Access Control and Declassification *

Gérard Boudol and Marija Kolundžija

INRIA, 06902 Sophia Antipolis, France
{Gerard.Boudol,Marija.Kolundzija}@sophia.inria.fr

Abstract. We integrate programming constructs for managing confidentiality in an ML-like imperative and higher-order programming language, dealing with both access control and information flow control. Our language includes in particular a construct for declassifying information, and constructs for granting, restricting or testing the read access level of a program. We introduce a type and effect system to statically check access rights and information flow. We show that typable programs are secure, that is, they do not attempt at making illegal read accesses, nor illegal information leakage. This provides us with a natural restriction on declassification, namely that a program may only declassify information that it has the right to read.
Key words: Access control, declassification, language-based security, secure information flow, stack inspection, type and effect systems.

1 Introduction

In a world where more and more information is digitalized, and where more and more people have access to it, most often by means of dedicated software, protecting confidential data is a concern of growing importance. Controlling access rights is obviously necessary, and access control techniques have indeed been developed and implemented long ago. However, as it has been argued in [7, 10, 18] for instance, access control is not enough to ensure end-to-end confidentiality. One issue is to prevent authorized users to publicly disclose confidential data, especially when the "users" are publicly accessible softwares receiving and storing these data. Therefore, one should have means to control that such programs do not contain security bugs, and in particular that programs do not implement illegal *flows of information*. This is the aim of the language-based approach to information-flow security.

Since the pioneering work of Denning [7], the classical way of abstractly specifying secure information flow is to use a *lattice of security levels*. The "objects" – information containers – of a system are then labelled by security levels, and information is allowed to flow from one object to another if the source object has a lower confidentiality level than the target one. That is, the ordering relation on security levels determines the legal flows, and a program is secure if, roughly

* Work partially supported by the ANR-06-SETI-010 grant. The second author is supported by a PhD scholarship from Torino University.

speaking, it does not set up illegal flows from inputs to outputs. This was first formally stated via a notion of *strong dependency* by Cohen in [6], which is most often referred to as *non-interference*, according to the terminology used by Goguen and Meseguer in [9].

A lot of work has been devoted to the design of methods for analyzing information flow in programs. Since the work of Volpano, Smith and Irvine [24], the classical approach by now is to use a type system for this purpose, with well-known advantages, like preventing some programming errors at an early stage, while avoiding the run-time overhead caused by dynamic checks. Type systems for secure information flow have been designed for various languages, culminating with Jif (or JFlow, see [13]) and Flow Caml [22] as regards the size of the language (see the survey [18] for further references). In this paper we shall base our study on Core ML, a call-by-value λ-calculus extended with imperative constructs, where the "information containers," to which security levels are assigned, are memory locations – references, in ML's jargon (in an extended setting, that could also be files, entries in a database, library functions, or class names as in [2]).

There is still a number of issues to investigate in order to make the techniques developed following the language-based approach to information-flow security useful in practice – see [26] for a review of some of the challenges. One of the challenges is *declassification*. Indeed, there are many useful programs that need to declassify information, from a confidential status to a less secret one. A typical example is a password checking procedure, which delivers to any user the result of comparing a submitted password with secret information contained in a database, thus leaking a bit of confidential information. Such a program is, by definition, ruled out by the non-interference requirement, which is therefore too strong to be used in practice. The problem of taking into account declassifying programs in checking secure information flow has recently motivated a lot of work, like for instance [1, 4, 5, 12, 15, 19]. We refer to [21] for a thorough discussion of this problem. In [1], the authors introduced a programming construct for managing declassification, that consists in declaring *flow policies* with a local scope. Then for instance, the critical parts of a password checking procedure should be enclosed into a statement that it is, for a while, legal to make information flow from a secret to a public level (exactly *which* information should flow is left to the programmer's responsibility). This is supported in [1] by the definition of a new confidentiality policy, called the *non-disclosure policy*, that generalizes non-interference while allowing one to deal with declassification.

There is no constraint on using the declassification construct in [1], and this is in contrast with most other studies, which aim at restricting the use of such an operation (sometimes without justifying such constraints, for lack of a corresponding extensional notion of security). In this paper, we shall show that putting access control into the picture provides us with a natural way to restrict declassification, namely: *a program may only declassify information that it has the right to read*. To do this, we integrate into the language a formal approach to access control that has recently been introduced to deal with the "stack inspection" mechanism of Java security architecture [8, 17, 23]. That is,

we add to the language of [1] [1] the constructs for managing access control that are considered in these papers, namely: a construct $(\ell \ltimes M)$ for restricting the access right of program M to be lower than ℓ, a dual construct ($\mathsf{enable}\ \ell\ \mathsf{in}\ M$) for granting read access level at least ℓ to M, and ($\mathsf{test}\ \ell\ \mathsf{then}\ M\ \mathsf{else}\ N$) for testing whether the access level ℓ is granted by the context or not, and behave accordingly. In these constructs, ℓ is a security level, that is a member of the lattice that is used for directing information flow. In particular, the security level at which a reference is classified also represents the right to access the information it contains, that is, to read the reference. For instance, in order to transfer information classified at level ℓ_1 (or lower) into level ℓ_2 (where $\ell_1 \not\preceq \ell_2$ in the security lattice), one may write, using notations that are explained in the next section:

$$(\mathsf{test}\ \ell_1\ \mathsf{then}\ (\mathsf{flow}\ \ell_1 \prec \ell_2\ \mathsf{in}\ x_{\ell_2} := \mathbf{!}\, y_{\ell_1})\ \mathsf{else}\ ())$$

Our main contribution is as follows. We extend the type system of [1] to deal with the access control constructs, thus integrating both access control and information flow control into a single static analysis technique. A similar integration was previously done in [2], but for a different language, which in particular does not involve declassification. The combination of access control and information flow is also discussed in [16] (with some further references), in a purely functional setting, following an approach which does not seem compatible with a bisimulation-based notion of a secure program, like the non-disclosure policy. Our main results are, first, a type safety property showing that access control is indeed ensured by the type system, that is, a typable program never attempts to read a reference for which it would not be granted the appropriate reading clearance. (A similar type safety result was established in [17, 23], but for a purely functional language, with no imperative feature, thus not dealing with access control in our sense, and without any consideration for information flow, and, a fortiori, declassification). Second, we extend the soundness result of [1], showing that secure information flow is ensured by our type system. In this way, we achieve the enforcement of "end-to-end confidentiality" in our language, while restricting declassification in a natural way.

