Informative Types and Effects for Hybrid Migration Control

Ana Almeida Matos and Jan Cederquist

Instituto de Telecomunicações (SQIG) and Instituto Superior Técnico,
Lisbon, Portugal
{ana.matos,jan.cederquist}@ist.utl.pt

Abstract Flow policy confinement is a property of programs whose declassifications respect the allowed flow policy of the context in which they execute. In a distributed setting where computation domains enforce different allowed flow policies, code migration between domains implies dynamic changes to the relevant allowed policy. Furthermore, when programs consist of more than one thread running concurrently, the same program might need to comply to more than one allowed flow policy simultaneously. In this scenario, confinement can be enforced as a migration control mechanism. In the present work we compare three type-based enforcement mechanisms for confinement, regarding precision and efficiency of the analysis. In particular, we propose an efficient hybrid mechanism based on statically annotating programs with the declassification effect of migrating code. This is done by means of an informative type and effect pre-processing of the program, and is used for supporting runtime decisions.

1 Introduction

Research in language based security has placed a lot of attention on the study of information flow properties and enforcement mechanisms [1]. Information flow security regards the control of how dependencies between information of different security levels can lead to information leakage during program execution. Information flow properties range in strictness from pure absence of information leaks, classically known as non-interference [2], to more flexible properties that allow for *declassification* to take place in a controlled manner [3].

Separating the problems of enabling and of controlling flexible information flow policies paves the way to a modular composition of security properties that can be studied independently. Here we consider a distributed setting with runtime remote thread creation, and the problem of ensuring that declassifications that are performed by mobile code comply to the flow policy that is allowed at the computation domain where they are performed. We refer to this property as *flow policy confinement*, and treat it as a migration control problem [4,5].

An illustrative scenario could be that of a set of personal mobile appliances, such as smartphones. Due to their inter-connectivity (web, Bluetooth), they form networks of highly responsive computing devices with relatively limited resources, and that handle sensitive information (personal location, contacts, passwords). This combination demands for scalable and efficient mechanisms for ensuring privacy in a distributed setting with code mobility. From an abstract perspective, each device forms a *computation domain* with specific capabilities and restrictions, and in particular information flow policies for protecting data and other computing *threads* that are running concurrently in the same domain. We refer to these policies as the *allowed flow policy* of the domain. Flow policy confinement ensures that domains do not execute code that might perform declassifications that break their own allowed policies.

Let us consider, for example, an application for supporting two users (Alice and Bob) in choosing the best meeting point and path for reaching each other by means of public transportation. In order to produce advice that takes into account the current context (recent user locations, traffic conditions, weather) threads containing code for building updated travel maps are downloaded by Alice and Bob during runtime (their travel). The recommended path and meeting point can be improved by deducing the users' personal preferences from data that it collects from the mobile devices (e.g. content of stored images, file types). Users might, however, have privacy restrictions regarding that data, in the form of allowed flow policies that the downloaded threads must comply to. The following naive program creates a thread for gathering data that helps select the meeting point. Since the meeting point will necessarily be revealed to Bob, this part of the program should only allowed to run if it respects which private information Alice accepts to leak to Bob.

```
newthread {                      // Creates thread at Alice's device
   ref zoo=0; ref bookstore=0;   // to choose between zoo or bookstore
   allowed                       // If allowed by Alice's policy
      (L_IMGS < L_BOB /\         // to leak image contents
       L_FILES < L_BOB)          // and file types to Bob
      flow (L_IMGS < L_BOB /\    //   Declares a declassification
            L_FILES < L_BOB)     //   with same policy
         processImgs(zoo);       //     weighs images with animals
         searchFiles(bookstore); //     weighs e-book files
         if (zoo > bookstore)    //     inspects sensitive data...
            meetAt(ZOO);         //       and influences meeting point
            meetAt(BOOKSTORE);
   meetAt(random);               // If not allowed, uses other criteria
} at D_ALICE
```

As the above code is deployed, device `D_ALICE` must decide whether it is safe to execute the thread or not. Clearly, the decision must be taken quickly so as to not disrupt the purpose of the application. Ultimately, it is based on an analysis of the code, giving special attention to the points where declassifications occur.

This paper addresses the technical problem of how to build suitable enforcement mechanisms that enable domains to check incoming code against their own allowed flow policies. Previous work [6] introduces, as a proof-of-concept, a runtime migration control mechanism for enforcing confinement that lacks precision and efficiency. In this paper we look closely at the problem of the overhead that is implied by using types and effects for checking programs during runtime.

We study three type and effect-based mechanisms for enforcing confinement that place different weight over static and run time: First, we present a purely static type and effect system. Second, we increase its precision by letting most of the control be done dynamically, at the level of the operational semantics. To this end, the migration instruction is conditioned by a type check by means of a standard type and effect checking system. Third, we provide a mechanism for removing the runtime weight of typing migrating programs. It consists in statically annotating programs with information about the declassifying behavior of migrating threads, in the form of a *declassification effect*, and using it to support efficient runtime checks.

