{(low, \mathsf{h1{+}h2})\}$ and, hence, these programs should not be considered as *what*-secure. However, being able to declassify the expression $\mathsf{h1{+}h2}$ is the very purpose of the escape hatch $(low, \mathsf{h1{+}h2})$ and, hence, the program $C_2 = [\mathsf{l{:}{=}h1{+}h2}]$ should be considered as *what*-secure. The inherent trade-off becomes apparent when considering $C_4 = \mathsf{h1{:}{=}0}$. If one classifies this program as *what*-secure then one arrives at a security condition that is not compositional (as, e.g., C_3 and C_5 are not *what*-secure). However, if one classifies C_4 as not *what*-secure then one arrives at a security condition that does not satisfy monotonicity because C_4 is *what*-secure for $\mathcal{H} = \emptyset$.[3]

4 A Sound Type System for Information Flow Security

We present a security type system that can be used as a basis for automating the information flow analysis. The type system provides an integrated control of the *where* dimension and of the *what* dimension of declassification.

Definition 9. *If* $(\mathcal{D}, \leq, \rightsquigarrow)$ *is an* MLS policy with exceptions *and* $(\mathcal{D}, \leq, \mathcal{H})$ *is an* MLS policy with escape hatches *then the tuple* $(\mathcal{D}, \leq, \rightsquigarrow, \mathcal{H})$ *is an* MLS policy controlling the where and what of declassification*.*

In the following, let $(\mathcal{D}, \leq, \rightsquigarrow, \mathcal{H})$ be a policy and *dom* be a domain assignment.

The core of the type system is the rule for declassification commands as this is where declassification actually occurs. Our security characterizations in Sections 2 and 3 provide some guidance for developing such a rule, but there are still some pitfalls that one must avoid. As an example, consider the rule below, where $\mathit{Var}(\mathit{Exp})$ denotes the set of identifiers occurring in the expression Exp:

$$\frac{\begin{array}{c} \mathit{dom}(\mathit{Id}) = D \qquad \forall D' \in \mathit{sources}(\mathit{Exp}) : D' \ (\leq \cup \rightsquigarrow)\ D \qquad \mathit{Exp} \equiv^{\mathcal{H}}_{D} \mathit{Exp} \\ \forall (D', \mathit{Exp}') \in \mathcal{H} : ((D' \leq D \wedge \mathit{Id} \in \mathit{Var}(\mathit{Exp}')) \implies \mathit{Exp} \equiv^{\mathcal{H}}_{D'} \mathit{Exp}) \end{array}}{[\mathit{Id}{:=}\mathit{Exp}]} \qquad (1)$$

In the above rule, the second premise ensures that declassification complies with $\rightsquigarrow$ or, in other words, that the *where* of declassification is localized according to the policy. The third premise ensures that executing the declassification command in $(D, \mathcal{H})$-equal states leads to D-equal states. Finally, the fourth premise controls the information flow into variables that occur in escape hatches.

Nevertheless, the above typing rule is not sound in a compositional security analysis. For instance, Rule (1) allows one to derive [h1:=0] as well as [l:=h1+h2], but the sequential composition of these commands leaks the initial value of h2 and, hence, does not comply with the two-level policy for $\mathcal{H} = \{(\mathit{low}, \texttt{h1+h2})\}$. In order to avoid such problems, the rule also needs to ensure that a declassification does not enable information leakage in assignments that are executed subsequently.[4] A solution would be to forbid assignments to variables that occur in escape hatches that contain complex expressions (i.e., expressions that are not identifiers). This solution can be implemented by adding the following condition as another premise to Rule (1):

$$\forall (D', \mathit{Exp}') \in \mathcal{H} : (\mathit{Id} \in \mathit{Var}(\mathit{Exp}') \implies \mathit{Exp}' = \mathit{Id})$$

[3] It is not an option to classify C_4 as not *what*-secure for $\mathcal{H} = \emptyset$ because then one would essentially have to classify *all* assignments as not *what*-secure.