Note. For lack of space, the proofs are omitted, or only briefly sketched.

2 The language

2.1 Security (pre-)lattices

The security levels are hierarchically organized in a *pre-lattice*, a structure defined as a pair $(\mathcal{L}, \preceq)$, where $\preceq$ is a preorder relation over the set $\mathcal{L}$, that is a reflexive and transitive, but not necessarily symmetric relation, such that for any $x, y \in \mathcal{L}$ there exist a meet $x \curlywedge y$ and a join $x \curlyvee y$ satisfying:

$$x \curlywedge y \preceq x \qquad x \preceq x \curlyvee y$$

$$x \curlywedge y \preceq y \qquad y \preceq x \curlyvee y$$

$$z \preceq x\ \&\ z \preceq y\ \Rightarrow\ z \preceq x \curlywedge y \qquad x \preceq z\ \&\ y \preceq z\ \Rightarrow\ x \curlyvee y \preceq z$$

[1] We do not consider threads in this paper. They would not cause any technical difficulty, but complicate the definitions and proofs.

The pre-lattices we use are defined as follows. We assume given a set $\mathcal{P}$ of *principals*, ranged over by p, $q \ldots$ (From an access control perspective, these are also called *permissions* [2, 8], or *privileges* [17, 23], while a "principal" is a set of permissions.) A *confidentiality level* is any set of principals, that is any subset ℓ of $\mathcal{P}$. The intuition is that whenever ℓ is the confidentiality label of an object, i.e. a reference, it represents a set of programs that are allowed to get the value of the object, i.e. to read the reference. Then a confidentiality level is similar to an access-control list (i.e. a set of permissions). From this point of view, a reference labelled $\mathcal{P}$ (also denoted $\bot$) is the most public one – every program is allowed to read it – whereas the label $\emptyset$ (also denoted $\top$) indicates a secret reference, and reverse inclusion of security levels may be interpreted as indicating allowed flows of information: if a reference u is labelled ℓ, and $\ell \supseteq \ell'$ then the value of u may be transferred to a reference v labelled ℓ', since the programs allowed to read this value from v were already allowed to read it from u.

We follow the approach of [1], where declassification is achieved by dynamically updating the lattice structure of confidentiality levels, by the means of local flow policies. A *flow policy* is a binary relation over $\mathcal{P}$. We let F, $G \ldots$ range over such relations. A pair $(p, q) \in F$ is to be understood as "information may flow from principal p to principal q", that is, more precisely, "*everything that principal p is allowed to read may also be read by principal q*". As a member of a flow policy, a pair (p, q) will be written $p \prec q$. We denote, as usual, by F^* the preorder generated by F (that is, the reflexive and transitive closure of F). Any flow policy F determines a preorder on confidentiality levels that extends reverse inclusion, as follows:

$$\ell \preceq_F \ell' \quad \Leftrightarrow_{\mathrm{def}} \quad \forall q \in \ell'.\ \exists p \in \ell.\ p\, F^*\, q$$

It is not difficult to see that the preorder $\preceq_F$ induces a pre-lattice structure on the set of confidentiality levels, where a meet is simply the union, and a join of ℓ and ℓ' is

$$\{\, q \mid \exists p \in \ell.\ \exists p' \in \ell'.\ p\, F^*\, q \ \&\ p'\, F^*\, q \,\}$$

This observation justifies the following definition.

Definition (Security Pre-Lattices) 2.1. *A* confidentiality level *is any subset ℓ of the set $\mathcal{P}$ of* principals*. Given a* flow policy *$F \subseteq \mathcal{P} \times \mathcal{P}$, the confidentiality levels are pre-ordered by the relation*

$$\ell \preceq_F \ell' \quad \Leftrightarrow_{\mathrm{def}} \quad \forall q \in \ell'.\ \exists p \in \ell.\ p\, F^*\, q$$

The meet *and* join*, w.r.t. F, of two security levels ℓ and ℓ' are respectively given by $\ell \cup \ell'$ and*

$$\ell \curlyvee_F \ell' = \{\, q \mid \exists p \in \ell.\ \exists p' \in \ell'.\ p\, F^*\, q \ \&\ p'\, F^*\, q \,\}$$

2.2 Syntax and operational semantics

The language we consider is a higher-order imperative language *à la* ML, extended with constructs for dynamically granting and testing access rights, as in [2, 8, 17, 23], and a construct for introducing local flow policies, as in [1]. The

$$
\begin{array}{rcll}
M,\ N \ldots \in \mathcal{E}xpr & ::= & V \mid (\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N') \mid (MN) & \textit{expressions} \\
& \mid & M\,;N \mid (\mathsf{ref}_{\ell,\theta}\,N) \mid (!\,N) \mid (M := N) & \\
& \mid & (\ell \ltimes M) \mid (\mathsf{enable}\ \ell\ \mathsf{in}\ M) \mid (\mathsf{test}\ \ell\ \mathsf{then}\ M\ \mathsf{else}\ N) & \\
& \mid & (\mathsf{flow}\ F\ \mathsf{in}\ M) & \\
V \in \mathcal{V}al & ::= & x \mid u_{\ell,\theta} \mid \mathsf{rec}f(x).M \mid tt \mid ff \mid () & \textit{values}
\end{array}
$$

Figure 1: Syntax

construct $(\ell \ltimes M)$ is used to restrict the access right of M by ℓ (this is similar to the "framed" expressions of [8], and to the "signed" expressions of [17]). This is a scoping construct: the current reading clearance is restored after termination of M. Our $(\mathsf{enable}\ \ell\ \mathsf{in}\ M)$ construct is slightly different from the one of [17, 23], where ℓ is restricted to be a singleton (our semantics is accordingly slightly more liberal). It is used to locally extend the read access right of M by ℓ. The test expression checks whether a given level is enabled by the current evaluation context. The local flow declaration $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ enables the policy F to be used while reducing M, usually for declassification purposes. (For more comments on the syntax, we refer to [1, 2, 8, 17].)