This work is formulated over an expressive distributed higher-order imperative lambda-calculus with remote thread creation. This language feature implies that programs might need to comply to more than one dynamically changing allowed flow policy simultaneously. The main contributions are:

1. A purely static type and effect system for enforcing flow policy confinement.
2. A type and effect system for checking migrating threads at runtime, that is more precise than the one in point 1.
3. A static-time informative pre-processing type and effect system for annotating programs with a declassification effect, for a more efficient and precise mechanism than the one in point 2.

We start by presenting the security setting (Section 2) and language (Section 3). The formal security property of Flow Policy Confinement (Section4) follows. Then, we study three type and effect-based enforcement mechanisms (Section 5) and draw conclusions regarding their efficiency and precision. Finally we discuss related work (Section 6) and conclude (Section 7). An extended version of this article (available from the authors) presents the detailed proofs.

2 Security Setting

The study of confidentiality traditionally relies on a lattice of security levels [7], corresponding to security clearances, that is associated to information containers in the programming language. The idea is that information pertaining to references labeled with l_2 can be legally transferred to references labeled with l_1 only if l_1 is at least as confidential as l_2. In this paper we do not deal explicitly with security levels, but instead with *flow policies* that define how information should be allowed to flow between security levels. Formally, flow policies can be seen as downward closure operators over a basic lattice of security levels [8].

Flow policies $A, F \in \boldsymbol{Flo}$ can be ordered according to their permissiveness by means of a *permissiveness relation* $\preccurlyeq$, where $F_1 \preccurlyeq F_2$ means that F_1 is at least as permissive as F_2. We assume that flow policies form a lattice that supports a pseudo-subtraction operation $\langle \boldsymbol{Flo}, \preccurlyeq, \curlywedge, \curlyvee, \mho, \mathit{\Omega}, \smile \rangle$, where: the meet operation $\curlywedge$ gives, for any two flow policies F_1, F_2, the strictest policy that allows for both F_1 and F_2; the join operation $\curlyvee$ gives, for any two flow policies F_1, F_2, the most permissive policy that only allows what both F_1 and F_2 allow; the most restrictive flow policy $\mho$ does not allow any information flows; and the most permissive flow policy Ω that allows all information flows. Finally, the pseudo-subtraction operation $\smile$ between two flow policies F_1 and F_2 (used only in Subsection 5.3) represents the most permissive policy that allows everything that is allowed by the first (F_1), while excluding all that is allowed by the second (F_2); it is defined as the relative pseudo-complement of F_2 with respect to F_1, i.e. the greatest F such that $F \curlywedge F_2 \preccurlyeq F_1$.

Considering a concrete example of a lattice of flow polices that meets the abstract requirements defined above can provide helpful intuitions. Flow policies that operate over the security lattice where security levels are sets of principals $p, q \in \boldsymbol{Pri}$ provide such a case. In this setting, security levels are ordered by means of the flow relation $\supseteq$. Flow policies then consist of binary relations on $\boldsymbol{Pri}$, which can be understood as representing additional directions in which information is allowed to flow between principals: a pair $(p, q) \in F$, most often written $p \prec q$, is to be understood as "information may flow from p to q". New more permissive security lattices are obtained by collapsing security levels into possibly lower ones, by closing them with respect to the valid flow policy. Writing $F_1 \preccurlyeq F_2$ means that F_1 allows flows between at least as many pairs of principals as F_2. The relation is here defined as $F_1 \preccurlyeq F_2$ iff $F_2 \subseteq F_1^*$ (where F^* denotes the reflexive and transitive closure of F): The meet operation is then defined as $\curlywedge = \cup$, the join operation is defined as $F_1 \curlyvee F_2 = F_1^* \cap F_2^*$, the top flow policy is given by $\mho = \emptyset$, the bottom flow policy is given by $\Omega = \boldsymbol{Pri} \times \boldsymbol{Pri}$, and the pseudo-subtraction operation is given by $\smile = -$.

3 Language

The language extends an imperative higher order lambda calculus that includes reference and concurrent thread creation, a declassification construct, and a policy-context testing construct, with basic distribution and code mobility features. Computation domains hold a local allowed flow policy, which imposes a limit on the permissiveness of the declassifications that are performed within the domain. A remote thread creation construct serves as a code migration primitive.

$$
\begin{array}{llcll}
\textit{Variables} & x & \in & \boldsymbol{Var} & \qquad \textit{Flow Policies} \quad A, F \in \boldsymbol{Flo} \\
\textit{Reference Names} & a & \in & \boldsymbol{Ref} & \qquad \textit{Domain Names} \quad d \in \boldsymbol{Dom} \\
\textit{Values} & V & \in & \boldsymbol{Val} & ::= () \mid x \mid a \mid (\lambda x.M) \mid tt \mid f\!f \\
\textit{Pseudo-values} & X & \in & \boldsymbol{Pse} & ::= V \mid (\varrho x.X) \\
\textit{Expressions} & M, N & \in & \boldsymbol{Exp} & ::= X \mid (M\ N) \mid (M; N) \mid (\text{if } M \text{ then } N_t \text{ else } N_f) \mid \\
 & & & & \quad (\text{ref}_\theta\ M) \mid (!\ N) \mid (M := N) \mid \mathbf{(flow}\ \boldsymbol{F}\ \mathbf{in}\ \boldsymbol{M}\mathbf{)} \mid \\
 & & & & \quad \mathbf{(allowed}\ \boldsymbol{F}\ \mathbf{then}\ \boldsymbol{N_t}\ \mathbf{else}\ \boldsymbol{N_f}\mathbf{)} \mid \mathbf{(thread}\ \boldsymbol{M}\ \mathbf{at}\ \boldsymbol{d}\mathbf{)}
\end{array}
$$