[4] Note that, in a concurrent program, such assignment may occur *after* the given declassification (sequential composition), *before* the declassification (backwards jumps due to loops), and also in a program executed by a concurrent thread.

$$\frac{}{\vdash \mathit{Const} : \emptyset} \qquad \frac{\mathit{dom}(\mathit{Id}) = D}{\vdash \mathit{Id} : \{D\}} \qquad \frac{\vdash \mathit{Exp}_1 : \mathcal{D}_1 \quad \ldots \quad \vdash \mathit{Exp}_m : \mathcal{D}_m}{\vdash \mathit{Op}(\mathit{Exp}_1, \ldots, \mathit{Exp}_m) : \bigcup_{i \in \{1,\ldots,m\}} \mathcal{D}_i}$$

Fig. 3. Type rules for expressions

$$\frac{}{\vdash \mathsf{skip}} \qquad \frac{\vdash \mathit{Exp} : \mathcal{D}' \quad \forall D \in \mathcal{D}' : D \leq \mathit{dom}(\mathit{Id}) \quad \mathit{Id} \leftarrow \mathit{Exp}}{\vdash \mathit{Id}{:=}\mathit{Exp}}$$

$$\frac{\vdash C \quad \vdash V}{\vdash \mathsf{fork}(CV)} \qquad \frac{\vdash \mathit{Exp} : \mathcal{D}' \quad \forall D \in \mathcal{D}' : D(\rightsquigarrow \cup \leq)\mathit{dom}(\mathit{Id}) \quad \mathit{Id} \leftarrow \mathit{Exp}}{\vdash [\mathit{Id}{:=}\mathit{Exp}]}$$

$$\frac{\vdash C_0 \quad \ldots \quad \vdash C_{n-1}}{\vdash \langle C_0, \ldots C_{n-1} \rangle} \qquad \frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1 \, ; C_2} \qquad \frac{\vdash B : \{\mathit{low}\} \quad \vdash C}{\vdash \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}}$$

$$\frac{\vdash C_1 \quad \vdash C_2 \quad \forall D : B \equiv_D B \Rightarrow C_1 \approxeq^{\rightsquigarrow}_D C_2 \quad \forall D : B \equiv^{\mathcal{H}}_D B \Rightarrow C_1 \approxeq^{\mathcal{H}}_D C_2}{\vdash \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}}$$

Fig. 4. Rules of the Integrated Security Type System

In the type system, we use the judgment $\vdash \mathit{Exp} : \mathcal{D}'$ instead of the function *sources*. Intuitively, $\vdash \mathit{Exp} : \mathcal{D}'$ means that if $\mathit{Id} \in \mathit{Var}(\mathit{Exp})$ then $\mathit{dom}(\mathit{Id}) \in \mathcal{D}'$ and that if $D \in \mathcal{D}'$ then there is a variable $\mathit{Id} \in \mathit{Var}(\mathit{Exp})$ with $\mathit{dom}(\mathit{Id}) = D$. The judgment is defined formally by the rules in Fig. 3, and it fulfills the requirements for the function *sources* as the following theorem shows.

Theorem 9. *If* $\vdash \mathit{Exp} : \mathcal{D}'$ *and* $\forall D' \in \mathcal{D}' : D' \leq D$ *then* $\mathit{Exp} \equiv_D \mathit{Exp}$.

To improve the readability of the typing rules, we introduce a judgment $\mathit{Id} \leftarrow \mathit{Exp}$. Intuitively, this judgment captures that Exp may be assigned to Id in a declassifying assignment. The following formal definition is based on the conditions that we have motivated earlier in this section.

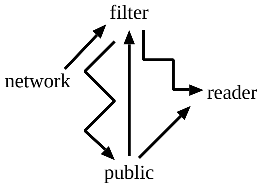
Definition 10. *We define the judgment* $\mathit{Id} \leftarrow \mathit{Exp}$ *by*

$$\begin{aligned}\mathit{Id} \leftarrow \mathit{Exp} \equiv\ & \forall D \in \mathcal{D} : ((D = \mathit{dom}(\mathit{Id}) \vee (D, \mathit{Id}) \in \mathcal{H}) \Rightarrow \mathit{Exp} \equiv^{\mathcal{H}}_D \mathit{Exp}) \\ & \wedge \forall (D', \mathit{Exp}') \in \mathcal{H} : (\mathit{Id} \in \mathit{Var}(\mathit{Exp}') \implies \mathit{Exp}' = \mathit{Id}).\end{aligned}$$

The integrated security type system for commands is presented in Fig. 4. Recall that we implicitly assume $(\mathcal{D}, \leq, \rightsquigarrow, \mathcal{H})$ to be an *MLS policy controlling the where and what of declassification*. To make the policy explicit, we use the notation $\vdash_{\mathcal{D},\leq,\rightsquigarrow,\mathcal{H}} V$ for denoting that $\vdash V$ is derivable with the typing rules.

