# Typing Illegal Information Flows as Program Effects

Ana Almeida Matos
Instituto Superior Técnico
SQIG – Instituto de Telecomunicações
ana.matos@ist.utl.pt

José Fragoso Santos
Inria Sophia Antipolis Méditérranée
jose.santos@inria.fr

## Abstract

Specification of information flow policies is classically based on a security labeling and a lattice of security levels that establishes how information can flow between security levels. We present a type and effect system for determining the least permissive relaxation of a given confidentiality policy that allows to type a program, given a fixed security labeling. To this end, sets of illegal information flows are represented as downward closure operators (here referred to as flow kernels) on a given lattice of security levels. Illegal information flows can then be seen as program effects, and their representation as flow kernels subsumes in granularity previous lattice-oriented representations of information flow policies. Effect soundness, optimality and preservation results are presented for the proposed type and effect system, for programs written in a concurrent higher-order imperative lambda-calculus with reference creation.

Our type and effect system provides a mechanism for deriving the flow kernel that characterizes the illegal flows that occur within a program, and which can be used to support runtime decisions of compliance to other policies. This point is illustrated by means of an application to a setting where local programs run under the control of a dynamic allowed flow policy.

***Categories and Subject Descriptors*** F.3.1 [*Semantics of Programming Languages*]: Program analysis

***General Terms*** Security, Languages, Verification

## 1. Introduction

Information flow security regards the compliance of program executions with a policy that specifies how information should be allowed to flow in a system. Information flow policies are usually based on a security lattice that structures security levels according to their relative degree of security, and a security labeling that assigns a security level to each resource of the system [6]. Noninterference [10] is the simplest information flow policy, which requires strict preservation of the meaning of a given ordering and assignment of security levels.

Since Volpano, Smith and Irvine's [20] first type system, a number of static analyses have been studied for enforcing different variations of noninterference [17, 18]. Most of these analyses are essentially aimed at distinguishing secure programs (that abide by a given security policy) from insecure programs (that may encode illegal information flows with respect to the given security policy). Therefore, in most type systems designed to check information flow policies, the type assigned to a program is not particularly meaningful. Implicitly, there are fundamentally only two types of programs: secure programs and insecure ones.

In this paper we present a type-based mechanism that extends the utility of classical information flow type systems, for besides allowing to decide typability, it provides information about the potential illegal flows that may result from the execution of any program. To this end, we propose to treat information flows that deviate from the original confidentiality policy as program effects [12], and use a type and effect system for assigning to each program a summary of these effects, here referred to as *declassification effect*. This point builds on previous observations [2, 5] that security levels can be seen as memory *regions* in which basic program *effects* such as reading, writing and allocation may occur. The illegal flows that are encoded in a program can then be approximated by recording how the basic effects of subexpressions are composed by each language construct. For example, in a conditional expression (if $M$ then $N_t$ else $N_f$) where $M$, $N_t$ and $N_f$ are expressions that might perform reading and writing operations, the regions that are involved in those operations can be described by associating reading and writing effects to each expression. If $M$ has reading effect $l$, this means that $M$ does not read memory regions above the level $l$. Dually, if $N_t$ or $N_f$ have a writing effect $l'$, this means that they do not write into memory regions below the level $l'$. While standard *checking* type systems reject programs for which $l'$ is not at least as confidential than $l$, our *informative* type system infers a potential information flow from level $l$ to $l'$.

Program effects must be composable in order to obtain, from the effects of sub-expressions, an effect that represents

*PLAS'12* June 15, Beijing, China.
Copyright © 2012 ACM ISBN 978-1-4503-1441-1/12/06. . . $10.00

the entire expression, possibly updated with its own intrinsic effects. A semi-lattice of program effects is therefore convenient, for which we take the lattice of downward closure operators (or kernels) [7] on the original lattice of security levels. The declassification effect of a program then corresponds to a kernel on the original security lattice. Representing sets of illegal flows by kernels leads to a natural framework for analyzing and enforcing information flow security.

Applications of the proposed framework go beyond statically deciding program security, as kernels provide valuable information for efficiently making dynamic security decisions. In this paper we demonstrate one such application, by considering a scenario where programs execute under the control of a dynamic runtime allowed flow policy that describes all the illegal information flows that may take place. We show that flow kernels present adequate properties for reasoning within this setting, by viewing the declassification effect as the least permissive authority to which a program complies. Indeed, once the flow kernel of a program has been extracted, its comparison to other flow policies can be done efficiently, without re-analyzing the program.

The main contributions of this paper are the following:

- A detailed discussion on the use of kernels as a means to specify relaxations of a given confidentiality policy. This representation is shown to offer strictly more granularity than previously used “flow policies”.
- A formulation of bisimulation-based noninterference and of a type and effect system for enforcing it over an expressive core-ML language that are explicitly customizable with any given kernel on the original security setting.
- A type and effect system for computing the strictest declassification effect to which a program complies. Or equivalently, for determining the least relaxation of the original security policy that renders the program secure.

The paper starts with a brief presentation of the language (Section 2), followed by a exposition on the use of kernels as security lattices which constitutes the framework for the remaining sections (Section 3). Then, a parameterizable version of noninterference is formulated (Section 4). The following two sections present the corresponding classical information flow type system (Section 5), and a new informative type system for calculating the declassification effect (Section 6). Then, an application to the problem of enabling external dynamic updates to allowed policies is presented (Section 7). The paper ends with a discussion of related work (Section 8) and concludes (Section 9). Proofs can be consulted in the companion technical report.

## 2. Language

In this section we define the target language of our study, an imperative higher-order $\lambda$-calculus with reference creation, a fairly standard core-ML [13] where programs run in a concurrent setting. Choosing such an expressive language facilitates the argumentation for key ideas of the paper, in particular the view of the extracted flow kernel in the role of a program effect. Furthermore, it supports the claim that they can be extended to any other setting.

| | | | |
|---|---|---|---|
| *Variables* | $x, y$ | *References* | $a, b, c$ |
| *Values* | $V$ | $::=$ | $() \mid x \mid a \mid (\lambda x.M) \mid tt \mid ff$ |
| *Pseudo-values* | $W$ | $::=$ | $V \mid (\rho x.W)$ |
| *Expressions* | $M, N$ | $::=$ | $W \mid (M\ N) \mid (M;N) \mid$ $(\text{if } M \text{ then } N_t \text{ else } N_f) \mid$ $(\text{ref}_{l,\theta}\ M) \mid (!\ N) \mid (M := N)$ |

**Figure 1.** Syntax of expressions.

***Syntax.*** The language of *expressions*, defined in Figure 1, is based on a call-by-value $\lambda$-calculus extended with the imperative constructs of ML, conditional branching and boolean values (here the $(\rho x.W)$ construct provides for recursive values). Variables $x$ and references $a, b, c$ are drawn from the disjoint countable sets *Var* and ***Ref***, respectively. This means in particular that reference names are not associated with any security labels or types at the language level. A mapping from references to security levels and types will be established in Section 4, during the security analysis. Nevertheless, reference names can be created at runtime, by a construct that is annotated with a type and security level that should be associated with the new reference. As we shall see, these security annotations do not play any role in the operational semantics (they will be used at a later stage of the analysis).

The evaluation relation is a transition relation between configurations of the form $\langle P, S\rangle$ where: $P \in \mathbb{N}^{\boldsymbol{Exp}}$ is a pool (multiset) of expressions that run concurrently and the *memory* or store $S : \boldsymbol{Ref} \rightarrow Val$ is a mapping from a finite set of references to values. The set brackets are omitted when pools of expressions are singletons.

***Operational Semantics.*** The semantics of the language is defined as a small step operational semantics on configurations given in Figure 2. The call-by-value evaluation order is specified by writing expressions using *evaluation contexts*. We write $\mathrm{E}[M]$ to denote an expression where the subexpression $M$ is placed in the evaluation context E, obtained by replacing the occurrence of $[]$ in E by $M$.

*Evaluation Contexts*

$$\mathrm{E} \quad ::= \quad [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E};N) \mid (\text{ref}_{l,\theta}\ \mathrm{E}) \mid (!\ \mathrm{E}) \mid (\mathrm{E} := N) \mid (V := \mathrm{E}) \mid (\text{if E then } N_t \text{ else } N_f)$$

We use some notations and conventions for defining transitions on configurations. Given a configuration $\langle P, S\rangle$, the sets $\text{dom}(S)$ and $\text{rn}(P)$ denote, respectively, the set of reference names that are mapped by $S$, and the set of reference names that occur in the expressions of $P$ (this notation is extended in the obvious way to expressions). The set of

$$\langle \mathrm{E}[((\lambda x.M)\ V)], S\rangle \underset{\varepsilon}{\rightarrow} \langle \mathrm{E}[\{x \mapsto V\}M], S\rangle$$
$$\langle \mathrm{E}[(\text{if } tt \text{ then } N_t \text{ else } N_f)], S\rangle \underset{\varepsilon}{\rightarrow} \langle \mathrm{E}[N_t], S\rangle$$
$$\langle \mathrm{E}[(\text{if } f\!f \text{ then } N_t \text{ else } N_f)], S\rangle \underset{\varepsilon}{\rightarrow} \langle \mathrm{E}[N_f], S\rangle$$
$$\langle \mathrm{E}[(V;N)], S\rangle \underset{\varepsilon}{\rightarrow} \langle \mathrm{E}[N], S\rangle$$
$$\langle \mathrm{E}[(\rho x.W)], S\rangle \underset{\varepsilon}{\rightarrow} \langle \mathrm{E}[(\{x \mapsto (\rho x.W)\}\ W)], S\rangle$$
$$\langle \mathrm{E}[(!\ a)], S\rangle \underset{\varepsilon}{\rightarrow} \langle \mathrm{E}[S(a)], S\rangle$$
$$\langle \mathrm{E}[(a := V)], S\rangle \underset{\varepsilon}{\rightarrow} \langle \mathrm{E}[()], [a := V]S\rangle$$
$$\langle \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)], S\rangle \xrightarrow[(a:\theta\ \mathrm{ref}_l)]{} \langle \mathrm{E}[a], [a := V]S\rangle,\ a \textit{ fresh in } S$$
$$\frac{\langle P, S\rangle \underset{lab}{\longrightarrow} \langle P', S'\rangle \quad \langle P \cup Q, S\rangle \textit{ well formed}}{\langle P \cup Q, S\rangle \underset{lab}{\longrightarrow} \langle P' \cup Q, S'\rangle}$$