Figure 1. Syntax of Expressions

3.1 Syntax

The syntax of expressions defined in Figure 1 is based on a λ-calculus extended with the imperative constructs of ML, conditional branching and boolean values, where the $(\varrho x.X)$ construct provides for recursive values. Names of references (a), domains (d), and threads (m, n), are drawn from disjoint countable sets $\boldsymbol{Ref}$, $\boldsymbol{Dom} \neq \emptyset$ and $\boldsymbol{Nam}$, respectively. References are information containers to which values of the language pertaining to a given type in $\boldsymbol{Typ}$ can be assigned.

Declassification is introduced in the language by means of *flow policy declarations* [9]. They have the form (flow F in M), and are used to locally weaken the information flow policy that is valid for the particular execution context, by enabling information flows that comply to the flow policy F to take place within the scope of the delimited block of code M. Expression M is executed in the context of the current flow policy extended with F; after termination the current flow policy is restored, that is, the scope of F is M. For context-policy awareness, programs can inspect the allowed flow policy of the current domain by means of the *allowed-condition*, which is written (allowed F then N_t else N_f). The construct tests whether the flow policy F is allowed by the current domain and executes branches N_t or N_f accordingly, in practice offering alternative behaviors to be taken in case the domains they end up are too restrictive. For migration and concurrency, the thread creator (thread M at d) spawns the thread M in domain d, to be executed concurrently with other threads at that domain.

Networks are flat juxtapositions of domains, each containing a store and a pool of threads, which are subjected to the allowed flow policy of the domain. Threads run concurrently in *pools* $P : \boldsymbol{Nam} \to \boldsymbol{Exp}$, which are mappings from thread names to expressions (denoted as sets of threads). *Stores* $S : \boldsymbol{Ref} \to \boldsymbol{Val}$ map reference names to values. *Position-trackers* $T : \boldsymbol{Nam} \to \boldsymbol{Dom}$, map thread names to domain names, and are used to keep track of the locations of threads in the network. The pool P containing all the threads in the network, the mapping T that keeps track of their positions, and the store S containing all the references in the network, form *configurations* $\langle P, T, S \rangle$. The flow policies that are allowed by each domain are kept by the *allowed-policy mapping* $W : \boldsymbol{Dom} \to \boldsymbol{Flo}$ from domain names to flow policies, which is considered fixed in this model.

3.2 Operational Semantics

The small step operational semantics of the language is defined in Figure 2. The '$W \vdash^{\Sigma}$' turnstile makes explicit the allowed flow policy of each domain in the network, and the *reference labeling* Σ that determines the type of values that is assigned to each reference name. Other security-related information, such as security levels, could be added for the purpose of an information flow analysis.

The call-by-value evaluation order is specified by representing expressions using *evaluation contexts*.

$$\begin{aligned} \textit{Evaluation Contexts}\ \mathrm{E} ::= & [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid (\mathrm{ref}_{\theta}\ \mathrm{E}) \mid (!\ \mathrm{E}) \\ & (\mathrm{E} := N) \mid (V := \mathrm{E}) \mid (\text{if E then } N_t \text{ else } N_f) \mid \textbf{(flow } \boldsymbol{F} \textbf{ in E)} \end{aligned} \quad (1)$$

$$
\begin{array}{rl}
W \vdash^{\Sigma} \langle\{\mathrm{E}[((\lambda x.M)\ V)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[\{x \mapsto V\}M]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle\{\mathrm{E}[(\text{if } tt \text{ then } N_t \text{ else } N_f)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[N_t]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \mathrm{E}[(\text{if } ff \text{ then } N_t \text{ else } N_f)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[N_f]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle\{\mathrm{E}[(V; N)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[N]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle\{\mathrm{E}[(\varrho x.X)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[(\{x \mapsto (\varrho x.X)\}\ X)]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle\{\mathrm{E}[(\text{flow } F \text{ in } V)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[V]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle\{\mathrm{E}[(!\ a)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[S(a)]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle\{\mathrm{E}[(a := V)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[()]^m\}, T, [a := V]S\rangle \\
W \vdash^{\Sigma} \langle\{\mathrm{E}[(\text{ref}_\theta\ V)]^m\}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[a]^m\}, T, [a := V]S\rangle,\ \textit{a fresh in S} \\
& \qquad\qquad\qquad\qquad\qquad \textit{and } \Sigma(a) = \theta
\end{array}
$$