The syntax is given in Figure 1, where x and f are any variables, ℓ is any confidentiality level, and F is any flow policy. A *reference* is a memory location u to which a confidentiality level ℓ is assigned. For technical reasons, we also record the type θ (see Section 3 below) of the contents of the reference. We denote by $\mathsf{loc}(M)$ the set of decorated locations $u_{\ell,\theta}$ occurring in M. These references are regarded as providing the *inputs* of the expression M. We let $\mathsf{fv}(M)$ be the set of variables occurring free in M, and we denote by $\{x \mapsto V\}M$ the capture-avoiding susbtitution of V for the free occurrences of x in M, where $V \in \mathcal{V}al$. We may write $\mathsf{rec}f(x).M$ as $\lambda x M$ whenever $f \notin \mathsf{fv}(M)$. As usual, $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$ denotes $(\lambda x M N)$.

The reduction relation is a transition relation between configurations of the form (M, μ) where μ, the *memory* (or *heap*), is a mapping from a finite set $\mathsf{dom}(\mu)$ of references to values. The operation of updating the value of a reference in the memory is denoted, as usual, $\mu[u_{\ell,\theta} := V]$. We say that the name u is *fresh for* μ if $v_{\ell,\theta} \in \mathsf{dom}(\mu) \ \Rightarrow\ v \neq u$. In what follows we shall only consider *well-formed* configurations, that is pairs (M, μ) such that $\mathsf{loc}(M) \subseteq \mathsf{dom}(\mu)$ and for any $u_{\ell,\theta} \in \mathsf{dom}(\mu)$ we have $\mathsf{loc}(\mu(u_{\ell,\theta})) \subseteq \mathsf{dom}(\mu)$ (this property will be preserved by the operational semantics). As usual (see [25]), the operational semantics consists in reducing a *redex* (reducible expression) inside an *evaluation context*. Reducible expressions are given by the following grammar:

$$
\begin{array}{rcl}
R & := & (\mathsf{if}\ tt\ \mathsf{then}\ M\ \mathsf{else}\ N) \mid (\mathsf{if}\ ff\ \mathsf{then}\ M\ \mathsf{else}\ N) \mid (\mathsf{rec}f(x).MV) \\
& \mid & V\,;N \mid (\mathsf{ref}_{\ell,\theta}\,V) \mid (!\,u_{\ell,\theta}) \mid (u_{\ell,\theta} := V) \\
& \mid & (\ell \ltimes V) \mid (\mathsf{enable}\ \ell\ \mathsf{in}\ V) \mid (\mathsf{test}\ \ell\ \mathsf{then}\ M\ \mathsf{else}\ N) \\
& \mid & (\mathsf{flow}\ \ell\ \mathsf{in}\ V)
\end{array}
$$

$$
\begin{array}{rll}
\ell \vdash_G (\mathbf{E}[(\mathsf{if}\ tt\ \mathsf{then}\ M\ \mathsf{else}\ N)], \mu) \to (\mathbf{E}[M], \mu) & & \\
\ell \vdash_G (\mathbf{E}[(\mathsf{if}\ ff\ \mathsf{then}\ M\ \mathsf{else}\ N)], \mu) \to (\mathbf{E}[N], \mu) & & \\
\ell \vdash_G (\mathbf{E}[(\mathsf{rec} f(x).MV)], \mu) \to (\mathbf{E}[\{x \mapsto V\}\{f \mapsto \mathsf{rec} f(x).M\}M], \mu) & & \\
\ell \vdash_G (\mathbf{E}[V\ ; N], \mu) \to (\mathbf{E}[N], \mu) & & \\
\ell \vdash_G (\mathbf{E}[(\mathsf{ref}_{\ell',\theta}\, V)], \mu) \to (\mathbf{E}[u_{\ell',\theta}], \mu \cup \{u_{\ell',\theta} \mapsto V\}) & & u \text{ fresh for } \mu \\
\ell \vdash_G (\mathbf{E}[(!\, u_{\ell',\theta})], \mu) \to (\mathbf{E}[V], \mu) & & \mu(u_{\ell',\theta}) = V\ \&\ \\
 & & \ell' \preceq_G \lfloor \mathbf{E} \rfloor_\ell \\
\ell \vdash_G (\mathbf{E}[(u_{\ell',\theta} := V)], \mu) \to (\mathbf{E}[()], \mu[u_{\ell',\theta} := V]) & & \\
\ell \vdash_G (\mathbf{E}[(\ell' \ltimes V)], \mu) \to (\mathbf{E}[V], \mu) & & \\
\ell \vdash_G (\mathbf{E}[(\mathsf{enable}\ \ell'\ \mathsf{in}\ V)], \mu) \to (\mathbf{E}[V], \mu) & & \\
\ell \vdash_G (\mathbf{E}[(\mathsf{test}\ \ell'\ \mathsf{then}\ M\ \mathsf{else}\ N)], \mu) \to (\mathbf{E}[M], \mu) & & \ell' \preceq_G \lfloor \mathbf{E} \rfloor_\ell \\
\ell \vdash_G (\mathbf{E}[(\mathsf{test}\ \ell'\ \mathsf{then}\ M\ \mathsf{else}\ N)], \mu) \to (\mathbf{E}[N], \mu) & & \ell' \not\preceq_G \lfloor \mathbf{E} \rfloor_\ell \\
(\mathbf{E}[(\mathsf{flow}\ F\ \mathsf{in}\ V)], \mu) \to (\mathbf{E}[V], \mu) & &
\end{array}
$$

Figure 2: Reduction

Evaluation contexts are given by:

$$
\begin{array}{lll}
\mathbf{E} ::= [] \mid \mathbf{E}[\mathbf{F}] & & \textit{evaluation contexts} \\
\mathbf{F} := (\mathsf{if}\ []\ \mathsf{then}\ M\ \mathsf{else}\ N) \mid ([]N) \mid (V[]) & & \textit{frames} \\
\quad \mid [] ; N \mid (\mathsf{ref}_{\ell,\theta}[]) \mid (![]) \mid ([] := N) \mid (V := []) & & \\
\quad \mid (\ell \ltimes []) \mid (\mathsf{enable}\ \ell\ \mathsf{in}\ []) \mid (\mathsf{flow}\ F\ \mathsf{in}\ []) & &
\end{array}
$$