**Figure 2.** Operational semantics.

variables occurring free in $M$ is denoted by $\mathrm{fv}(M)$. We restrict our attention to well formed configurations $\langle P, S\rangle$ satisfying the conditions that $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$, and that, for any $a \in \mathrm{dom}(S)$, $\mathrm{rn}(S(a)) \subseteq \mathrm{dom}(S)$. The capture-avoiding substitution of $W$ for the free occurrences of $x$ in $M$ is denoted by $\{x \mapsto W\}M$. The operation of adding or updating the image of an object $z$ to $z'$ in a mapping $Z$ is denoted $[z := z']Z$.

For convenience of the analysis that follows, each single transition of the operational semantics is labeled with information regarding the security level and type of references that are created at runtime. Transitions are thus decorated with labels $lab$ with the syntax

$$\textit{Labels} \quad lab \ ::= \ \varepsilon \mid a : \theta\ \mathrm{ref}_l \mid lab\ \centerdot\ lab'$$

of the form $a : \theta\ \mathrm{ref}_l$, when a reference named $a$, type $\theta$ and security level $l$ is created during that step, and $\varepsilon$, when no reference is created. Labels propagate to transitions of pools of expressions in the obvious way. The relation $\underset{lab}{\longrightarrow}^*$ denotes the reflexive and transitive closure of the transition relation $\underset{lab'}{\longrightarrow}$, where $lab$ is the concatenation of each $lab'$ in the individual steps. Empty labels ($\varepsilon$) may be omitted.

## 3. Relaxing security settings

Confidentiality policies are classically founded on the representation of confidentiality requirements by a *lattice of security levels* $\mathcal{L} = \langle L, \sqsubseteq, \sqcap, \sqcup, \top, \bot\rangle$ and a *security labeling* that maps each resource to a security level [6]. Intuitively, information pertaining to references labeled with $l_2$ can be legally transfered to references labeled with $l_1$ only if $l_2 \sqsubseteq l_1$, in which case $l_1$ is said to be at least as confidential than $l_2$. Each pair $\langle \mathcal{L}, \Delta\rangle$ consisting of a security lattice $\mathcal{L}$ and a security labeling $\Delta : \textbf{\textit{Ref}} \rightarrow L$ is here referred to as a *security setting*. In this section, we show how downward closure operators (or kernels) can be used to express arbitrary relaxations on any given security setting and how to construct the least permissive relaxation of a security setting that is consistent with a given set of potentially illegal information flows.

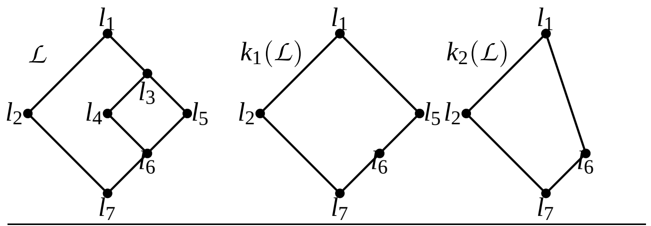


**Figure 3.** An example of two kernels on a given lattice.

As a running example, consider the leftmost confidentiality lattice that is depicted in the Hasse diagram of Figure 3. In this figure, the most confidential security level $\top$ corresponds to $l_1$, whereas the lowest level $\bot$ corresponds to $l_7$. As usual, information is allowed to flow upwards, for instance from security level $l_6$ to security level $l_3$, since $l_6 \sqsubseteq l_3$, but not from $l_4$ to $l_5$, since $l_4 \not\sqsubseteq l_5$. Under the security lattice $\mathcal{L}$ and considering a security labeling $\Delta$ such that $\Delta(x) = l_5$ and $\Delta(y) = l_4$, the program

$$(x := (!\ y)) \tag{1}$$

clearly sets up an illegal information flow.

***Kernels as relaxed security settings.*** An operator $k : L \rightarrow L$, defined on an arbitrary lattice $\mathcal{L} = \langle L, \sqsubseteq, \sqcap, \sqcup, \top, \bot\rangle$, is a *downward closure operator* if it is monotone, idempotent and restrictive, that is for every level $l \in L$, $k(l) \sqsubseteq l$. The image of every kernel $k : L \rightarrow L$, when equipped with the same order relation of $\mathcal{L}$, is also a lattice. In fact, it is a sublattice of $\mathcal{L}$, denoted by $k(\mathcal{L}) = \langle k(L), \sqsubseteq^k, \sqcup^k, \sqcap^k, \bot^k, \top^k\rangle$, where for every set of security levels $I \subseteq k(L)$:

i. for every $l_1, l_2 \in k(L)$, we have $l_1 \sqsubseteq^k l_2$ if $l_1 \sqsubseteq l_2$

ii. $\sqcup^k I = \sqcup I$

iii. $\sqcap^k I = k(\sqcap I)$

iv. $\bot^k = \bot$

v. $\top^k = k(\top)$

Informally, kernels map elements of a lattice to lower ones, while preserving their relative order. From an information flow analysis standpoint, when applied to security lattices, this means that kernels preserve the original legal information flows, and possibly introduce new flows due to the collapsing of security levels. Thus, we overload the notation so that for two arbitrary security levels $l_1, l_2 \in L$ we write $l_1 \sqsubseteq^k l_2$ if $k(l_1) \sqsubseteq^k k(l_2)$. We say that a kernel on $\mathcal{L}$ *admits* an information flow $(l_1, l_2) \in L \times L$, if $k(l_1) \sqsubseteq^k k(l_2)$. Analogously, a kernel $k$ is said to *admit* a set of information flows $F$ if it admits every flow $(l_1, l_2) \in F$.

Given a security setting $\langle \mathcal{L}, \Delta\rangle$, every kernel $k$ on $\mathcal{L}$ yields a new setting $\langle k(\mathcal{L}), k \circ \Delta\rangle$, hereby denoted $\langle \mathcal{L}, \Delta, k\rangle$. The new security setting can be viewed as a relaxation of the original one, since it collapses an arbitrary number of levels of $\mathcal{L}$ into lower ones, possibly allowing information flows that are not admitted by the initial setting. Concretely, if two references $a_1, a_2 \in \textbf{\textit{Ref}}$ are labeled with distinct security levels $l_1$ and $l_2$ respectively, such that $l_1 \not\sqsubseteq l_2$, then the content of

$a_1$ cannot be legally transfered to $a_2$. However, the same information flow is deemed legal when considering a security setting generated by a kernel $k$ on $\mathcal{L}$ such that $k(l_1) \sqsubseteq k(l_2)$.

In the example of Figure 3, the two rightmost lattices represent the image of two kernels $k_1$ and $k_2$ on $\mathcal{L}$, such that:

- $k_1(l_3) = l_5$, $k_1(l_4) = l_6$ and $k_1(l) = l$ for all other levels.
- $k_2(l_3) = l_6$, $k_2(l_4) = l_6$, $k_2(l_5) = l_6$ and $k_2(l) = l$ for all other levels.

Informally, each security level of the original lattice $\mathcal{L}$ that is not shown in the representation of a kernel is mapped by the kernel to the highest level that is depicted below it. For instance, level $l_3$ is collapsed into level $l_5$ by kernel $k_1$, and into level $l_6$ by kernel $k_2$. As a result, program (1) is still illegal under the security setting $\langle k_1(\mathcal{L}), k_1 \circ \Delta \rangle$, since $k_1(l_4) = l_6 \not\sqsubseteq^{k_1} k_1(l_5) = l_5$, but is legal under the security setting $\langle k_2(\mathcal{L}), k_2 \circ \Delta \rangle$, since $k_2(l_4) = l_6 \sqsubseteq^{k_2} k_2(l_5) = l_6$.

***Lattice of security settings*** Given a lattice $\mathcal{L} = \langle L, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$, the set of all kernels on $\mathcal{L}$, denoted by $dco(\mathcal{L})$, when ordered according to the usual order relation $\preccurlyeq$ such that, for any two kernels $k_1, k_2 : L \to L$

$$k_1 \preccurlyeq k_2 \Leftrightarrow \forall l \in L.\ k_1(l) \sqsubseteq k_2(l)$$

yields a lattice $\langle dco(\mathcal{L}), \preccurlyeq, \curlyvee, \curlywedge, \Omega, \mho \rangle$, where for every set of kernels $K \subseteq dco(\mathcal{L})$, and for any level $l \in L$ we have:

i. $(\curlywedge K)(l) = l$ if for each $k \in K$, $k(l) = l$

ii. $(\curlyvee K)(l) = \sqcup\{k(l) \mid k \in K\}$

iii. $\mho(l) = l$

iv. $\Omega(l) = \bot$

This lattice can be interpreted as a *lattice of relaxations* of the original security setting. Accordingly, given two kernels $k_1$ and $k_2$ on $\mathcal{L}$, $k_1$ is less permissive than $k_2$ if $k_2 \preccurlyeq k_1$, since $k_2$ collapses more levels of the original lattice into each other, thus possibly admitting more information flows than those which are admitted by $k_1$. That is, all the information flows admitted by $k_1$ are also admitted by $k_2$. Consequently the composition of a given a kernel $k_1$ with a more permissive kernel $k_2$ always yields the more permissive kernel $k_2$. These intuitions are precisely stated in the following lemma.