$$
\boldsymbol{\frac{W(T(m)) \preccurlyeq F}{W \vdash^{\Sigma} \langle\{\mathrm{E}[(\textbf{allowed } F \textbf{ then } N_t \textbf{ else } N_f)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[N_t]^m\}, T, S\rangle}}
$$

$$
\boldsymbol{\frac{W(T(m)) \not\preccurlyeq F}{W \vdash^{\Sigma} \langle\{\mathrm{E}[(\textbf{allowed } F \textbf{ then } N_t \textbf{ else } N_f)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[N_f]^m\}, T, S\rangle}}
$$

$$
\boldsymbol{W \vdash^{\Sigma} \langle\{\mathrm{E}[(\textbf{thread } N \textbf{ at } d)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle\{\mathrm{E}[()]^m, N^n\}, [n := d]T, S\rangle,}
$$
$$
\boldsymbol{n \textit{ fresh in } \mathbf{T}}
$$

$$
\frac{W \vdash^{\Sigma} \langle P, T, S\rangle \xrightarrow[F]{} \langle P', T', S'\rangle \quad \langle P \cup Q, T, S\rangle \textit{ is well formed}}{W \vdash^{\Sigma} \langle P \cup Q, T, S\rangle \xrightarrow[F]{} \langle P' \cup Q, T', S'\rangle}
$$

Figure 2. Operational Semantics

We write E$[M]$ to denote an expression where the sub-expression M is placed in the evaluation context E, obtained by replacing the occurrence of [] in E by M. The flow policy that is permitted by the evaluation context E is denoted by $\lceil \mathrm{E} \rceil$. It consists a lower bound (see Section 2) to all the flow policies that are declared by the context:

$$
\begin{array}{cc}
\lceil [] \rceil = \mho, & \lceil (\text{flow } F \text{ in E}) \rceil = F \curlywedge \lceil \mathrm{E} \rceil, \\
\lceil \mathrm{E}'[\mathrm{E}] \rceil = \lceil \mathrm{E} \rceil, & \textit{when } \mathrm{E}' \textit{ does not contain flow declarations}
\end{array} \qquad (2)
$$

The following basic notations and conventions are useful for defining transitions. For a mapping Z, we define dom(Z) as the domain of a given mapping Z. We say a name is fresh in Z if it does not occur in dom(Z). We denote by rn(P) and dn(P) the set of reference and domain names, respectively, that occur in the expressions of P. We let fv(M) be the set of variables occurring free in M. We restrict our attention to well formed configurations $\langle P, T, S\rangle$ satisfying the conditions that rn$(P) \subseteq$ dom(S), that dn$(P) \subseteq$ dom(W), that dom$(P) \subseteq$ dom(T), and that, for any $a \in$ dom(S), rn$(S(a)) \subseteq$ dom(S) and dn$(S(a)) \subseteq$ dom(W).

We denote by $\{x \mapsto W\}M$ the capture-avoiding substitution of W for the free occurrences of x in M. The operation of adding or updating the image of an object z to z' in a mapping Z is denoted $[z := z']Z$.

The transition rules of our semantics are decorated with the flow policy declared by the evaluation context where the step is performed. The lifespan of the flow declaration terminates when the expression M that is being evaluated terminates (that is, M becomes a value). In particular, the evaluation of (flow F in M) simply consists in the evaluation of M, annotated with a flow policy that is at least as permissive as F. The flow policy that decorates the transition steps is used only by the rules for (allowed F then N_t else N_f), where the choice of the branch depends on whether F is allowed to be declared or not. The thread creation construct functions as a migration construct when the new domain of the created thread is different from that of the parent thread. The last rule establishes that the execution of a pool of threads is compositional (up to the expected restriction on the choice of new names). Notice that W, representing the allowed flow policies associated to each domain, is never changed.

For simplicity, we assume memory to be shared by all programs and every computation domain, in a transparent form. This does not remove the distributed nature of the model, as programs' behavior depends on where they are [6].

4 Security Property

In a distributed setting with concurrent mobile code, programs might need to comply simultaneously to different allowed flow policies that change dynamically. The property of flow policy confinement deals with this difficulty by placing individual restrictions on each step that might be performed by a part of the program, taking into account the possible location where it might take place.

Compatibility. Since we are considering a higher-order language, values stored in memory can be used by programs to build expressions that are then executed. In order to avoid deeming all such programs insecure, memories are assumed to be compatible to the given security setting and typing environment, requiring typability of their contents with respect to the relevant type system and parameters. Informally, a memory S is said to be (W, Σ, Γ)-compatible if for every reference $a \in \text{dom}(S)$ its value $S(a)$ is typable. This predicate will be defined for each security analysis, and can be shown to be preserved by the semantics.

Flow Policy Confinement. The property is defined co-inductively for *located threads*, consisting of pairs $\langle d, M^m \rangle$ that carry information about the location d of a thread M^m. The location of each thread determines which allowed flow policy it should obey at that point, and is used to place a restriction on the flow policies that decorate the transitions: at any step, they should comply to the allowed flow policy of the domain where the thread who performed it is located.