Note that the rule for conditionals has two semantic side conditions. In this respect our presentation of the typing rules is similar to the one of the typing rules for intransitive noninterference in [9]. In that article, it is demonstrated how such semantic side conditions can be syntactically approximated by safe approximation relations in a sound way, and similar constructions are possible for our side conditions. Moreover, the premises of the typing rules for assignments and declassification involve the judgment $\mathit{Id} \leftarrow \mathit{Exp}$. Due to space limitations, we also omit the fairly straightforward syntactic approximation of Definition 10.

(a)



(b)

$$
\begin{aligned}
dom(\mathsf{mail}) &= \mathit{network}\\
dom(\mathsf{rmail}) &= \mathit{reader}\\
dom(\mathsf{pcheck}) &= \mathit{public}\\
dom(\mathsf{fmail}) &= \mathit{filter}\\
dom(\mathsf{fcheck}) &= \mathit{filter}
\end{aligned}
$$

(c)

$$
\begin{aligned}
&\mathcal{H} =\\
&\{(\mathit{reader}, \mathsf{mail}),\\
&\ (\mathit{public}, \mathsf{noMalware(mail)}),\\
&\ (\mathit{reader}, \mathsf{fmail}),\\
&\ (\mathit{public}, \mathsf{fcheck})\}
\end{aligned}
$$

Fig. 5. (a) MLS policy with exceptions, (b) domain assignment, (c) escape hatches

```
fcheck:=noMalware(mail); % check that the mail contains no malware
[pcheck:=fcheck];        % make check result public
if check then fmail:=mail % copy the mail into an auxiliary variable
         else fmail:=0 fi; % set the auxiliary variable to a dummy value
[rmail:=fmail]           % forward mail to reader
```

Fig. 6. An example for a filter program

Theorem 10 (Soundness of Security Type System).

1. *If* $\vdash_{\mathcal{D},\leq,\rightsquigarrow,\mathcal{H}} V$ *then* V *is* where-*secure.*
2. *If* $\vdash_{\mathcal{D},\leq,\rightsquigarrow,\mathcal{H}} V$ *then* V *is* what$_1$*-secure.*
3. *If* $\vdash_{\mathcal{D},\leq,\rightsquigarrow,\mathcal{H}} V$ *then* V *is* what$_2$*-secure for all* $(\mathcal{D}, \leq, \rightsquigarrow, \mathcal{H}')$ *with* $\mathcal{H} \subseteq \mathcal{H}'$.

That is, the type system is sound with respect to the security characterizations introduced in Sect. 2 and 3. In particular, the *what* and *where* of declassification in type-correct programs complies with the respectively given policy.

5 An Exemplary Security Analysis

In our application scenario, an e-mail arrives via a network and is forwarded to a user. Before the user reads an e-mail in the mail reader, the e-mail must pass a filter. The filter shall check whether the e-mail is infected by malware and shall also make the result of the check publicly available, e.g., to permit the computation of statistics about the infection rate of incoming e-mail. For this scenario, we can distinguish four security domains, a domain for the network, a domain for the filter, a domain for the mail reader, and a domain for public information. The main security requirements are that all e-mail from the network passes the filter before reaching the reader and that no e-mails are made public.

The resulting security policy is depicted in Fig. 5. The first security requirement is captured by this policy as the only path from domain *network* to domain *reader* is via domain *filter*. The second requirement is captured by the set of escape hatches as the only escape hatch with variable mail as expression has *reader* as target domain. The first requirement concerns the *where* dimension while the second requirement concerns the *what* dimension of declassification. A simple example for a filter program is depicted in Fig. 6. Note that declassifying assignments are used to declassify the result of the malware check (which depends on the variable mail) to domain *public* and to declassify an incoming mail to domain *reader*. The filter program forwards mail only if the malware check was negative. While this *what* aspect of declassification is not captured in our security policy, it would also be possible to define an MLS policy that captures this aspect. We refrain from pursuing such possibilities here.

An analysis of the filter program with the typing rules from Fig. 4 yields that the program is type correct (three applications of the rule for sequential composition, one application of the rule for conditionals, three applications of the rule for assignments, and two applications of the rule for declassifying assignments). Theorem 10 allows us to conclude that the program in Fig. 6 is *where*-secure, $what_1$-secure, and $what_2$-secure for the MLS policy in Fig. 5.

6 Related Work

Declassification is a current topic in language-based information flow security and there already is a variety of approaches to controlling declassification [16]. In the *what* dimension this survey lists, for instance, [8, 13], and in the *where*/*when* dimension, for instance, [4, 10, 9]. *Non-disclosure* is a recent approach in the *where* dimension that aims at multi-threaded programs [2, 1]. The idea is to expand the flow relation $\leq$ according to annotations at the executing sub-programs. A given expansion of $\leq$ localizes *where* declassification can occur in the program. The construction of expansions implicitly assumes that the exceptions that are permitted correspond to a transitive relation, an assumption that we do not need to make for WHERE.