The operational semantics of an expression depends upon a given global flow policy G and a default confidentiality level ℓ, which represents the access right assigned to the evaluated expression – as if we had to evaluate $(\bot \ltimes (\mathsf{enable}\ \ell\ \mathsf{in}\ M))$. Then the statements defining the operational semantics have the form

$$\ell \vdash_G (M, \mu) \to (M', \mu')$$

Given a global flow policy G, for any confidentiality level ℓ representing the current access level, we define the level granted by the evaluation context $\mathbf{E}$, denoted $\lfloor \mathbf{E} \rfloor_\ell$, as follows:

$$
\begin{array}{rl}
\lfloor [] \rfloor_\ell = \ell & \\
\lfloor \mathbf{E}[\mathbf{F}] \rfloor_\ell = & \begin{cases}
(\lfloor \mathbf{E} \rfloor_\ell) \curlywedge_G \ell' & \text{if } \mathbf{F} = (\ell' \ltimes []) \\
(\lfloor \mathbf{E} \rfloor_\ell) \curlyvee_G \ell' & \text{if } \mathbf{F} = (\mathsf{enable}\ \ell'\ \mathsf{in}\ []) \\
\lfloor \mathbf{E} \rfloor_\ell & \text{otherwise}
\end{cases}
\end{array}
$$

Computing $\lfloor \mathbf{E} \rfloor_\ell$ is a form of "stack inspection," see [8, 17, 23]. The reduction rules are given in Figure 2. They are fairly standard, as regards the functional and imperative fragment of the language. One may observe that $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ behaves exactly as M, and that the semantics of the constructs for managing access rights are as one might expect, given the definition of $\lfloor \mathbf{E} \rfloor_\ell$ above. We

denote by $\xrightarrow{*}$ the reflexive and transitive closure of the reduction relation. More precisely, we define:

$$\frac{}{\ell \vdash_G (M,\mu) \xrightarrow{*} (M,\mu)} \qquad \frac{\ell \vdash_G (M,\mu) \xrightarrow{*} (M'',\mu'') \quad \ell \vdash_G (M'',\mu'') \to (M',\mu')}{\ell \vdash_G (M,\mu) \xrightarrow{*} (M',\mu')}$$

An expression is said to *converge* if, regardless of the memory, its evaluation terminates on a value, that is:

$$M\Downarrow \quad \Leftrightarrow_{\text{def}} \quad \forall \mu\, \exists V \in \mathcal{V}al\, \exists \mu'.\ (M,\mu) \xrightarrow{*} (V,\mu')$$

One can see that a constraint that could block the reduction is the dynamic checking of read access rights, that is the condition $\ell' \preceq_G \lfloor \mathbf{E} \rfloor_\ell$ when reading a reference $u_{\ell',\theta}$ in the memory, with ℓ as the given reading clearance. (Since we are only considering well-formed configurations, the value $\mu(u_{\ell',\theta})$ is always defined.) Indeed for instance the expression (omitting the types attached to references)

$$(\mathsf{flow}\ H \prec L\ \mathsf{in}\ v_L := !\, u_H)$$

is blocked if the current access right is $\{L\}$, and can only proceed if this right is at least $\{H\}$. This indicates that, using $(\mathsf{flow}\ F\ \mathsf{in}\ M)$, one can only declassify information that one has the right to read. This is because the local flow policy declarations do not interfere with access control, i.e. they do not play any role in the definition of $\lfloor \mathbf{E} \rfloor_\ell$. More generally, to let information flow, such as in $v_{\ell'} := !\, u_\ell$, a program must have the right to read it.

One can easily check the usual property (see [25]) that, given a pair (ℓ, G), a closed expression is either a value, or a reducible expression, or is a faulty expression, either for typing reasons, or because it does not have the appropriate access rights, that is:

Lemma and Definition 2.2. *Let M be a closed expression. Then, for any ℓ and G, either $M \in \mathcal{V}al$, or for any μ there exist F, M' and μ' such that $\ell \vdash_G (M,\mu) \underset{F}{\longrightarrow} (M',\mu')$, or M is (ℓ,G)-*faulty*, that is:*

(i) $M = \mathbf{E}[(\mathsf{if}\ V\ \mathsf{then}\ N\ \mathsf{else}\ N')]$ *and* $V \notin \{tt, ff\}$, *or*

(ii) $M = \mathbf{E}[(VV')]$ *and V is not a functional value* $\mathsf{rec} f(x).M'$, *or*

(iii) $M = \mathbf{E}[(!\, V)]$ *or* $M = \mathbf{E}[(V := V')]$ *where V is not a reference $u_{\ell',\theta}$, or*

(iv) $M = \mathbf{E}[(!\, u_{\ell',\theta})]$ *with* $\ell' \not\preceq_G \lfloor \mathbf{E} \rfloor_\ell$.

Our main aim in this paper is to show that we can design a type system that guarantees both secure information flow, as in [1], and, as in [2, 17], the fact that a well-typed expression is never stuck, and therefore that the run-time checking of the access rights is useless for such expressions.

3 The type and effect system

Our type system elaborates on the one of [1], and, as such, is actually a *type and effect system* [11]. This is consistent with our "*state-oriented*" approach – as opposed to the "value-oriented" approach of [8, 16, 17, 23] for instance – where

only the access to the "information containers", that is, to the references in the memory, is protected by access rights. In particular, a value is by itself neither "secret" nor "public," and the types do not need to be multiplied by the set of confidentiality levels. Then the types are

$$\tau,\ \sigma,\ \theta \ldots\ ::=\ t\ \mid\ \mathsf{bool}\ \mid\ \mathsf{unit}\ \mid\ \theta\,\mathsf{ref}_\ell\ \mid\ (\tau \xrightarrow[\ell,F]{s} \sigma)$$

where t is any type variable and s is any "security effect" – see below. Notice that a reference type $\theta\,\mathsf{ref}_\ell$ records the type θ of values the reference contains, as well as the "region" ℓ where it is created, which is the confidentiality level at which the reference is classified. Since a functional value wraps a possibly effectful computation, its type records this *latent effect* [11], which is the effect the function may have when applied to an argument. It also records the "latent reading clearance" ℓ and the "latent flow policy" F, which are assumed to hold when the function is called. The judgements of the type and effect system have the form

$$\ell; F; \Gamma \vdash_G M : s, \tau$$

where Γ is a typing context, assigning types to variables, and s is a security effect, that is a triple (ℓ_0, ℓ_1, ℓ_2) of confidentiality levels. The intuition is:

- ℓ is the current read access right that is in force when reducing M;
- F is the current flow policy, while G is the given global flow policy;
- ℓ_0, also denoted by $s.\mathsf{c}$, is the *confidentiality level* of M. This is an upper bound (up to the current flow relation) of the confidentiality levels of the references the expression M reads that may influence its resulting *value*;
- ℓ_1, also denoted $s.\mathsf{w}$, is the *writing effect*, that is a lower bound (w.r.t. the relation $\preceq$) of the level of references that the expression M may update;
- ℓ_2, also denoted $s.\mathsf{t}$, is an upper bound (w.r.t. the current flow relation) of the levels of the references the expression M reads that may influence its *termination*. We call this the *termination effect* of the expression.