Definition 1 ((W, Σ, Γ)-Confined Located Threads). *Consider an allowed-policy mapping W, a reference labeling Σ, and a typing environment Γ. A set $\mathcal{C}$*

of located threads is a set of (W, Σ, Γ)-confined located threads *if the following holds for all* $\langle d, M^m \rangle \in \mathcal{C}$*, for all* T *such that* $T(m) = d$*, and for all* (W, Σ, Γ)*-compatible memories* S*:*

– $W \vdash^{\Sigma} \langle \{M^m\}, T, S\rangle \xrightarrow[F]{} \langle \{M'^m\}, T', S'\rangle$ *implies* $W(T(m)) \preccurlyeq F$ *and also* $\langle T'(m), M'^m\rangle \in \mathcal{C}$*. Furthermore,* S' *is still* (W, Σ, Γ)*-compatible.*
– $W \vdash^{\Sigma} \langle \{M^m\}, T, S\rangle \xrightarrow[F]{} \langle \{M'^m, N^n\}, T', S'\rangle$ *implies* $W(T(m)) \preccurlyeq F$ *and also* $\langle T'(m), M'^m\rangle, \langle T'(n), N^n\rangle \in \mathcal{C}$*. Furthermore,* S' *is still* (W, Σ, Γ)*-compatible.*

Note that for any W, Σ, and Γ there exists a set of (W, Σ, Γ)-confined located threads, like for instance $\boldsymbol{Dom} \times (\boldsymbol{Val} \times \boldsymbol{Nam})$. Furthermore, the union of a family of sets of (W, Σ, Γ)-confined located threads is a set of (W, Σ, Γ)-confined located threads. The largest set of (W, Σ, Γ)-confined threads is denoted by $\mathcal{C}_W^{\Sigma,\Gamma}$.

We say that a thread M^m is (W, Σ, Γ)-confined when located at d, if $\langle d, M^m\rangle \in \mathcal{C}_W^{\Sigma,\Gamma}$. A well formed *thread configuration* $\langle P, T\rangle$, satisfying the applicable rules of a well formed configuration, is said to be (W, Σ, Γ)-confined if all located threads in $\{\langle T(m), M^m\rangle \mid M^m \in P\}$ are (W, Σ, Γ)-confined.

Notice that this property speaks strictly about what *flow declarations* a thread can do *while* it is at a specific domain. In particular, it does not restrict threads from migrating to more permissive domains in order to perform a declassification. More importantly, the property does not deal with information flows. So for instance it offers no assurance that information leaks that are encoded at each point of the program do obey the declared flow policies for that point. Such an analysis can be done independently, cf. *non-disclosure* in [9].

5 Enforcement Mechanisms

In this section we start by studying a type system for statically ensuring that global computations always comply to the locally valid allowed flow policy. This type system is inherently restrictive, as the domains where each part of the code will actually compute cannot in general be known statically (Subsection 5.1). We then present a more precise type system to be used at runtime by the semantics of the language for checking migrating threads against the allowed flow policy of the destination domain (Subsection 5.2). Finally, we propose a yet more precise type and effect system that computes information about the declassification behaviors of programs. This information will be used more efficiently at runtime by the semantics of the language in order to control migration of programs.

5.1 Purely Static Type Checking

We have seen that in a setting where code can migrate between domains with different allowed security policies, the computation domain might change *during* computation, along with the allowed flow policy that the program must comply to. This can happen in particular within the branch of an allowed condition:

$$(\text{allowed } F \text{ then } (\text{thread } (\text{flow } F \text{ in } M_1) \text{ at } d) \text{ else } M_2) \tag{3}$$

$$[\textsc{Nil}]\ W;\Gamma \vdash^{\Sigma}_{A} () : \mathsf{unit} \qquad [\textsc{Bt}]\ W;\Gamma \vdash^{\Sigma}_{A} tt : \mathsf{bool} \qquad [\textsc{Bf}]\ W;\Gamma \vdash^{\Sigma}_{A} \mathit{ff} : \mathsf{bool}$$

$$[\textsc{Loc}]\ W;\Gamma \vdash^{\Sigma}_{A} a : \Sigma(a)\ \mathrm{ref} \qquad [\textsc{Var}]\ \Gamma, x:\tau \vdash^{\Sigma}_{A} x : \tau$$

$$[\textsc{Abs}]\ \frac{W;\Gamma, x:\tau \vdash^{\Sigma}_{A} M : \sigma}{W;\Gamma \vdash^{\Sigma}_{A'} (\lambda x.M) : \tau \xrightarrow{A} \sigma} \qquad [\textsc{Rec}]\ \frac{W;\Gamma, x:\tau \vdash^{\Sigma}_{A} X : \tau}{W;\Gamma \vdash^{\Sigma}_{A} (\varrho x.X) : \tau}$$

$$[\textsc{Ref}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \theta}{W;\Gamma \vdash^{\Sigma}_{A} (\mathrm{ref}_{\theta}\ M) : \theta\ \mathrm{ref}} \qquad [\textsc{Der}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \theta\ \mathrm{ref}}{W;\Gamma \vdash^{\Sigma}_{A} (!\ M) : \theta}$$