LEMMA 1. *Given a lattice $\mathcal{L}$ and two kernels $k_1, k_2$ on $\mathcal{L}$ such that $k_2 \preccurlyeq k_1$, the following holds:*

*i.* $k_1 \circ k_2 = k_2 \circ k_1 = k_2$

*ii.* $\forall l_1, l_2 \in \mathcal{L} \ :\ l_1 \sqsubseteq^{k_1} l_2 \Rightarrow l_1 \sqsubseteq^{k_2} l_2$

Given two arbitrary kernels $k_1, k_2$ on $\mathcal{L}$, the kernel $k_1 \curlyvee k_2$ is the most permissive kernel that does not permit any more flows than those which are permitted by both $k_1$ and $k_2$. Conversely, $k_1 \curlywedge k_2$ can be seen as the least permissive kernel that admits all the information flows which are either allowed by $k_1$ or by $k_2$. The identity mapping $\mho$ is the least permissive kernel, since it admits no illegal information flows with respect to the original lattice (every security level is mapped to itself) and $\Omega$ is the most permissive kernel since it maps every security level to the bottom of the original lattice, thus collapsing all levels into the bottom level.

Returning to Figure 3, in order to verify that $k_2 \preccurlyeq k_1$, it is enough to notice that the lattice corresponding to $k_2$ is a sub-lattice of the lattice corresponding to $k_1$.

***Admitting sets of illegal information flows*** In this paper, the kernels on $\mathcal{L}$ are used to over-approximate arbitrary sets of illegal information flows. To this end, we introduce an operator $\Rsh\colon dco(\mathcal{L}) \times (L \times L) \to dco(\mathcal{L})$, that given a kernel $k$ on $\mathcal{L}$ and two levels $l_1, l_2 \in L$ yields the least permissive kernel below $k$ that admits the flow $(l_1, l_2)$, denoted by $\Rsh_k [l_1, l_2]$ and defined as follows:

$$\Rsh_k [l_1, l_2](l) = \begin{cases} k(l \sqcap l_2) & \text{if } l \sqsubseteq l_1 \\ k(l) & \text{otherwise} \end{cases}$$

Naturally, if $l_1 \sqsubseteq l_2$, then $\Rsh_k [l_1, l_2] = k$. One can prove that given an arbitrary set of information flows $F$, the order by which individual flows are taken from $F$ to construct the intended kernel does not influence the result. In the simplest case, given a kernel $k$ and two flows $(l_1, l_2), (l_3, l_4) \in L \times L$:

$$\Rsh_{(\Rsh_k[l_1,l_2])} [l_3, l_4] = \Rsh_{(\Rsh_k[l_3,l_4])} [l_1, l_2]$$

This allows us to overload the notation and extend the $\Rsh$ to sets of information flows $F$, where the operator $\Rsh\colon dco(\mathcal{L}) \times \mathcal{P}(L \times L) \to dco(\mathcal{L})$ assigns to each kernel $k$ and arbitrary set of information flows $F$ the least permissive kernel below $k$ that admits all the flows in $F$:

$$\Rsh_k F = \begin{cases} k & \text{if } F = \emptyset \\ \Rsh_{(\Rsh_k[l_1,l_2])} F' & \text{if } F = F' \cup \{(l_1, l_2)\} \end{cases}$$

Throughout the paper, $\Rsh_\mho F$ is abbreviated to $\Rsh F$. The following lemma states that $\Rsh_k F$ satisfies the desired property.

LEMMA 2. *Given a lattice of security levels $\mathcal{L}$, a kernel $k$ on $\mathcal{L}$ and a binary relation $F \subseteq L \times L$, the greatest kernel $k^*$ such that $k^* \preccurlyeq k$ and $k^*$ admits all the information flows in $F$ is given by $\Rsh_k F$.*

Note that there is no bijection between the set of kernels on $\mathcal{L}$ and the set of all binary relations (here regarded as sets of information flows) on $L$. Indeed, it is easy to see that the least permissive kernels that admit two given sets of information flows $F_1$ and $F_2$, respectively, may coincide. For instance, in the example shown in Figure 3, kernel $k_2$ is the least permissive kernel that admits both $\{(l_3, l_4), (l_3, l_5)\}$ and $\{(l_3, l_6)\}$. In this sense, the least permissive kernel that admits a given set of information flows $F$ can be seen as an over-approximation of $F$.

Note that any set of information flows $F \subseteq L \times L$ can be easily preprocessed in order to reduce the number of recursive calls of the $\Rsh$ operator. Firstly, all flows in $F$ that are permitted by the original lattice can be eliminated. Additionally, for every three security levels $l_1, l_2, l_3$, if $(l_1, l_3), (l_2, l_3) \in F$, then both flows can be equivalently replaced by a single flow $(l_1 \sqcup l_2, l_3)$. Conversely, if $(l_1, l_2), (l_1, l_3) \in F$, then both flows can be equivalently replaced by a single flow $(l_1, l_2 \sqcap l_3)$.

Hence, the recursive application of these two rules to all the flows in $F$, yields a new set of information flows $F'$ corresponding to the same kernel and which is strictly smaller than the number of levels of the original security lattice.

### 3.1 Security levels as sets of principals

In order to compare the granularity of representing security settings by means of kernels with other approaches, we now consider a scenario in which confidentiality levels are sets of principals $p, q \in \boldsymbol{Pri}$ that have read-access rights to references. Thus, given a reference $a \in \boldsymbol{Ref}$ with label $l$, principal $p$ is allowed to read the value of $a$ if $p \in l$. In this scenario, the base lattice of security levels is given by $\langle \mathcal{P}(\boldsymbol{Pri}), \supseteq \rangle$. Observe that $\emptyset$ corresponds to the highest confidentiality level since no principal is allowed to read the value of references labeled with $\emptyset$, whereas $\boldsymbol{Pri}$ is the lowest confidentiality level, since every principal is allowed to read the value of references labeled with $\boldsymbol{Pri}$. As an example, the leftmost imagine in Figure 4 depicts the lattice that corresponds to the set of principals $\boldsymbol{Pri} = \{A, B, C\}$ (Alice, Bob and Charlie).

A *flow relation* [2, 14] $f$ is a reflexive and transitive binary relation on $\boldsymbol{Pri}$ that can be used to specify relaxations over a principal-based security lattice, so that if $(p, q) \in f$, then information may flow from principal $p$ to principal $q$. That is, information that principal $p$ is allowed to read may also be read by principal $q$. As kernels, flow relations can also be ordered according to their relative permissivity. Naturally, a flow relation $f_1$ is more permissive than a flow relation $f_2$ iff $f_1 \supseteq f_2$. In fact, the set of all the flow relations over $\boldsymbol{Pri}$, denoted $\mathcal{F}(\boldsymbol{Pri})$, ordered under reverse subset inclusion is a complete lattice $\langle \mathcal{F}(\boldsymbol{Pri}), \preccurlyeq, \curlyvee, \curlywedge, \Omega, \mho \rangle$, where for any family of flow relations $F \subseteq \mathcal{F}(\boldsymbol{Pri})$:

i. $\curlywedge F = \cup F$

ii. $\curlyvee F = \cap F$

iii. $\Omega = \boldsymbol{Pri} \times \boldsymbol{Pri}$

iv. $\mho = \{(p, p) \mid p \in \boldsymbol{Pri}\}$

An operator on a lattice is said to be *co-additive* if it preserves the meet operation. More precisely, for principal-based lattices, this means that given an arbitrary lattice $\mathcal{P}(\boldsymbol{Pri})$, a kernel $k : \boldsymbol{Pri} \to \boldsymbol{Pri}$ is co-additive if for every set $L \subseteq \boldsymbol{Pri}$, we have $k(\bigcup L) = \bigcup\{k(l) \mid l \in L\}$. It is interesting to observe that there is a one-to-one correspondence between the set of flow relations and the set of co-additive downward closure operators on the corresponding lattice. Indeed, every flow relation $f$ on $\boldsymbol{Pri}$ corresponds to a co-additive downward closure operator on the lattice $\langle \mathcal{P}(\boldsymbol{Pri}), \supseteq \rangle$ given by:

$$l \uparrow_f = \{q \mid \exists p \in l.\ (p, q) \in f\}$$

Intuitively, the operator maps confidentiality levels to the actual set of principals that are allowed to read at that level, in the presence of the flow relation $f$. So for instance, the label $\{$Alice, Bob$\}$ allows Alice, Bob and Charlie as readers, under the policy $\{$Alice,Charlie$\}$. Conversely, every co-additive downward closure operator $k$ on $\langle \mathcal{P}(\boldsymbol{Pri}), \supseteq \rangle$ corresponds to a flow relation $f_k$:

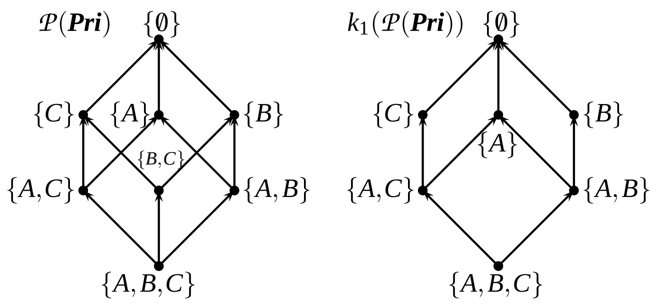


**Figure 4.** An example of a kernel that cannot be represented using flow relations.

$$f_k = \{(p, q) \mid p, q \in \boldsymbol{Pri} \ \&\ q \in k(\{p\})\}$$

For any set of principals $\boldsymbol{Pri}$, the set of co-additive kernels on $\mathcal{P}(\boldsymbol{Pri})$, when ordered in the usual way, is also a lattice. In fact, the lattice of co-additive kernels on $\mathcal{P}(\boldsymbol{Pri})$ is order-isomorphic to the lattice of flow relations on $\boldsymbol{Pri}$. Hence, given a security setting $\langle \mathcal{P}(\boldsymbol{Pri}), \Delta \rangle$, where $\Delta : \boldsymbol{Ref} \to \mathcal{P}(\boldsymbol{Pri})$, every flow relation $f$ generates a relaxation of the original security setting given by $\langle \uparrow_f (\mathcal{P}(\boldsymbol{Pri})), \uparrow_f \circ \Delta \rangle$.