In the following we shall denote $s.\mathsf{c} \curlyvee_{F \cup G} s.\mathsf{t}$ by $s.\mathsf{r}$, assuming that F and G are understood from the context. There is actually an implicit parameter in the type system, which is a set $\mathcal{T}$ of expressions that is used in the typing of conditional branching. The single property that we will assume about this set in our proof of type soundness is that it only contains strongly converging expressions, that is:

$$M \in \mathcal{T}\ \Rightarrow\ M{\Downarrow}$$

According to the intuition above, the security effects $s = (c, w, t)$ are ordered componentwise, in a covariant manner as regards the confidentiality level c and the termination effect t, and in a contravariant way as regard the writing effect w. Then we abusively denote by $\bot$ and $\top$ the triples $(\bot, \top, \bot)$ and $(\top, \bot, \top)$ respectively. In the typing rules for compound expressions, we will use the join operation on security effects:

$$s \curlyvee_F s' \ =_{\mathrm{def}}\ (s.\mathsf{c} \curlyvee_F s'.\mathsf{c}, s.\mathsf{w} \cup s'.\mathsf{w}, s.\mathsf{t} \curlyvee_F s'.\mathsf{t})$$

as well as the following convention:

$$\frac{}{\ell; F; \Gamma \vdash_G u_{\ell',\theta} : \bot, \theta\,\mathsf{ref}_{\ell'}}\ (\textsc{Loc}) \qquad \frac{}{\ell; F; \Gamma, x : \tau \vdash_G x : \bot, \tau}\ (\textsc{Var})$$

$$\frac{\ell; F; \Gamma, x : \tau, f : (\tau \xrightarrow[\ell,F]{s} \sigma) \vdash_G M : s, \sigma}{\ell'; F'; \Gamma \vdash_G \mathsf{rec} f(x).M : \bot, (\tau \xrightarrow[\ell,F]{s} \sigma)}\ (\textsc{Fun}) \qquad \frac{}{\ell; F; \Gamma \vdash_G () : \bot, \mathsf{unit}}\ (\textsc{Nil})$$

$$\frac{}{\ell; F; \Gamma \vdash_G tt : \bot, \mathsf{bool}}\ (\textsc{BoolT}) \qquad \frac{}{\ell; F; \Gamma \vdash_G ff : \bot, \mathsf{bool}}\ (\textsc{BoolF})$$

$$\frac{\ell; F; \Gamma \vdash_G M : s, \mathsf{bool} \quad \ell; F; \Gamma \vdash_G N_i : s_i, \tau \quad s.\mathsf{r} \preceq_{F \cup G} s_0.\mathsf{w} \cup s_1.\mathsf{w}}{\ell; F; \Gamma \vdash_G (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) : s \curlyvee s_0 \curlyvee s_1 \curlyvee (\bot, \top, t), \tau}\ (\textsc{Cond})$$

where

$$t \;=\; \begin{cases} \bot & \text{if } N_0, N_1 \in \mathcal{T} \\ s.\mathsf{c} & \text{otherwise} \end{cases}$$

$$\frac{\begin{array}{ll} \ell; F; \Gamma \vdash_G M : s, \tau \xrightarrow[\ell',F']{s'} \sigma & \ell' \preceq_G \ell \quad s.\mathsf{t} \preceq_{F \cup G} s''.\mathsf{w} \\ \ell; F; \Gamma \vdash_G N : s'', \tau & s.\mathsf{r} \curlyvee s''.\mathsf{r} \preceq_{F \cup G} s'.\mathsf{w} \end{array}}{\ell; F \cup F'; \Gamma \vdash_G (MN) : s \curlyvee s' \curlyvee s'' \curlyvee (\bot, \top, s.\mathsf{c} \curlyvee s''.\mathsf{c}), \sigma}\ (\textsc{App})$$

$$\frac{\ell; F; \Gamma \vdash_G M : s, \tau \quad \ell; F; \Gamma \vdash_G N : s', \sigma \quad s.\mathsf{t} \preceq_{F \cup G} s'.\mathsf{w}}{\ell; F; \Gamma \vdash_G M\,; N : (\bot, s.\mathsf{w}, s.\mathsf{t}) \curlyvee s', \sigma}\ (\textsc{Seq})$$

$$\frac{\ell; F; \Gamma \vdash_G M : s, \theta \quad s.\mathsf{r} \preceq_{F \cup G} \ell'}{\ell; F; \Gamma \vdash_G (\mathsf{ref}_{\ell',\theta}\, M) : (\bot, s.\mathsf{w}, s.\mathsf{t}), \theta\,\mathsf{ref}_{\ell'}}\ (\textsc{Ref}) \qquad \frac{\ell; F; \Gamma \vdash_G M : s, \theta\,\mathsf{ref}_{\ell'} \quad \ell' \preceq_G \ell}{\ell; F; \Gamma \vdash_G (!\,M) : s \curlyvee (\ell', \top, \bot), \theta}\ (\textsc{Deref})$$

$$\frac{\begin{array}{ll} \ell; F; \Gamma \vdash_G M : s, \theta\,\mathsf{ref}_{\ell'} & s.\mathsf{t} \preceq_{F \cup G} s'.\mathsf{w} \\ \ell; F; \Gamma \vdash_G N : s', \theta & s.\mathsf{r} \curlyvee s'.\mathsf{r} \preceq_{F \cup G} \ell' \end{array}}{\ell; F; \Gamma \vdash_G (M := N) : (\bot, s.\mathsf{w} \cup s'.\mathsf{w} \cup \ell', s.\mathsf{t} \curlyvee s'.\mathsf{t}), \mathsf{unit}}\ (\textsc{Assign})$$