$$[\textsc{Ass}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \theta\ \mathrm{ref} \quad W;\Gamma \vdash^{\Sigma}_{A} N : \theta}{W;\Gamma \vdash^{\Sigma}_{A} (M := N) : \mathsf{unit}} \qquad [\textsc{Seq}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \tau \quad W;\Gamma \vdash^{\Sigma}_{A} N : \sigma}{W;\Gamma \vdash^{\Sigma}_{A} (M;N) : \sigma}$$

$$[\textsc{Cond}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \mathsf{bool} \quad \begin{array}{c} W;\Gamma \vdash^{\Sigma}_{A} N_t : \tau \\ W;\Gamma \vdash^{\Sigma}_{A} N_f : \tau \end{array}}{W;\Gamma \vdash^{\Sigma}_{A} (\mathrm{if}\ M\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f) : \tau}$$

$$[\textsc{App}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \tau \xrightarrow{A} \sigma \quad W;\Gamma \vdash^{\Sigma}_{A} N : \tau}{W;\Gamma \vdash^{\Sigma}_{A} (M\ N) : \sigma} \qquad [\textbf{Flow}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} N : \tau \quad A \preccurlyeq F}{W;\Gamma \vdash^{\Sigma}_{A} (\mathrm{flow}\ F\ \mathrm{in}\ N) : \tau}$$

$$[\textbf{Allow}]\ \frac{\begin{array}{c} W;\Gamma \vdash^{\Sigma}_{A \curlywedge F} N_t : \tau \\ W;\Gamma \vdash^{\Sigma}_{A} N_f : \tau \end{array}}{W;\Gamma \vdash^{\Sigma}_{A} (\mathrm{allowed}\ F\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f) : \tau}$$

$$[\textbf{Mig}]\ \frac{W;\Gamma \vdash^{\Sigma}_{W(d)} M : \mathsf{unit}}{W;\Gamma \vdash^{\Sigma}_{A} (\mathrm{thread}\ M\ \mathrm{at}\ d) : \mathsf{unit}}$$

Figure 3. Type and effect system for checking Confinement

In this program, the flow declaration of the policy F is executed only if F has been tested as being allowed by the domain where the program was started. It might then seem that the flow declaration is "protected" by an appropriate allowed construct. However, by the time the flow declaration is performed, the thread is already located at another domain, where that flow policy might not be allowed. It is clear that a static enforcement of a confinement property requires tracking the possible locations where threads might be executing at each point.

Figure 3 presents a new type and effect system [10] for statically enforcing confinement over a migrating program. The type system guarantees that when operations are executed by a thread within the scope of a flow declaration, the declared flow complies to the allowed flow policy of the current domain. The typing judgments have the form

$$W;\Gamma \vdash^{\Sigma}_{A} M : \tau \tag{4}$$

meaning that expression M is typable with type τ in typing context $\Gamma : \boldsymbol{Var} \to \boldsymbol{Typ}$, which assigns types to variables. In addition to the reference mapping Σ, the turnstile has as parameter the flow policy A that is *allowed by the context*, which includes all flow policies that have been positively tested by the program as being allowed at the computation domain where the expression M is running. Finally, W represents the mapping of domain names to allowed flow policies.

Types have the standard syntax (t is a type variable)

$$\tau, \sigma, \theta \in \textit{\textbf{Typ}} ::= t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta \text{ ref} \mid \tau \xrightarrow{A} \sigma \quad (5)$$

where the reference type records the type θ of values that the reference contains, and the functional type records the latent allowed policy A that is used to type the application of the function to an argument.

Our type system applies restrictions to programs in order to ensure that flow declarations can only declare flow policies that are allowed by the context (rule Flow). These restrictions are relaxed when typing the first branch of allowed conditions, by extending the flow policy allowed by the context with the policy that guards the condition (rule Allow). In rule Mig, the flow policy allowed by the context is adjusted to that of the destination computation domain $W(d)$ that is specified by the (thread M at d) construct.

Note that if an expression is typable with respect to an allowed flow policy A, then it is also so for any more permissive allowed policy A'. In particular, due to the Abs rule, the process of typing an expression is not deterministic. For instance, the expression $(\lambda x.())$ can be given any type of the form $\tau \xrightarrow{F} \mathsf{unit}$.

We refer to the enforcement mechanism that consists of statically type checking all threads in a network according to the type and effect system of Figure 3, with respect to the allowed flow policies of each thread's initial domain, using the semantics represented in Figure 2, as *Enforcement mechanism I.*

Soundness. Enforcement mechanism I guarantees security of networks with respect to confinement, as is formalized by the following result. The (W, Σ, Γ)-compatibility predicate that is used to define confinement here requires all references $a \in \text{dom}(S)$ to store a value satisfying $W; \Gamma \vdash^{\Sigma}_{\mho} S(a) : \Sigma(a)$.