Very few approaches limit declassification in more than one dimension.

According to [16], *relaxed noninterference* [7] mainly addresses the *what* dimensions, but it also addresses some aspects of the *where* dimension. Relaxed noninterference has a syntactic flavor as declassification may only involve syntactically equivalent λ-terms.[5] While this approach appears quite restrictive, the benefit is that one obtains some localization in the program as declassification can only happen *where* a particular syntactic expression occurs. Since *relaxed noninterference* only considers a two-level policy, there is no notion of limiting *where* declassification can occur in the flow policy.

According to [16], *abstract noninterference* [5] mainly addresses the *what*-dimension. In fact, it is a generalization of selective dependency like *delimited release* [13], WHAT_1, and WHAT_2. However, abstract noninterference also has similarities to robust declassification [17], which is a prominent representative for controlling the *who* dimension.

Another aspect, in which our work differs from many other approaches, is that we address concurrent programs. Lifting a security analysis from a sequential to a concurrent setting is often nontrivial as one must consider the possibility of races

[5] In [7] Li and Zdancewic use a $\beta - \eta$-equivalence. But they already point out, that it is not clear if this is an useful choice or what would be more useful.

and address the danger of internal timing leaks. For an overview on approaches addressing concurrency, we can only refer to [12] due to space restrictions.

7 Conclusion

While a number of approaches to controlling declassification in a language-based security analysis has been proposed in recent years, little work has addressed controlling multiple dimensions of declassification in an integrated fashion.

The aim of our investigation was to more adequately control the *where* and *what* of declassification. For controlling the *where* dimension, we proposed the condition WHERE, and we proved that it is compositional and satisfies the prudent principles of declassification (unlike, e.g., intransitive noninterference). For controlling *what*, we proposed the conditions WHAT_1 and WHAT_2, and we identified an inherent trade-off between the monotonicity principle and compositionality. To our knowledge, the soundness result for our type system is the first such result that clearly identifies which aspects of *where* and *what* are controlled.

The starting point for deriving our novel security characterizations was the strong security condition. The advantages of this condition include that it is compositional and robust with respect to choices of the scheduler (see [15] for a more detailed analysis). The strong security condition also rules out dangers of internal leaks in concurrent programming without making any assumptions about the possibilities of race conditions in a program. As a consequence, this condition is somewhat restrictive, which is technically due to the use of a strong bisimulation relation that requires a lock-step execution of related programs. While a less restrictive baseline characterization would be desirable, we do not know of any convincing solutions for controlling the *where* dimension in multi-threaded programs based on a less restrictive security condition.

Acknowledgments. We thank Henning Sudbrock for helpful comments. We also thank the anonymous reviewers for their suggestions.

This work was funded by the DFG in the Computer Science Action Program and by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the authors' views, and the Commission, the DFG, and the authors are not liable for any use that may be made of the information contained therein.

References

1. A. Almeida Matos. *Typing secure information flow: declassification and mobility.* PhD thesis, École Nationale Supérieure des Mines de Paris, 2006.
2. A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *In Proc. IEEE Computer Security Foundations Workshop*, 2005.
3. E. Cohen. Information transmission in sequential programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
4. M. Dam and P. Giambiagi. Information flow control for cryptographic applets, 2003. Presentation at Dagstuhl Seminar on Language-Based Security, http://kathrin.dagstuhl.de/03411/Materials2/.

5. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197, 2004.
6. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, USA, 1982.
7. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005.
8. G. Lowe. Quantifying information flow. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, page 18, Washington, DC, USA, 2002.
9. H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS 2004*, LNCS 3303, pages 129–145, Taipei, Taiwan, 2004.
10. A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate Non-Interference. *Journal of Computer Security*, 12(1):37–81, 2004.
11. A. Sabelfeld. Confidentiality for Multithreaded Programs via Bisimulation. In *Proceedings of Andrei Ershov 5th International Conference on Perspectives of System Informatics*, number 2890 in LNCS, pages 260–274, 2003.
12. A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.
13. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security*, 2004.
14. A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. In *Proceedings of the 8th European Symposium on Programming*, LNCS, pages 50–59, 1999.
15. A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–215, Cambridge, UK, 2000.
16. A. Sabelfeld and D. Sands. Dimensions and Principles of Declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society, 2005.
17. S. Zdancewic and A. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 15–26, Washington - Brussels - Tokyo, 2001.