$$\frac{\ell \curlywedge_G \ell'; F; \Gamma \vdash_G M : s, \tau}{\ell; F; \Gamma \vdash_G (\ell' \ltimes M) : s, \tau}\ (\textsc{Restric}) \qquad \frac{\ell \curlyvee_G \ell'; F; \Gamma \vdash_G M : s, \tau}{\ell; F; \Gamma \vdash_G (\mathsf{enable}\ \ell'\ \mathsf{in}\ M) : s, \tau}\ (\textsc{Enable})$$

$$\frac{\ell'; F; \Gamma \vdash_G M : s, \tau \quad \ell; F; \Gamma \vdash_G N : s', \tau}{\ell; F; \Gamma \vdash_G (\mathsf{test}\ \ell'\ \mathsf{then}\ M\ \mathsf{else}\ N) : s \curlyvee s', \tau}\ (\textsc{Test})$$

$$\frac{\ell; F \cup F'; \Gamma \vdash_G M : s, \tau \quad s.\mathsf{c} \preceq_{F \cup F' \cup G} c \quad s.\mathsf{t} \preceq_{F \cup F' \cup G} t}{\ell; F; \Gamma \vdash_G (\mathsf{flow}\ F'\ \mathsf{in}\ M) : (c, s.\mathsf{w}, t), \tau}\ (\textsc{Flow})$$

Figure 3: The Type and Effect System

Convention. *In the type system, when the security effects occurring in the context of a judgement* $\ell; F; \Gamma \vdash_G M : s, \tau$ *involve the join operation* $\curlyvee$*, it is*

assumed that the join is taken w.r.t. $F \cup G$, i.e. it is $\curlyvee_{F \cup G}$. We recall that by $s.\mathsf{r}$ we mean $s.\mathsf{c} \curlyvee_{F \cup G} s.\mathsf{t}$.

The typing system is given in Figure 3. Notice that this system is syntax-directed: there is exactly one rule per construction of the language. In particular, there is no subtyping rule. As an example, one can see for instance that the expression

$$(\mathsf{test}\ \{p\}\ \mathsf{then}\ (\mathsf{flow}\ p \prec q\ \mathsf{in}\ v_{\{q\}} := !\, u_{\{p\}})\ \mathsf{else}\ ())$$

(similar to the one given in the Introduction) is typable. One should also notice that our typing rules allow the programmer to make a dynamic use of the test construct, as in

$$\begin{array}{l}\mathsf{let}\ x = \lambda y(\mathsf{test}\ \ell\ \mathsf{then}\ v_\ell := !\, u_\ell\ \mathsf{else}\ ())\ \mathsf{in}\\ (\mathsf{if}\ M\ \mathsf{then}\ (\mathsf{enable}\ \ell\ \mathsf{in}\ x())\ \mathsf{else}\ x())\end{array}$$

Compared to the system of [1], which does not involve constructs for managing access control, the main difference is in the (Deref) rule, where we have the constraint that the level of the reference that is read should be less than the access level granted by the context. Notice that this constraint only involves the global flow policy G, not the local one F. This is the way to ensure that declassification does not modify the access rights. There is a similar constraint in the rule for typing application, where the access level required by the (body of the) function should indeed be granted by the context. It is easy to see that typing enjoys a "weakening" property, asserting that if an expression is typable in the context of some access right ℓ, then it is also typable in the context of a more permissive reading clearance:

Lemma 3.1. *If $\ell; F; \Gamma \vdash_G M : s, \tau$ and $\ell \preceq_G \ell'$ then $\ell'; F; \Gamma \vdash_G M : s, \tau$.*

Proof: by induction on the inference of the typing judgemnent. ❑

Similarly, one could show that relaxing (that is, extending) the global or local flow policy does not affect typability.

4 Type safety

The proof of type safety follows the usual steps [25]: we prove the Subject Reduction property, showing that typing is preserved along reductions, while the effects are decreasing; then we prove that faulty expressions are not typable. This, together with Lemma 2.2, will entail type safety. In the proof of the Subject Reduction property we use the following observation:

Remark 4.1. *For any value $V \in \mathcal{V}al$, if V is typable with type τ in the context Γ, then for any ℓ, F and G we have $\ell; F; \Gamma \vdash_G V : \bot, \tau$.*

We shall also need the following lemma, where we use the flow policy $\lceil \mathbf{E} \rceil$ granted by the evaluation context $\mathbf{E}$, which is defined as follows:

$$\lceil [\,] \rceil = \emptyset$$

$$\lceil \mathbf{E}[\mathbf{F}] \rceil = \begin{cases} \lceil \mathbf{E} \rceil \cup F & \text{if } \mathbf{F} = (\mathsf{flow}\ F\ \mathsf{in}\ [\,]) \\ \lceil \mathbf{E} \rceil & \text{otherwise} \end{cases}$$

LEMMA 4.2. *If* $\ell; F; \Gamma \vdash_G \mathbf{E}[M] : s, \tau$, *then there exist* s_0 *and* σ *such that* $\lfloor \mathbf{E} \rfloor_\ell; F \cup \lceil \mathbf{E} \rceil; \Gamma \vdash_G M : s_0, \sigma$, *and if* $\lfloor \mathbf{E} \rfloor_\ell; F \cup \lceil \mathbf{E} \rceil; \Gamma \vdash_G N : s_1, \sigma$ *with* $s_1 \preceq_G s_0$ *then* $\ell; F; \Gamma \vdash_G \mathbf{E}[N] : s', \tau$ *for some* s' *such that* $s' \preceq_G s$.

PROOF: by induction on $\mathbf{E}$. ❑

PROPOSITION (SUBJECT REDUCTION) 4.3. *If* $\ell; F; \Gamma \vdash_G M : s, \tau$ *and* $\ell \vdash_G (M, \mu) \rightarrow (M', \mu')$ *with* $u_{\ell,\theta} \in \mathsf{dom}(\mu) \Rightarrow \ell; F; \Gamma \vdash_G \mu(u_{\ell,\theta}) : \bot, \theta$ *then* $\ell; F; \Gamma \vdash_G M' : s', \tau$ *for some* s' *such that* $s' \preceq_G s$.