Theorem 1 (Soundness of Enforcement Mechanism I). *Consider a fixed allowed-policy mapping W, a given reference labeling Σ and typing environment Γ, and a thread configuration $\langle P, T\rangle$ such that for all $M^m \in P$ there exists τ such that $W; \Gamma \vdash^{\Sigma}_{W(T(m))} M : \tau$. Then $\langle P, T\rangle$ is (W, Σ, Γ)-confined.*

Proof. By showing that the set $\{\langle d, M^m\rangle \mid m \in \textit{\textbf{Nam}} \text{ and } \exists\tau.\, W; \Gamma \vdash^{\Sigma}_{W(d)} M : \tau\}$ is a set of (W, Σ, Γ)-confined located threads, using induction on the inference of $W; \Gamma \vdash^{\Sigma}_{W(d)} M : \tau$.

Precision. Given the purely static nature of this migration control analysis, some secure programs are bound to be rejected. There are different ways to increase the precision of a type system, which are all intrinsically limited to what can conservatively be predicted before runtime. For example, for the program

$$(\text{if } (!\,a) \text{ then } (\text{thread } (\text{flow } F \text{ in } M) \text{ at } d_1) \text{ else } (\text{thread } (\text{flow } F \text{ in } M) \text{ at } d_2)) \quad (6)$$

it is in general not possible to predict which branch will be executed (or, in practice, to which domain the thread will migrate), for it depends on the contents of the memory. It will then be rejected if $W(d_2) \not\preccurlyeq F$ or $W(d_1) \not\preccurlyeq F$.

5.2 Runtime Type Checking

In this subsection we study a hybrid mechanism for enforcing confinement, that makes use of a relaxation of the type system of Figure 3 during runtime. Migration is now controlled by means of a runtime check for typability of migrating threads with respect to the allowed flow policy of the destination domain. The condition represents the standard theoretical requirement of checking incoming code before allowing it to execute in a given machine.

The relaxation is achieved by replacing rule Mig by the following one:

$$[\textsc{Mig}]\ \frac{\Gamma \vdash^{\Sigma}_{\Omega} M : \mathsf{unit}}{\Gamma \vdash^{\Sigma}_{A} (\text{thread } M \text{ at } d) : \mathsf{unit}} \tag{7}$$

The new type system no longer imposes *future* migrating threads to conform to the policy of their destination domain, but only to the most permissive allowed flow policy Ω. The rationale is that it only worries about confinement of the non-migrating parts of the program. This is sufficient, as all threads that are to be spawned by the program will be re-checked at migration time.

The following modification to the migration rule of the semantics of Figure 2 introduces the runtime check that controls migration (n fresh in T). The idea is that a thread can only migrate to a domain if it respects its allowed flow policy:

$$\frac{\Gamma \vdash^{\Sigma}_{W(d)} N : \mathsf{unit}}{W \vdash^{\Sigma} \langle \{\mathrm{E}[(\text{thread } N \text{ at } d)]^{m}\}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{n} \langle \{\mathrm{E}[()]^{m}, N^{n}\}, [n := d]T, S\rangle} \tag{8}$$

The new remote thread creation rule (our migration primitive), now depends on typability of the migrating thread. The typing environment Γ (which is constant) is now an implicit parameter of the operational semantics. If only closed threads are considered, then also migrating threads are closed. The allowed flow policy of the destination site now determines whether or not a migration instruction may be consummated, or otherwise block execution.

Notice that, thanks to postponing the migration control to runtime, the type system no longer needs to be parameterized with information about the allowed flow policies of all domains in the network, which in practice could be impossible. The only relevant one are those of the destination domain of migrating threads.

We refer to the enforcement mechanism that consists of statically type checking all threads in a network according to the type and effect system of Figure 3 modified using the new Mig rule represented in Rule (8), with respect to the allowed flow policies of each thread's initial domain, using the semantics of Figure 2 modified according to Rule (7), as *Enforcement mechanism II.*

Soundness. Enforcement mechanism II guarantees security of networks with respect to confinement, as is formalized by the following result. The (W, Σ, Γ)-compatibility predicate that is used to define confinement here requires all references $a \in \mathrm{dom}(S)$ to store a value satisfying $\Gamma \vdash^{\Sigma}_{\mho} S(a) : \Sigma(a)$.

Theorem 2 (Soundness of Enforcement Mechanism II). *Consider a fixed allowed-policy mapping W, a given reference labeling Σ and typing environment Γ, and a thread configuration $\langle P, T\rangle$ such that for all $M^m \in P$ there exists τ such that $\Gamma \vdash^{\Sigma}_{W(T(m))} M : \tau$. Then $\langle P, T\rangle$ is (W, Σ, Γ)-confined.*

Proof. By showing that the set $\{\langle d, M^m\rangle \mid m \in \boldsymbol{Nam} \text{ and } \exists\tau \,.\, \Gamma \vdash^{\Sigma}_{W(d)} M : \tau\}$ is a set of (W, Σ, Γ)-confined located threads, using induction on the inference of $\Gamma \vdash^{\Sigma}_{W(d)} M : \tau$.