A Operational Semantics of MWL

The intuition of a *deterministic judgment* of the form $\langle\!| C, s |\!\rangle \rightarrowtriangle \langle\!| W, t |\!\rangle$ is that command C performs a computation step in state s, yielding a state t and a vector of commands W, which has length zero if C terminated, length one if it has neither terminated nor spawned any threads, and length greater than one if new threads were spawned. The transition arrow is labeled to distinguish ordinary computation steps (labeling: $\rightarrowtriangle_o$) from declassification steps (labeling: $\rightarrowtriangle_d^{\mathcal{D}_1 \to \mathcal{D}_2}$). An inductive definition of the semantics is given by the rules in Fig. 7.

To model concurrent computations, the deterministic judgment is lifted to a *nondeterministic judgment* of the form $\langle\!| V, s |\!\rangle \rightarrow \langle\!| V', t |\!\rangle$. The intuitive meaning

$$\frac{}{\llangle \mathsf{skip}, s\rrangle \rightarrowtriangle_o \llangle \langle\rangle, s\rrangle} \qquad \frac{\llangle \mathit{Exp}, s\rrangle \downarrow n}{\llangle \mathit{Id}{:=}\mathit{Exp}, s\rrangle \rightarrowtriangle_o \llangle \langle\rangle, [\mathit{Id} = n]s\rrangle} \qquad \frac{}{\llangle \mathsf{fork}(CV), s\rrangle \rightarrowtriangle_o \llangle \langle C\rangle V, s\rrangle}$$

$$\frac{\llangle B, s\rrangle \downarrow \mathsf{True}}{\llangle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rrangle \rightarrowtriangle_o \llangle C_1, s\rrangle} \qquad \frac{\llangle B, s\rrangle \downarrow \mathsf{False}}{\llangle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rrangle \rightarrowtriangle_o \llangle C_2, s\rrangle}$$

$$\frac{\llangle B, s\rrangle \downarrow \mathsf{True}}{\llangle \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rrangle \rightarrowtriangle_o \llangle C; \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rrangle} \qquad \frac{\llangle B, s\rrangle \downarrow \mathsf{False}}{\llangle \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}, s\rrangle \rightarrowtriangle_o \llangle \langle\rangle, s\rrangle}$$

$$\frac{\llangle C_1, s\rrangle \rightarrowtriangle_o \llangle \langle\rangle, s'\rrangle}{\llangle C_1; C_2, s\rrangle \rightarrowtriangle_o \llangle C_2, s'\rrangle} \qquad \frac{\llangle C_1, s\rrangle \rightarrowtriangle_o \llangle C_1' V, s'\rrangle}{\llangle C_1; C_2, s\rrangle \rightarrowtriangle_o \llangle \langle C_1'; C_2\rangle V, s'\rrangle}$$

$$\frac{\llangle \mathit{Exp}, s\rrangle \downarrow n \quad \mathit{sources}(\mathit{Exp}) = \mathcal{D}_1 \quad \mathit{dom}(\mathit{Id}) = D_2}{\llangle [\mathit{Id}{:=}\mathit{Exp}], s\rrangle \rightarrowtriangle_d^{\mathcal{D}_1 \to D_2} \llangle \langle\rangle, [\mathit{Id} = n]s\rrangle} \qquad \frac{\llangle C_1, s\rrangle \rightarrowtriangle_d^{\mathcal{D}_1 \to D_2} \llangle \langle\rangle, s'\rrangle}{\llangle C_1; C_2, s\rrangle \rightarrowtriangle_d^{\mathcal{D}_1 \to D_2} \llangle C_2, s'\rrangle}$$

Fig. 7. Deterministic operational semantics of MWL

is that some thread C_i in V performs a step in state s resulting in the state t and some thread pool W'. The global thread pool V' results then by replacing C_i with W'. This is formalized by the rules in Fig. 8.

$$\frac{\llangle C_i, s\rrangle \rightarrowtriangle_o \llangle W', s'\rrangle}{\llangle \langle C_0 \ldots C_{n-1}\rangle, s\rrangle \to \llangle \langle C_0 \ldots C_{i-1}\rangle W' \langle C_{i+1} \ldots C_{n-1}\rangle, s'\rrangle}$$

$$\frac{\llangle C_i, s\rrangle \rightarrowtriangle \llangle W', s'\rrangle}{\llangle \langle C_0 \ldots C_{n-1}\rangle, s\rrangle \to \llangle \langle C_0 \ldots C_{i-1}\rangle W' \langle C_{i+1} \ldots C_{n-1}\rangle, s'\rrangle}$$

Fig. 8. Non-deterministic operational semantics of MWL