PROOF SKETCH: we have $M = \mathbf{E}[R]$ and $M' = \mathbf{E}[N]$ with $(R, \mu) \rightarrow (N, \mu')$, where R is a redex. We proceed by induction on the context $\mathbf{E}$. In the case where $\mathbf{E} = []$, the proof is as usual for the functional and imperative part of the language (we need some auxiliary properties, see [25]). Let us just examine the cases where a security construct is involved. In the cases of

$$\begin{aligned} \ell \vdash_G ((\ell' \ltimes V), \mu) &\rightarrow (V, \mu) \\ \ell \vdash_G ((\mathsf{enable}\ \ell'\ \mathsf{in}\ V), \mu) &\rightarrow (V, \mu) \\ \ell \vdash_G ((\mathsf{flow}\ F\ \mathsf{in}\ V), \mu) &\rightarrow (V, \mu) \end{aligned}$$

we use the Remark 4.1 above. The cases

$$\begin{aligned} \ell \vdash_G ((\mathsf{test}\ \ell'\ \mathsf{then}\ M'\ \mathsf{else}\ M''), \mu) &\rightarrow (M', \mu) \quad \text{with } \ell' \preceq_G \ell \\ \ell \vdash_G ((\mathsf{test}\ \ell'\ \mathsf{then}\ M''\ \mathsf{else}\ M'), \mu) &\rightarrow (M', \mu) \quad \text{with } \ell' \not\preceq_G \ell \end{aligned}$$

are immediate (in the first case we use Lemma 3.1). Now if $\mathbf{E} = \mathbf{E}'[\mathbf{F}]$ we use the Lemma 4.2 above. ❑

Now we show that the faulty expressions, as defined in Lemma and Definition 2.2, are not typable.

LEMMA 4.4. *The* (ℓ, G)*-faulty expressions are not typable in the context of access right* ℓ *and global flow policy* G.

PROOF: let $M = \mathbf{E}[(\mathbf{!}\, u_{\ell',\theta})]$ with $\ell' \not\preceq_G \lfloor \mathbf{E} \rfloor_\ell$, and assume that $\ell; F, \Gamma \vdash_G M : s, \tau$. Then by Lemma 4.2 one would have $\lfloor \mathbf{E} \rfloor_\ell; F \cup \lceil \mathbf{E} \rceil; \Gamma \vdash_G (\mathbf{!}\, u_{\ell',\theta}) : s', \sigma$ for some s' and σ, but this is only possible, by the (DEREF) rule, if $\ell' \preceq_G \lfloor \mathbf{E} \rfloor_\ell$, a contradiction. The other cases are standard. ❑

An immediate consequence of these results and Lemma 2.2 is:

THEOREM (TYPE SAFETY) 4.5. *Let* M *be a typable closed expression, with* $\ell; F; \Gamma \vdash_G M : s, \tau$, *and let* μ *be such that* $u_{\ell,\theta} \in \mathsf{dom}(\mu) \Rightarrow \ell; F; \Gamma \vdash_G \mu(u_{\ell,\theta}) : \bot, \theta$. *Then either the reduction of* (M, μ) *with respect to* (ℓ, G) *does not terminate, or there exist a value* $V \in \mathcal{V}al$ *and a memory* μ' *such that* $\ell \vdash_G (M, \mu) \stackrel{*}{\rightarrow} (V, \mu')$ *with* $\ell; F; \Gamma \vdash_G V : \bot, \tau$.

In particular, this shows that the dynamic checking of the reading clearance (by means of a "stack inspection" mechanism) is actually not needed regarding a typable program, which never attempts to access a reference for which it would not have the appropriate access right.

5 Secure information flow

In [1], the authors have proposed a generalization of the usual non-interference property that allows one to deal with declassification. The idea is to define a program as secure if it satisfies a "local non-interference" property, called the *non-disclosure policy*, which roughly states that the program is, at each step of its execution, non-interfering with respect to the current flow policy, that is the global flow policy extended by the one granted by the evaluation context. Technically, a program will be considered as secure if it is bisimilar to itself. As usual, this property relies on preserving the "low equality" of memory. Roughly speaking, two memories are equal up to level ℓ if they assign the same value to every location with security level lower than ℓ, with respect to a given flow policy. In order to deal with reference creation, we only compare memories on the domain of references they share – then the "low equality" of memory is actually not an equivalence (it is not transitive). Nevertheless, we keep the standard terminology. The "low equality" of memories, with respect to a current flow policy F, and to a security level ℓ regarded as "low," is thus defined:

$$\mu \doteq^{F,\ell} \nu \quad \Leftrightarrow_{\text{def}} \quad \forall u_{\ell',\theta} \in \mathsf{dom}(\mu) \cap \mathsf{dom}(\nu).\ \ell' \preceq_F \ell \ \Rightarrow\ \mu(u_{\ell',\theta}) = \nu(u_{\ell',\theta})$$

From an information flow point of view, the notion of a secure program actually depends on the default access right. For instance, the assignment $v_{\ell'} := !u_\ell$ where $\ell \not\preceq_G \ell'$, which is usually taken as a typical example of an unsecure program, is indeed secure (in the sense of [6]) in the context of a default access level ℓ'' such that $\ell \not\preceq_G \ell''$ (but in that case this program attempts a confidentiality violation, as regards access control). Then our definition of secure programs is parameterized by a default access level, and, as in [1], by a global flow policy.

Definition (Bisimulation) 5.1. *A (ℓ, G, ℓ')-*bisimulation *is a symmetric relation $\mathcal{R}$ on expressions such that if $M\,\mathcal{R}\,N$ and $\ell \vdash_G (M,\mu) \to (M',\mu')$ with $M = \mathbf{E}[R]$ where R is a redex, and if ν is such that $\mu \doteq^{G \cup \lceil \mathbf{E} \rceil, \ell'} \nu$ and $u_{\ell'',\theta} \in \mathsf{dom}(\mu' - \mu)$ implies that u is fresh for ν, then there exist N' and ν' such that $\ell \vdash_G (N,\nu) \stackrel{*}{\to} (N',\nu')$ with $M'\,\mathcal{R}\,N'$ and $\mu' \doteq^{G,\ell'} \nu'$.*

Remarks and Notation 5.2.