It is important to understand that, as a consequence of the above observations, there are kernels on $\mathcal{P}(\boldsymbol{Pri})$ that cannot be equivalently expressed as flow relations on $\boldsymbol{Pri}$. In this sense, by representing relaxations over an initial security setting using arbitrary kernels, more granularity is achieved than by using flow relations. To illustrate this remark, and returning to Figure 4, suppose we want to establish a confidentiality policy such that principal $A$ is allowed to read everything that can be read by both $B$ and $C$. This policy is precisely captured by the following kernel $k_1$ on $\mathcal{P}(\boldsymbol{Pri})$

- $k_1(\{B, C\}) = \{A, B, C\}$
- $k_1(l) = l$ for every other set $l \subseteq \{A, B, C\}$

that is represented in the leftmost lattice of Figure 4. One can easily check that $k_1$ is not co-additive, since:

$$k_1(\{B, C\}) = \{A, B, C\} \neq \bigcup_{p \in \{B, C\}} k_1(p) = \{B, C\}$$

This is thus an example of a policy that cannot be expressed using flow relations.

## 4. Noninterference

In this section we introduce a bisimulation-based definition of noninterference that is parameterized with a security setting of the form $\langle \mathcal{L}, \Sigma, k \rangle$, where $k$ is a kernel on $\mathcal{L}$. The definition also makes use of a *reference labeling* $\Sigma : \boldsymbol{Ref} \to L \times \boldsymbol{Typ}$, whose left projection $\Sigma_1$ corresponds to the security labeling $\Delta$ (see previous section), and right projection corresponds to the *type labeling* $\Sigma_2 : \boldsymbol{Ref} \to \boldsymbol{Typ}$.

Given a security lattice $\mathcal{L}$ and a security labeling $\Sigma_1$, two memories $S_1$ and $S_2$ are said to be indistinguishable at level $l \in L$ with respect to a kernel $k$, written $S_1 =_l^{\mathcal{L}, \Sigma_1, k} S_2$, if they coincide in all references assigned to security levels less or equal than $l$. Formally, $S_1 =_l^{\mathcal{L},\Sigma_1,k} S_2$ if and only if for every reference $a \in \boldsymbol{Ref}$, if $\Sigma_1(a) \sqsubseteq^k l$, then $S_1(a) = S_2(a)$ holds.

The language defined in Section 2 is a higher-order language, where values stored in memory can be used by programs to build expressions that are then executed. For example, the expression $((!\ a)\ ())$ can evolve into an insecure program when running on a memory that maps a reference $a$ to a lambda-abstraction whose body consists of an insecure expression. In order to avoid considering all such programs insecure, it is necessary to make assumptions concerning the contents of the memory. Here, memories are assumed to be compatible to the given security setting and typing environment, requiring typability of their contents with respect to the type system that is defined in the next section. A memory $S$ is then said to be $(\mathcal{L},\Sigma,k,\Gamma)$-*compatible* if for every reference $a \in \mathrm{dom}(S)$ its value $S(a)$ satisfies $\Gamma \vdash^k_{\mathcal{L},\Sigma} S(a) : \Sigma_2(a)$.

Since we are considering pools of expressions that run concurrently, our information flow property is formulated in terms of a bisimulation [3], based on the small-step semantics defined in Section 2. The following relation pairs pools of expressions that show the same behavior on the low part of two states.

DEFINITION 1 ($\approx^{\mathcal{L},\Sigma,k}_{\Gamma,l}$). *A $(\mathcal{L},\Sigma,k,\Gamma,l)$-bisimulation is a symmetric relation $\mathcal{R}^\Sigma$ on pools of expressions that satisfies, for all $(\mathcal{L},\Sigma,k,\Gamma)$-compatible memories $S_1, S_2$:*
*$P_1\ \mathcal{R}^\Sigma\ P_2$ and $\langle P_1,S_1\rangle \underset{lab}{\longrightarrow} \langle P_1',S_1'\rangle$ and $S_1 =_l^{\mathcal{L},\Sigma,k} S_2$ implies*

- *If $lab = \varepsilon$, then $\exists P_2', S_2'$ such that:*
  *$\langle P_2,S_2\rangle \underset{lab}{\longrightarrow}^* \langle P_2',S_2'\rangle$ and $S_1' =_l^{\mathcal{L},\Sigma,k} S_2'$ and $P_1'\ \mathcal{R}^\Sigma\ P_2'$*
- *If $lab = a : \theta\ \mathrm{ref}_{l'}$ and $a \notin \mathrm{dom}(S_2)$, then $\exists P_2', S_2'$ such that:*
  *$\langle P_2,S_2\rangle \underset{lab}{\longrightarrow}^* \langle P_2',S_2'\rangle$ and $S_1' =_l^{\mathcal{L},[a:=(l',\theta)]\Sigma,k} S_2'$ and*
  *$P_1' \mathcal{R}^{[a:=(l',\theta)]\Sigma} P_2'$*

*The largest $(\mathcal{L},\Sigma,k,\Gamma,l)$-bisimulation is denoted by $\approx^{\mathcal{L},\Sigma,k}_{\Gamma,l}$.*

Note that for any $\mathcal{L}$, $\Sigma$, $k$ and $l$, the set of pairs of values is an $(\mathcal{L},\Sigma,k,\Gamma,l)$-bisimulation. Furthermore, the union of all $(\mathcal{L},\Sigma,k,\Gamma,l)$-bisimulations is the largest $(\mathcal{L},\Sigma,k,\Gamma,l)$-bisimulation. Since the domain of the stores is extended during computation, the reference labeling $\Sigma$ must also be extended. In this point, the above definition differs from [2], as a consequence of keeping the reference labeling independent from the programming language level.

Due to the partiality of the equality condition defined by $=_l^{\mathcal{L},\Sigma,k}$, the relation $\approx^{\mathcal{L},\Sigma,k}_{\Gamma,l}$ is not reflexive. In fact, a program is bisimilar to itself only if the high part of the state never interferes with the low part, i.e., if no security leak can occur. This motivates the definition of the security property:

DEFINITION 2 ($(\mathcal{L},\Sigma,k,\Gamma)$-Noninterference). *A pool of expressions $P$ satisfies Noninterference with respect to a security setting $(\mathcal{L},\Sigma,k)$ and a typing environment $\Gamma$, written $P$ satisfies $(\mathcal{L},\Sigma,k,\Gamma)$-Noninterference, if it satisfies $P \approx^{\mathcal{L},\Sigma,k}_{\Gamma,l} P$ for all security levels $l \in L$.*

The above definition requires information flows occurring at any computation step and performed by any expression in the pool to comply with the security setting $(\mathcal{L},\Sigma,k)$. Making the $(\mathcal{L},\Sigma,k)$ parameters explicit highlights the fact that the security definition depends on the permissivity of the considered relaxation of the original security setting. Noninterference is thus defined as a *class* of information flow properties. In particular, one can prove that for kernels $k_1,k_2$ such that $k_2 \preccurlyeq k_1$, if $P$ satisfies $(\mathcal{L},\Sigma,k_1,\Gamma)$-noninterference then $P$ satisfies $(\mathcal{L},\Sigma,k_2,\Gamma)$-noninterference.

## 5. Customizing a type and effect system for information flow

In this section we present a type and effect system [12] that is designed to accept programs in the language of Section 2 that satisfy the noninterference as formulated in the previous section. The type and effect system, presented in Figure 5 is therefore formulated in terms of the parameters $(\mathcal{L},\Sigma,k,\Gamma)$, where typing judgments are of the form $\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s, \tau$. Their meaning is that an expression $M$ is typable with respect to the security lattice $\mathcal{L}$, reference labeling $\Sigma$ and kernel $k$, with type $\tau$ and security effect $s$, in the typing context $\Gamma : Var \rightarrow \boldsymbol{Typ}$, which assigns types to variables. The security effect $s$ is composed of three security levels: $s.r$ is the *reading effect*, an upper-bound on the security levels of the references that are read by $M$; $s.w$ is the writing effect, a lower bound on the security levels of the references that are written by $M$; and $s.t$ is the *termination effect*, an upper bound on the levels of the references on which the termination of expression $M$ might depend. The reading and termination effects are composed in a covariant way, whereas the writing effect is contravariant. Intuitively, the reading effect is used in combination with the writing effect to control direct and implicit leaks, and is also used to determine the termination effect, which is used in combination with the writing effect to control termination leaks. In this type system, the termination effect of an expression is always lower than its reading effect. Types have the following syntax (where $t$ is a type variable):

$$\tau,\sigma,\theta \ \in \ \boldsymbol{Typ} \ ::= \ t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathsf{ref}_l \mid \tau \xrightarrow{s} \sigma$$

Typable expressions that reduce to a function that takes a parameter of type $\tau$ and returns an expression of type $\sigma$, with a *latent* effect $s$ are assigned the function type $\tau \xrightarrow{s} \sigma$. Explanations to the rational of these rules are similar to the ones found in the type system of Figure 6 that appears in the next section, and are given explicitly in [2].