(i) *For any ℓ, G and ℓ' there exists a (ℓ, G, ℓ')-bisimulation, like for instance the set $\mathcal{V}al \times \mathcal{V}al$ of pairs of values.*

(ii) *The union of a family of (ℓ, G, ℓ')-bisimulations is a (ℓ, G, ℓ')-bisimulation. Consequently, there is a largest (ℓ, G, ℓ')-bisimulation, which we denote $\mathrel{\sqcup}^{\ell,G,\ell'}$. This is the union of all such bisimulations.*

One should observe that the relation $\mathrel{\sqcup}^{\ell,G,\ell'}$ is not reflexive. Indeed, a process which is not bisimilar to itself, like $v_{\ell'} := !\,u_{\ell''}$ where $\ell \not\preceq_G \ell'' \preceq_G \ell$, is not secure. As in [20], our definition states that a program is secure, with respect to a default access level and a given global flow policy, if it is bisimilar to itself:

Definition (The Non-Disclosure Policy) 5.3. *A process P satisfies the* non-disclosure policy *(or is* secure *from the confidentiality point of view) with respect to the default access level ℓ and the global flow policy G if it satisfies $P \mathrel{\sqcup}^{\ell,G,\ell'} P$ for all ℓ'. We then write $P \in \mathcal{ND}(\ell, G)$.*

For explanations and examples regarding the non-disclosure policy, we refer to [1]. Our second main result is that the type system guarantees secure information flow, whatever the default access right is:

THEOREM (SOUNDNESS). *If M is typable in the context of the access level ℓ, a global flow policy G and a local policy F, that is if for some Γ, s and τ we have $\ell; F; \Gamma \vdash_G M : s, \tau$, then M satisfies the non-disclosure policy with respect to $F \cup G$, that is $M \in \mathcal{ND}(\ell', F \cup G)$, for all ℓ'.*

The proof of this result is very similar to the one of Type Soundness in the revised version of [1]. As usual with bisimulation proofs, one has to find an appropriate candidate relation, that contains the pairs (M, M) of typable expressions, and which is closed with respect to the co-inductive property. The constructs for managing access control do not add much complexity to this proof.

6 Conclusion

We have shown a way of integrating access control and information flow control in the setting of a high-level programming language, involving a declassification construct. Our "state-oriented" approach, that we share with [2], differs from the "value-oriented" approach (that one has to adopt when dealing with purely functional languages) that is followed in [8, 17, 23] to deal with stack inspection, and [16, 22] as regards information flow control (see also [18] for further references). We think that assuming that confidentiality levels are assigned to "information containers" is more in line with the usual way of dealing with confidentiality than assigning security levels to values, like boolean *tt* and *ff*, integers or functions. In this way, the safety property guaranteed by access control is quite simple and natural, and this also provides a natural restriction on the use of declassification, and, more generally, information flow.

References

1. A. ALMEIDA MATOS, G. BOUDOL, *On declassification and the non-disclosure policy,* CSFW'05 (2005) 226-240. Revised version accepted for publication in the J. of Computer Security, available from the authors web page.
2. A. BANERJEE, D. A. NAUMANN, *Stack-based access control for secure information flow,* J. of Functional Programming Vol. 15, special issue on Language-Based Security (2005) 131-177.
3. G. BOUDOL, *On typing information flow,* Intern. Coll. on Theoretical Aspects of Computing, Lecture Notes in Comput. Sci. 3722 (2005) 366-380.
4. N. BROBERG, D. SANDS, *Flow locks: towards a core calculus for dynamic flow policies,* ESOP'06, Lecture Notes in Comput. Sci. 3924 (2006) 180-196.
5. S. CHONG, A. C. MYERS, *Security policies for downgrading,* 11th ACM Conf. on Computer and Communications Security (2004).
6. E. COHEN, *Information transmission in computational systems,* 6th ACM Symp. on Operating Systems Principles (1977) 133-139.
7. D. E. DENNING, *A lattice model of secure information flow,* CACM Vol. 19 No. 5 (1976) 236-243.

8. C. Fournet, A. Gordon, *Stack inspection: theory and variants,* POPL'02 (2002) 307-318.
9. J. A. Goguen, J. Meseguer, *Security policies and security models,* IEEE Symp. on Security and Privacy (1982) 11-20.
10. B. W. Lampson, *A note on the confinement problem,* CACM Vol. 16 No. 10 (1973) 613-615.
11. J. M. Lucassen, D. K. Gifford, *Polymorphic effect systems,* POPL'88 (1988) 47-57.
12. P. Li, S. Zdancewic, *Downgrading policies and relaxed noninterference,* POPL'05 (2005) 158-170.
13. A. Myers, *JFlow: practical mostly-static information flow control,* POPL'99 (1999).
14. A. C. Myers, B. Liskov, *A decentralized model for information flow control,* ACM Symp. on Operating Systems Principles (1997) 129-142.
15. A. C. Myers, A. Sabelfeld, S. Zdancewic, *Enforcing robust declassification and qualified robustness,* J. of Computer Security Vol. 14, No. 2 (2006) 157-196.
16. F. Pottier, S. Conchon, *Information flow inference for free,* ICFP'00 (2000) 46-57.
17. F. Pottier, C. Skalka, S. Smith, *A systematic approach to static access control,* ACM TOPLAS Vol. 27, No. 2 (2005) 344-382.
18. A. Sabelfeld, A. C. Myers, *Language-based information-flow security,* IEEE J. on Selected Areas in Communications Vol. 21 No. 1 (2003) 5-19.
19. A. Sabelfeld, A. C. Myers, *A model for delimited information release,* Intern. Symp. on Software Security, Lecture Notes in Comput. Sci. to appear (2003).
20. A. Sabelfeld, D. Sands, *Probabilistic noninterference for multi-threaded programs,* CSFW'00 (2000).
21. A. Sabelfeld, D. Sands, *Dimensions and principles of declassification,* CSFW'05 (2005) 255-269.
22. V. Simonet, *The Flow Caml system: documentation and user's manual,* INRIA Tech. Rep. 0282 (2003).
23. C. Skalka, S. Smith, *Static enforcement of security with types,* ICFP'00 (2000) 34-45.
24. D. Volpano, G. Smith, C. Irvine, *A sound type system for secure flow analysis,* J. of Computer Security, Vol. 4, No 3 (1996) 167-187.
25. A. Wright, M. Felleisen, *A syntactic approach to type soundness,* Information and Computation Vol. 115 No. 1 (1994) 38-94.
26. S. Zdancewic, *Challenges for information-flow security,* PLID'04 (2004).