All the operations on security levels (lower bounds, upper bounds and comparison between security levels) are expressed in terms of the lattice $k(\mathcal{L})$, corresponding to the relaxation of $\mathcal{L}$ yielded by $k$. We use a (join) semi-lattice on security effects, that is obtained from the pointwise composition of the lattice of security effects. More precisely, $s \sqsubseteq^k s'$ iff $s.r \sqsubseteq^k s'.r$ & $s'.w \sqsubseteq^k s.w$ & $s.t \sqsubseteq^k s'.t$, from which follows

$$[\textsc{Nil}]\ \Gamma \vdash^k_{\mathcal{L},\Sigma} () : \mathsf{unit} \qquad [\textsc{BoolT}]\ \Gamma \vdash^k_{\mathcal{L},\Sigma} tt : \mathsf{bool} \qquad [\textsc{BoolF}]\ \Gamma \vdash^k_{\mathcal{L},\Sigma} ff : \mathsf{bool} \qquad [\textsc{Loc}]\ \Gamma \vdash^k_{\mathcal{L},\Sigma} a : \Sigma_2(a)\ \mathrm{ref}_{\Sigma_1(a)}$$

$$[\textsc{Var}]\ \Gamma, x:\tau \vdash^k_{\mathcal{L},\Sigma} x : \tau \qquad [\textsc{Abs}]\ \frac{\Gamma, x:\tau \vdash^k_{\mathcal{L},\Sigma} M : s,\sigma}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (\lambda x.M) : \tau \xrightarrow[k]{s} \sigma} \qquad [\textsc{Rec}]\ \frac{\Gamma, x:\tau \vdash^k_{\mathcal{L},\Sigma} W : s,\tau}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (\rho x.W) : s,\tau}$$

$$[\textsc{Ref}]\ \frac{\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s,\theta \qquad s.r \sqsubseteq^k l}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (\mathrm{ref}_{l,\theta}\ M) : s \sqcup^k \langle \bot^k, l, \bot^k \rangle, \theta\ \mathrm{ref}_l} \qquad [\textsc{Der}]\ \frac{\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s,\theta\ \mathrm{ref}_l}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (!\ M) : s \sqcup^k \langle l, \top^k, \bot^k \rangle, \theta}$$

$$[\textsc{Assign}]\ \frac{\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s,\theta\ \mathrm{ref}_l \qquad \Gamma \vdash^k_{\mathcal{L},\Sigma} N : s',\theta \qquad \begin{array}{c} s.t \sqsubseteq^k s'.w \\ s.r, s'.r \sqsubseteq^k l \end{array}}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (M := N) : s \sqcup^k s' \sqcup^k \langle \bot^k, l, \bot^k \rangle, \mathsf{unit}}$$

$$[\textsc{Cond}]\ \frac{\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s,\mathsf{bool} \qquad \Gamma \vdash^k_{\mathcal{L},\Sigma} N_t : s_t,\tau \qquad \Gamma \vdash^k_{\mathcal{L},\Sigma} N_f : s_f,\tau \qquad s.r \sqsubseteq^k s_t.w, s_f.w}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (\text{if } M \text{ then } N_t \text{ else } N_f) : s \sqcup^k s_t \sqcup^k s_f \sqcup^k \langle \bot^k, \top^k, s.r \rangle, \tau}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s, \tau \xrightarrow[k]{s'} \sigma \qquad \Gamma \vdash^k_{\mathcal{L},\Sigma} N : s'',\tau \qquad \begin{array}{c} s.t \sqsubseteq^k s''.w \\ s.r, s''.r \sqsubseteq^k s'.w \end{array}}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (M\,N) : s \sqcup^k s' \sqcup^k s'' \sqcup^k \langle \bot^k, \top^k, s.r \sqcup^k s''.r \rangle, \sigma}$$

$$[\textsc{Seq}]\ \frac{\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s,\tau \qquad \Gamma \vdash^k_{\mathcal{L},\Sigma} N : s',\sigma \qquad s.t \sqsubseteq^k s'.w}{\Gamma \vdash^k_{\mathcal{L},\Sigma} (M;N) : s \sqcup^k s', \sigma}$$

**Figure 5.** A customizable type and effect system for checking information flow.

$$s \sqcup^k s' = \langle s.r \sqcup^k s'.r, s.w \sqcap^k s'.w, s.t \sqcup^k s'.t \rangle \quad \text{and:}$$
$$\bot^k = \langle \bot^k, \top^k, \bot^k \rangle$$

We abbreviate $\Gamma \vdash^k_{\mathcal{L},\Sigma} M : \bot^k, \tau$ by $\Gamma \vdash^k_{\mathcal{L},\Sigma} M : \tau$. One can easily check that security effects that are assigned to an expression when typed with respect to a kernel $k$ are always closed under $k$.

The type system of Figure 5 is similar to the one in [2], where kernels take the place of the (less generic, see Subsection 3.1) flow policies. Another difference lies in the fact that the reference labeling is here an explicit parameter of the type system. As a result, in rule Loc, the reference type is constructed by resorting to the parameter $\Sigma$.

The following type and effect preservation result states that the type of an expression is preserved by reduction, and that its security effects "weaken" along the computations, as reads, updates and creation of references are performed and conditional branches are discarded. Values that are stored in the initial state are assumed to have the correct type.

Theorem 3 (Type and effect preservation). *Given a security setting $(\mathcal{L}, \Sigma, k)$ and typing environment $\Gamma$, if for an expression $M$ there exist $s$ and $\tau$ such that $\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s, \tau$, and if $\langle M, S \rangle \xrightarrow[lab]{} \langle M', S' \rangle$ for a memory $S$ that is $(\mathcal{L}, \Sigma, k, \Gamma)$-compatible, then:*

- *If $lab = \varepsilon$, then there is an effect $s'$ such that $s' \sqsubseteq s$ and $\Gamma \vdash^k_{\mathcal{L},\Sigma} M' : s', \tau$.*
- *If $lab = a : \theta\ \mathrm{ref}_l$ for some reference name $a$, type $\theta$ and security level $l$, then there is an effect $s'$ such that $s' \sqsubseteq s$ and $\Gamma \vdash^k_{\mathcal{L},[a:=(l,\theta)]\Sigma} M' : s', \tau$.*

Given two kernels $k_1$ and $k_2$ such that $k_1 \preccurlyeq k_2$, every expression $M$ that is typable with respect to $k_2$ is also typable with respect to $k_1$, since all the information flows that are allowed by $k_2$ are also allowed by $k_1$. Conversely, if every program $c$ that is typable with respect to $k_2$ is also typable with respect to $k_1$, then $k_1 \preccurlyeq k_2$. The following lemma formalizes this result.

Lemma 4. *Given a security lattice $\mathcal{L}$ and a reference labeling $\Sigma$, and for any two kernels $k_1, k_2$ on $\mathcal{L}$, the following two propositions are equivalent:*

- $k_1 \preccurlyeq k_2$
- *For every program $M$, if $\Gamma \vdash^{k_2}_{\mathcal{L},\Sigma} M : s_2, \tau$ for some security effect $s_2$ and type $\tau$, then $\Gamma \vdash^{k_1}_{\mathcal{L},\Sigma} M : s_1, \tau$ for some security effect $s_1$.*

From the previous lemma, we can easily conclude that given two kernels $k_1$ and $k_2$ on $\mathcal{L}$, the most permissive kernel that types all the programs which are simultaneously typable with respect to $k_1$ and $k_2$ is $k_1 \curlyvee k_2$. Analogously, the least permissive kernel that allows typing all the programs that are either typable with respect to $k_1$, or typable with respect to $k_2$ corresponds to $k_1 \curlywedge k_2$.

We now formalise soundness of the type system of Figure 5 with respect to the noninterference property of Definition 2.

Theorem 5 (Soundness for Noninterference). *Given a security setting $(\mathcal{L}, \Sigma, k)$ and a typing environment $\Gamma$, if for every $M \in P$ there exist $s$ and $\tau$ such that $\Gamma \vdash^k_{\mathcal{L},\Sigma} M : s, \tau$, then $P$ satisfies $(\mathcal{L}, \Sigma, k, \Gamma)$-Noninterference.*

$$[\textsc{Nil}_{\mathrm{I}}]\ \Gamma \vdash_{\mathcal{L},\Sigma} () : \mathsf{unit} \qquad [\textsc{BoolT}_{\mathrm{I}}]\ \Gamma \vdash_{\mathcal{L},\Sigma} tt : \mathsf{bool} \qquad [\textsc{BoolF}_{\mathrm{I}}]\ \Gamma \vdash_{\mathcal{L},\Sigma} f\!f : \mathsf{bool} \qquad [\textsc{Loc}_{\mathrm{I}}]\ \Gamma \vdash_{\mathcal{L},\Sigma} a : \Sigma_2(l)\ \mathrm{ref}_{\Sigma_1(l)}$$

$$[\textsc{Var}_{\mathrm{I}}]\ \Gamma, x:\tau \vdash_{\mathcal{L},\Sigma} x : \tau \qquad [\textsc{Abs}_{\mathrm{I}}]\ \frac{\Gamma, x:\tau \vdash_{\mathcal{L},\Sigma} M : s,\sigma}{\Gamma \vdash_{\mathcal{L},\Sigma} (\lambda x.M) : \tau \xrightarrow{s} \sigma} \qquad [\textsc{Rec}_{\mathrm{I}}]\ \frac{\Gamma, x:\tau \vdash_{\mathcal{L},\Sigma} W : s,\tau}{\Gamma \vdash_{\mathcal{L},\Sigma} (\rho x.W) : s,\tau}$$

$$[\textsc{Ref}_{\mathrm{I}}]\ \frac{\Gamma \vdash_{\mathcal{L},\Sigma} M : s,\theta}{\Gamma \vdash_{\mathcal{L},\Sigma} (\mathrm{ref}_{l,\theta}\ M) : s \sqcup \langle \bot, l, \bot, \Rsh \{(s.r, l)\}\rangle, \theta\ \mathrm{ref}_l} \qquad [\textsc{Der}_{\mathrm{I}}]\ \frac{\Gamma \vdash_{\mathcal{L},\Sigma} M : s,\theta\ \mathrm{ref}_l}{\Gamma \vdash_{\mathcal{L},\Sigma} (!\ M) : s \sqcup \langle l, \top, \bot, \Omega\rangle, \theta}$$

$$[\textsc{Ass}_{\mathrm{I}}]\ \frac{\Gamma \vdash_{\mathcal{L},\Sigma} M : s,\theta\ \mathrm{ref}_l \qquad \Gamma \vdash_{\mathcal{L},\Sigma} N : s',\theta}{\Gamma \vdash_{\mathcal{L},\Sigma} (M := N) : s \sqcup s' \sqcup \langle \bot, l, \bot, \Rsh \{(s.t, s'.w), (s.r, l), (s'.r, l)\}\rangle, \mathsf{unit}}$$

$$[\textsc{Cond}_{\mathrm{I}}]\ \frac{\Gamma \vdash_{\mathcal{L},\Sigma} M : s,\mathsf{bool} \qquad \Gamma \vdash_{\mathcal{L},\Sigma} N_t : s_t,\tau \qquad \Gamma \vdash_{\mathcal{L},\Sigma} N_f : s_f,\tau}{\Gamma \vdash_{\mathcal{L},\Sigma} (\mathrm{if}\ M\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f) : s \sqcup s_t \sqcup s_f \sqcup \langle \bot, \top, s.r, \Rsh \{(s.r, s_t.w), (s.r, s_f.w)\}\rangle, \tau}$$

$$[\textsc{App}_{\mathrm{I}}]\ \frac{\Gamma \vdash_{\mathcal{L},\Sigma} M : s, \tau \xrightarrow{s'} \sigma \qquad \Gamma \vdash_{\mathcal{L},\Sigma} N : s'',\tau}{\Gamma \vdash_{\mathcal{L},\Sigma} (M\,N) : s \sqcup s' \sqcup s'' \sqcup \langle \bot, \top, s.r \sqcup s''.r, \Rsh \{(s.t, s''.w), (s.r, s'.w), (s''.r, s'.w)\}\rangle, \sigma}$$

$$[\textsc{Seq}_{\mathrm{I}}]\ \frac{\Gamma \vdash_{\mathcal{L},\Sigma} M : s,\tau \qquad \Gamma \vdash_{\mathcal{L},\Sigma} N : s',\sigma}{\Gamma \vdash_{\mathcal{L},\Sigma} (M;N) : s \sqcup s' \sqcup \langle \bot, \top, \bot, \Rsh \{(s.t, s'.w)\}\rangle, \sigma}$$

**Figure 6.** An informative type and effect system for the declassification effect

Note that the type system can be proved sound for concurrent expressions due to the fact that termination leaks are typed away from singular expressions.

## 6. A type system for determining the declassification effect

In the previous section it was established that, given an initial security lattice $\mathcal{L}$ and a reference labeling $\Sigma$, every program that is typable, using the type system of Figure 5, with respect to a certain and kernel $k$, is in fact typable with respect to all kernels $k'$ such that $k' \preccurlyeq k$. This section presents a type system that assigns to each program the strictest kernel with respect to which it is typable. This kernel can be interpreted as a description of all the potential illegal information flows that may take place during the execution of the program. It will be referred to as the *declassification effect* of the program.

Our approach is based on the observation that the illegal flows that are encoded in a program can be seen as side effects of that program. Indeed, when viewing security levels associated to references as regions of the memory [5], basic forms of program effects such as reading and writing effects consist simply of security levels. By identifying sets of information flows with kernels, we are in fact constructing a composite form of program effect.

Figure 6 presents the type and effect system for determining the declassification effect. Typing judgments are of the form $\Gamma \vdash_{\mathcal{L},\Sigma} M : s,\tau$, differing from those of Figure 5 since the former type system is parameterized by a kernel whereas this type system is not. Moreover, a fourth component $s.d$ is added to security effects, representing the declassification effect of the expression, a flow kernel that is a lower bound on the illegal flows that might occur during the execution of $M$, and is composed in a contravariant way.

The type syntax is analogous to the one in the previous section, where the latent effect on the function type now uses the new security effects with four components. Moreover, the semi-lattice of security effects uses the relation

$$s \sqsubseteq s' \overset{def}{\Leftrightarrow} s.r \sqsubseteq s'.r \ \&\ s'.w \sqsubseteq s.w \ \&\ s.t \sqsubseteq s'.t \ \&\ s'.d \preccurlyeq s.d$$

which entails that

$$s \sqcup s' = \langle s.r \sqcup s'.r, s.w \sqcap s'.w, s.t \sqcup s'.t, s.d \curlywedge s'.d\rangle \quad \text{and:}$$
$$\bot = \langle \bot, \top, \bot, \Omega\rangle$$

In each rule, the declassification effect of an expression is at least as low in the lattice of kernels as is the declassification effects of its sub-expressions, more precisely it is at least as low as the meet ($\curlywedge$) of those kernels. In some rules, the resulting kernel is further lowered in order to take in to account new potentially illegal flows that are detected by the rule. It is elucidative to notice the correspondence between the flows that lead to the update of the declassification effect and those that are restricted by means of the $\sqsubseteq$ relation in the type system of Figure 5. Indeed, the intuitions that justify them are also analogous.

In rule $\textsc{Ref}_{\mathrm{I}}$, the direct flow from the level $s.r$ associated to the value of the expression to the new reference that is created at level $l$ is introduced; the corresponding termination leak is accounted for simultaneously, since also in this type system the termination effect of an expression is always lower than its reading effect. In rule $\textsc{Ass}_{\mathrm{I}}$, the declassification effect of the assignment expression is updated with the flow that results from the potential termination leak caused by the evaluation of the leftmost expression at level $s.t$, that could be registered while evaluating the rightmost expression at level $s'.w$; furthermore, the values into which the left and the rightmost expressions are evaluated determine which reference is written and with which value, which could lead to a direct leak from the level of their reading effects $s.r$ and $s'.r$ (respectively) to the level $l$ of the assigned reference (the corresponding termination leaks are accounted for simultaneously). In rule COND$_I$, the implicit leak from the level $s.r$ associated to the value of the tested expression into the writing levels $s_t$ and $s_f$ of the branches is introduced (as well as the corresponding termination leaks). Rule APP$_I$ registers the potential termination leak that results from the evaluation of the leftmost expression at level $s.t$ and is potentially registered during the evaluation of the rightmost expression at level $s''.w$; furthermore, the values into which the left and the rightmost expressions are evaluated determine which function is being applied and to which value, which could be registered at level $s'.w$ during the evaluation of the body of the function (similarly regarding the corresponding termination leaks). Finally, rule SEQ$_I$ incorporates the potential termination leak resulting from the evaluation of the leftmost expression from level $s.t$ that could be registered by the rightmost expression at level $s'.w$.

As an example of the computation of the declassification effect consider the following two programs

$$((a_{l_3} := (!\ b_{l_4}) * (!\ a_{l_3})); (c_{l_5} := (!\ a_{l_3}))) \qquad (2)$$

$$(\text{if } ((!\ b_{l_4}) > (!\ c_{l_5})) \text{ then } (d_{l_6} := (!\ d_{l_6}) + 1) \text{ else } ()) \qquad (3)$$

where the security setting is given by the lattice of Figure 3 with the security labeling that is obtained by assigning each variable to the level corresponding to its subscript. It is easy to see that expression (2) corresponds to kernel $k_1$ and expression (3) corresponds to kernel $k_2$. This example gives an intuition as to why, when collapsing a level to another level, the analysis collapses all the levels in between. In expression (2), after the first assignment, information pertaining to level $l_3$ depends on information pertaining to level $l_4$, however, since $l_4 \sqsubseteq l_3$, the associated kernel is still the identity kernel $\mho$. The second assignment encodes an explicit dependency between levels $l_3$ and $l_5$ and therefore the analysis collapses level $l_3$ to $l_5$. However, there is also a dependency between levels $l_4$ and $l_5$ (due to the first assignment). This dependency is assumed since $\mho(l_4) = l_4 \sqsubseteq \mho(l_3) = l_3$ and hence the analysis also collapses $l_4$ to $l_6 = l_4 \sqcap l_5$. In expression (3), the level of the guard of the if is $l_1$, which entails a dependency between levels $l_6$ and $l_1$. Thus, the analysis collapses level $l_1$ to $l_6$ resulting in kernel $k_2$.

The type and effect system of Figure 6 accepts more programs than the one in Figure 5. More precisely, it accepts all programs that are typable with respect to a security labeling that maps all references to the lowest confidentiality level $\bot$. As expected, types and effects (and in particular the declassification effect) are preserved and weakened by reduction, as the effects are performed.

THEOREM 6 (Type and effect preservation). *Given a security setting $(\mathcal{L}, \Sigma)$ and typing environment $\Gamma$, if for an expression $M$ there exist an effect $s$ and a type $\tau$ such that $\Gamma \vdash_{\mathcal{L},\Sigma} M : s, \tau$, and if $\langle M, S\rangle \xrightarrow[lab]{} \langle M', S'\rangle$ for a a memory $S$ that is $(\mathcal{L}, \Sigma, \Gamma, \Omega)$-compatible, then:*

- *If $lab = \varepsilon$, then there is an effect $s' \sqsubseteq s$ and a type $\tau$ such that $\Gamma \vdash_{\mathcal{L},\Sigma} M' : s', \tau$.*
- *If $lab = a : \theta\ \text{ref}_l$ for some reference name $a$, type $\theta$ and security level $l$, then there is an effect $s' \sqsubseteq s$ and a type $\tau$ such that $\Gamma \vdash_{\mathcal{L},[a:=(l,\theta)]\Sigma} M' : s', \tau$.*

The following theorem states that the declassification effect assigned by the type system of Figure 6 to a given program $M$ corresponds to the smallest kernel that allows typing the program with respect to the type system of Figure 5.

THEOREM 7 (Soundness and optimality of the type system). *Given a security lattice $\mathcal{L}$, a reference labeling $\Sigma$, a typing environment $\Gamma$ and an expression $M$ such that $\Gamma \vdash_{\mathcal{L},\Sigma} M : s, \tau$ for some type $\tau$ and security effect $s$, then the following propositions hold:*

*i. There is a security effect $s'$, such that $\Gamma \vdash^{s.d}_{\mathcal{L},\Sigma} M : s', \tau$.*
*ii. If there is a kernel $k$ on $\mathcal{L}$ and a security effect $s'$ such that $\Gamma \vdash^{k}_{\mathcal{L},\Sigma} M : s', \tau$, then $k \preccurlyeq s.d$.*

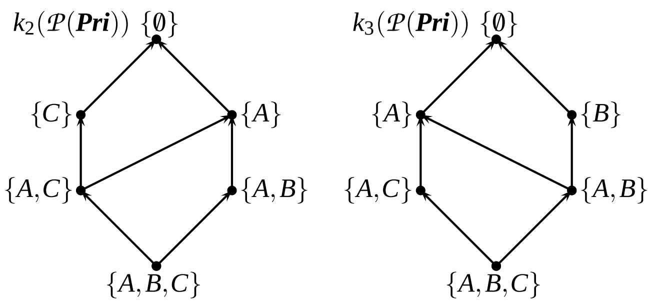
While the purpose of the type system that was presented in the previous section was to accept/reject programs according to their potential to perform illegal information flows, the main goal of the type system that is presented here is to extract a conservative approximation of those flows.

Notice that although the above result is formulated in terms of single expressions, it can be used in concurrent settings. Indeed, for a pool of expressions with declassification effect $k_1, \ldots, k_n$, Theorem 7 implies typability of all expressions with respect to any flow kernel $k$ such that $k \preccurlyeq k_1 \curlywedge \ldots \curlywedge k_n$. In general, the principles behind our informative type system can be applied to any setting that can be tackled by means of a state-oriented checking-type system.

## 6.1 Flow relations as declassification effects

In Section 3.1, flow relations were presented as an alternative means to describe flexible information flow policies, that is, relaxations of the initial (principal-based) security setting. Here we show that when taking the lattice of flow relations as the lattice of declassification effects, there ceases to be an optimal declassification effect to describe the information flows that are entailed by each program, i.e., Theorem 2 does not hold in this restricted setting.

Given a security setting $\langle \mathcal{L}, \Delta\rangle$ and an expression $M$, Theorem 7 states that the declassification effect computed by the informative type system corresponds to the least permissive kernel $k_M$ that allows to type $M$ according to the checking type system. However, this kernel may not be a co-additive kernel, i.e. it may not representable as a flow relation. Aiming at an approximation, one might hope to determine the greatest co-additive kernel that allows to type $M$, which by Theorem 7 is necessarily lower than $k_M$. However, the least upper bound of all the co-additive kernels that are lower than



$k_2(\mathcal{P}(\textbf{Pri}))$ $\{\emptyset\}$
$\{C\}$
$\{A\}$
$\{A,C\}$
$\{A,B\}$
$\{A,B,C\}$
$k_3(\mathcal{P}(\textbf{Pri}))$ $\{\emptyset\}$
$\{A\}$
$\{B\}$
$\{A,C\}$
$\{A,B\}$
$\{A,B,C\}$

**Figure 7.** Possible non-optimal co-additive kernels for accepting program (4).

$k_M$ is not necessarily lower than $k_M$. That is, for an arbitrary kernel $k_M : L \rightarrow L$, the kernel $k_M^* : L \rightarrow L$ given by

$$k_M^* = \curlyvee\{k' \mid k' \preccurlyeq k_M \;\&\; k' \text{ is co-additive}\}$$

is not necessarily lower than $k_M$. Thus concluding that when using flow relations as declassification effects there are programs for which there is no optimal declassification effect.

Returning to the example presented in Section 3.1 for the set of principals $\textbf{Pri} = \{A,B,C\}$, and considering a security labeling $\Sigma_1 : \textbf{Ref} \rightarrow \mathcal{P}(\textbf{Pri})$, such that $\Sigma_1(a) = \{B,C\}$ and $\Sigma_1(b) = \{A\}$, the program

$$(b_{\{A\}} := (!\ a_{\{B,C\}})) \tag{4}$$

has declassification effect $\Rsh_{\mho} \{(\{B,C\},\{A\})\}$, which corresponds to the kernel $k_1$ presented in Figure 4. Since $k_1$ is not co-additive, the strictest flow policy to which the above program complies must correspond to the strictest co-additive kernel below $k_1$. Figure 7 illustrates the two highest co-additive kernels below $k_1$, denoted by $k_2$ and $k_3$ respectively. Since $k_2$ and $k_3$ are not comparable to each other, we must conclude that there is no strictest flow policy among the set of flow policies to which program (4) complies (one must choose either between $k_2$ or $k_3$).

## 7. Flow kernels for a hybrid analysis

In this section we flesh out a possible application of flow kernels as a way of instrumenting the dynamic decision of whether a certain program should be allowed to run, with some degree of static processing.

Let us consider a scenario where programs run locally under a dynamic allowed information flow policy $k_A$ that describes all the information flows that can take place. In this scenario, under the security setting $\langle \mathcal{L}, \Delta \rangle$ and considering an allowed flow kernel $k_A$, the desired security property that we aim at corresponds to $(\mathcal{L}, \Sigma, k_A, \Gamma)$-Noninterference.

Since $k_A$ is unknown before program execution, when using the type and effect system of Figure 5 to certify compliance with the intended security property, the program must be (re)typed with respect to $k_A$. However, by making use of the informative type system of Figure 6, one can determine the declassification effect of the program, which corresponds to the strictest kernel to which the program complies, thus avoiding to retype it again. Instead, it is only necessary to compare the declassification effect of the program with the allowed flow kernel. In fact, one can deduce from Lemma 4 and Theorems 7 and 5 the following result:

COROLLARY 8 (Usefulness of the declassification effect). *Consider a security lattice $\mathcal{L}$, a reference labeling $\Sigma$, a typing environment $\Gamma$ and an expression $M$ such that $\Gamma \vdash_{\mathcal{L},\Sigma} M : s, \tau$ for some type $\tau$ and security effect $s$. Then for any allowed flow kernel $k_A$ we have that:*

- *if $k_A \preccurlyeq s.d$ then $P$ satisfies $(\mathcal{L}, \Sigma, k_A, \Gamma)$-Noninterference;*
- *$\Gamma \vdash^{k_A}_{\mathcal{L},\Sigma} M : s', \tau$ for some security effect $s'$ iff $k_A \preccurlyeq s.d$.*

In other words, this corollary states that it is enough to compare the allowed flow kernel with the extracted declassification effect in order to conclude for the security of the expression. Furthermore, it clarifies that there is no loss in precision with respect to the checking type and effect system.

In order to make the applicability of this result more concrete, we extend the language introduced in Section 2 to a scenario in which the allowed flow kernel can change even during program execution. In order to keep track of the declassification effect that is associated with each thread, expressions are given a name $n, m \in \textbf{Nam}$, and configuration include a policy mapping from thread names to kernels $D : \textbf{Nam} \rightarrow dco(L)$. This set, together with the pool $P$ containing all the (named) threads in the system, the store $S$ containing all the references, and the allowed flow kernel $k_A$, form configurations $\langle P, S, D, k_A \rangle$, over which the evaluation relation is defined below. For a given pool of named threads $P$, the set of thread names that occur in it is given by $\text{dom}(P)$.

The first rule is constructed over the semantics defined in Figure 2, transposing the rules for pools of single expressions to the new form of configurations, which includes naming every thread. The assumption of well-formedness includes, in addition to the requirements described for the base language, $\text{dom}(P) \subseteq \text{dom}(D)$.

$$\frac{\langle \{M\}, S \rangle \xrightarrow[lab]{} \langle \{M'\}, S' \rangle \quad \langle \{M^m\} \cup P, S, D, k_A \rangle \text{ is well formed}}{\langle \{M^m\} \cup P, S, D, k_A \rangle \rightarrow \langle \{M'^m\} \cup P, S', D, k_A \rangle}$$

The second rule describes how the allowed flow kernel can change during computations. During this transition, the pool of threads cannot compute.

$$\frac{\dagger_{k_F} P = P'}{\langle P, S, D, k_A \rangle \rightarrow \langle P', S, D, k_F \rangle}$$

Any thread that no longer complies with the new allowed kernel $k_F$ is eliminated, where $\dagger$ preempts threads whose (original) declassification effect no longer complies with the currently allowed kernel:

$$\dagger_{k_F} P = \begin{cases} \emptyset & \text{if } P = \emptyset \\ \{M^k\} \cup \dagger_{k_F} P' & \text{if } P = P' \cup \{M^k\} \text{ and } k_F \preccurlyeq k \\ \dagger_{k_F} P' & \text{if } P = P' \cup \{M^k\} \text{ and } k_F \not\preccurlyeq k \end{cases}$$

In practice, the preemption operation is a dynamic mechanism for ensuring that only threads that comply with the current allowed flow policy are permitted to execute. However, it makes use of the declassification effect that was computed statically. In this sense, it implements a simple hybrid enforcement mechanism. In order to formulate the security property, we introduce the notion of reachable configuration:

DEFINITION 3 (Reachable configuration). *We say that a configuration $\langle P', S', D, k'_A \rangle$ is reachable from configuration $\langle P, S, D, k_A \rangle$ if and only if*

- *$\langle P', S', D, k'_A \rangle = \langle P, S, D, k_A \rangle$, or*
- *there is a configuration $\langle P'', S'', D, k''_A \rangle$ that is reachable from $\langle P, S, D, k_A \rangle$, and $\langle P'', S'', D, k''_A \rangle \rightarrow \langle P', S', D, k'_A \rangle$*

The following property is ensured in this setting:

PROPOSITION 9 (Confinement to a dynamic flow policy). *Consider a set of threads $P$, a policy mapping $D$, and an initial allowed flow policy $k_A$ such that for all $M^n \in P$, we have that $\Gamma \vdash_{\mathcal{L},\Sigma} M : s, \theta$ with $k_A \preccurlyeq D(n) \preccurlyeq s.d$. If the configuration $\langle P', S', D, k'_A \rangle$ is reachable from $\langle P, S, D, k_A \rangle$, then $P|_{\mathrm{dom}(P')}$ satisfies $(\mathcal{L}, \Sigma, k_A, \Gamma)$-Noninterference.*

In other words, if a certain allowed flow policy $k_A$ is ruling at some point of the computation over a pool $P$ of threads, then all the original threads are ensured to comply with $k_A$. The result is restricted to the threads that are in the domain of $P'$, for at each point, nothing can be stated about threads that have been preempted earlier.

The problem of dealing with dynamic allowed flow policies is a complex one, and it is not within the aims of this paper to provide a final solution for it. Here we take a strong yet simple approach, that highlights the usefulness of the declassification effect as a means to describe the information flows that take place within a program. More flexible and permissive settings could be conceived. For instance, one could envisage an analysis that only requires the compliance of the rest of the program that is still to be executed to newly established allowed flow kernels. While this would lead to accept more program executions (since by Theorem 6 the declassification effect weakens, i.e., becomes stricter), it would not ensure that the program does not leak illegal information (with respect to the current policy) due to past computations.

## 8. Related work

***Type and effect systems for information flow.*** Gifford and Lucassen first introduced the concepts of effect and region in the design of a type and effect system for a higher-order language with imperative features [12].

Type systems for enforcing a variety of flexible information flow policies have been studied extensively [17, 18, 20], including type systems for higher-order languages with imperative features ([2, 5, 15], to name a few). Most works on functional languages take the orientation of associating security levels to values, as opposed to the state-oriented view of [2, 5]. In these works, there is no counterpart to the informative-type system presented here. A type system for determining a declassification effect is presented first in [1], but differs in the fundamental point that while here the declassification effect is obtained directly from the actual dependencies that are encoded in programs, in the earlier work it is based on flow policies that are declared by a declassification construct, and are therefore potentially more coarse.

In [2, 5], it is observed that information flow control is rooted on the notions of reading and writing effects where regions can be viewed as confidentiality levels that can be conditioned by a type and effect system. To the best of our knowledge the view of the actual information flows as a program effect is novel.

***Inference of declassification policies.*** Recently, Vaughan and Chong [19] present an expressive language for writing complex declassification policies, and a dataflow analysis for inferring them from a simple imperative programming language. Their policies are partially ordered by a *reveals no more information than* relation, and provide a least upper bound operator *and*, as well as a bottom *Reveal()*. Rewrite rules provide a *normalization process* for simplifying policies and establishing equivalences between them. The security condition is input-output oriented.

Closer to our representation of security lattices by kernels, is the work on the abstract interpretation [4] view of information flow by Giacobazzi and Mastroeni. They introduced [9] the notion of abstract noninterference, a weakening of noninterference given in terms of observers modeled by means of abstract interpretations of concrete semantics. In abstract noninterference the power of an observer is modeled as an upward closure operator on the domain of the program. The same authors [8] also introduce a proof system inductive on the syntax of programs that is based on the derivation of abstract noninterference assertions, which specify the noninterference degree of a program relatively to a given model of an attacker. Apart from differences in language expressivity and treatment of declassification, the precise connections and differences between this framework and ours requires further investigation.

***Dynamic allowed flow policies*** Hicks et. al [11] approach the problem of allowing dynamic external updates of an allowed flow policy, there referred to as the "permission context", for a purely functional sequential scenario without declassification, in the context of a dynamic allowed-flow policy. In their work, changes over the permission context are restricted. The property that is formulated, dubbed "noninterference between updates", guarantees that, if an update occurs during computation, then the rest of that computation complies with the new policy. The fact that this property allows for the program to implement illegal flows that have been set up before the update of the allowed policy is recognized by the authors while pointing out that a "better property is needed". While it is not our claim that the property presented in Section 7 is a better solution, for it can be criticized for being overly restrictive, we believe that the mechanism presented in this paper can be used to instrument more complex and flexible properties for solving this problem and others that require dynamic comparisons between policies.

## 9. Conclusion

Parameterizing noninterference by means of a flow kernel can be seen as a declassification condition that satisfies Sabelfeld and Sands' principles of *semantic consistency*, *conservativity* and *non-occlusion* [18]. Rocha et. al [16] argue for a strict separation between the programming and the specification of the information flow policy that the program should comply to. It is observed that most flexible information flow policies are strongly connected to the use of program annotations such as declassification operations, which are often viewed as forms of policy specification in themselves. Setting aside the fact that declassification declarations can be used for purposes other than policy specification, it is clear that it is important to be able to express and check information flow policies externally to the program. In this paper we propose an approach to this problem, by presenting a way of extracting a description of the strictest information flow policy that each program complies with.

By considering a very expressive programming language, and by establishing a connection between a standard type system for checking information flow and the new informative-type system for determining the declassification effect, we pave the way for the design of other enforcement mechanisms based on the extraction of flow kernels representing a declassification effect to any language and state-oriented information flow type system that could be considered. As future work, we plan to generalize our results one step further in the concurrent setting, in order to handle distribution and program migration. We envisage a scenario where the static analysis mechanism that is proposed in this paper can instrument runtime decisions such as migration control in a distributed setting.

***Acknowledgments*** The authors would like to thank the Indes team at INRIA and all anonymous reviewers for discussions and comments that have improved the final outcome of the paper. This work was partially supported by the Portuguese Goverment via the PhD grant SFRH/BD/71471/2010 and the KLog project PTDC/MAT/68723/2006, financed by the Fundação para a Ciência e Tecnologia.

## References

[1] A. Almeida Matos. Flow-policy awareness for distributed mobile code. In *Proc. of CONCUR 2009 - Concurrency Theory*, volume 5710 of *Lecture Notes in Computer Science*. Springer, 2009.

[2] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. *Journal of Computer Security*, 17(5):549–597, 2009.

[3] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1–2):109–130, 2002.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[5] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(02), 2005.

[6] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[7] P. Dwinger. On the closure operators of a complete lattice. In *Indagationes Math.*, volume 16, pages 560–563, 1954.

[8] R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In *Conf. of the European Association for Computer Science Logic, volume 3210 of LNCS*, pages 280–294. Springer-Verlag, 2004.

[9] R. Giacobazzi and I. Mastroeni. A proof system for abstract non-interference. *Journal of Logic and Computation*, 20(2):449–479, 2010.

[10] J. Goguen and J. Meseguer. Security policies and security models. In *Proc. of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[11] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Workshop on Foundations of Comp. Security*, pages 7–18, 2005.

[12] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL'88: 15th ACM symposium on Principles of programming languages*, pages 47–57. ACM Press, 1988.

[13] R. Milner, M. Tofte, R. Harper, and David MacQueen. *The definition of Standard ML*. MIT Press, revised edition, 1997.

[14] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *19th IEEE Symposium on Security and Privacy*, pages 186–197. IEEE Computer Society, 1998.

[15] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.

[16] B. Rocha, S. Bandhakavi, J. den Hartog, W. Winsborough, and S. Etalle. Towards static flow-based declassification for legacy untrusted programs. In *SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 93–108. IEEE Computer Society, 2010.

[17] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[18] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.

[19] J. Vaughan and S. Chong. Inference of expressive declassification policies. In *Proc. of the 2011 IEEE Symposium on Security and Privacy*, pages 180–195. IEEE Computer Society, 2011.

[20] D. M. Volpano, G. Smith, and C. E. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–188, 1996.