Safety, precision and efficiency. The proposed mechanism does not offer a safety result, guaranteeing that programs never "get stuck". Indeed, the side condition of the thread creation rule introduces the possibility for the execution of a thread to block, since no alternative is given. This can happen in Example 3 (in page 8), if the flow policy F is not permitted by the allowed policy of the domain of the branch that is actually executed, then the migration will not occur, and execution will not proceed. In order to have safety, we could design the thread creation instruction as including an alternative branch for execution in case the side condition fails. Nevertheless, Example 3 might have better been written

$$(\text{thread } (\text{allowed } F \text{ then } (\text{flow } F \text{ in } M_1) \text{ else } M_2) \text{ at } d) \tag{9}$$

in effect using the allowed condition for encoding such alternative behaviors.

Returning to Example 6 (in page 6), thanks to the relaxed Mig rule, this program is now *always* accepted statically by the type system. Depending on the result of the test, the migration might also be allowed to occur if a safe branch is chosen. This means that enforcement mechanism II accepts more secure programs. Because of the possibility of blockage mentioned above, an information flow analysis might reject some of the programs accepted here, in case for instance the reference a is assigned a "high" security level, and a "low" write is performed after the test. This issue is however orthogonal to our aims here.

A drawback with this enforcement mechanism lies in the computation weight of the runtime type checks. This is particularly acute for an expressive language such as the one we are considering. Indeed, recognizing typability of ML expressions has exponential (worst case) complexity [11].

5.3 Static Informative Typing for Runtime Effect Checking

We have seen that bringing the type-based migration control of programs to runtime allows to increase the precision of the confinement analysis. This is, however, at the cost of performance. It is possible to separate the program analysis as to what are the declassification operations that are performed by migrating threads, from the safety problem of determining whether those declassification operations should be allowed at a given domain. To achieve this, we now present an *informative* type system [8] that statically calculates a summary of all the declassification operations that might be performed by a program, in the form of a *declassification effect.* Furthermore, this type system produces a version of the

$$[\textsc{Nil}_\textsc{I}]\ \Gamma \vdash^{\Sigma} () \hookrightarrow () : \mho, \mathsf{unit} \quad [\textsc{Bt}_\textsc{I}]\ \Gamma \vdash^{\Sigma} tt \hookrightarrow tt : \mho, \mathsf{bool} \quad [\textsc{Bf}_\textsc{I}]\ \Gamma \vdash^{\Sigma} \mathit{ff} \hookrightarrow \mathit{ff} : \mho, \mathsf{bool}$$

$$[\textsc{Loc}_\textsc{I}]\ \Gamma \vdash^{\Sigma} a \hookrightarrow a : \mho, \Sigma(a) \text{ ref} \qquad [\textsc{Var}_\textsc{I}]\ \Gamma, x : \tau \vdash^{\Sigma} x \hookrightarrow x : \mho, \tau$$

$$[\textsc{Abs}_\textsc{I}]\ \frac{\Gamma, x : \tau \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \sigma}{\Gamma \vdash^{\Sigma} (\lambda x.M) \hookrightarrow (\lambda x.\hat{M}) : \mho, \tau \xrightarrow{s} \sigma} \qquad [\textsc{Rec}_\textsc{I}]\ \frac{\Gamma, x : \tau \vdash^{\Sigma} X \hookrightarrow \hat{X} : s, \tau}{\Gamma \vdash^{\Sigma} (\varrho x.X) \hookrightarrow (\varrho x.\hat{X}) : s, \tau}$$

$$[\textsc{Ref}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \theta' \quad \theta \preccurlyeq \theta'}{\Gamma \vdash^{\Sigma} (\text{ref}_\theta\ M) \hookrightarrow (\text{ref}_\theta\ \hat{M}) : s, \theta \text{ ref}} \qquad [\textsc{Der}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \theta \text{ ref}}{\Gamma \vdash^{\Sigma} (!\ M) \hookrightarrow (!\ \hat{M}) : s, \theta}$$

$$[\textsc{Ass}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \theta \text{ ref} \quad \Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s', \theta' \quad \theta \preccurlyeq \theta'}{\Gamma \vdash^{\Sigma} (M := N) \hookrightarrow (\hat{M} := \hat{N}) : s \curlywedge s', \mathsf{unit}}$$

$$[\textsc{Seq}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \quad \Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s', \sigma}{\Gamma \vdash^{\Sigma} (M; N) \hookrightarrow : s \curlywedge s', \sigma}$$

$$[\textsc{Cond}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \mathsf{bool} \quad \begin{array}{c}\Gamma \vdash^{\Sigma} N_t \hookrightarrow \hat{N}_t : s_t, \tau_t \\ \Gamma \vdash^{\Sigma} N_f \hookrightarrow \hat{N}_f : s_f, \tau_f\end{array} \quad \tau_t \approx \tau_f}{\Gamma \vdash^{\Sigma} (\text{if } M \text{ then } N_t \text{ else } N_f) \hookrightarrow (\text{if } \hat{M} \text{ then } \hat{N}_t \text{ else } \hat{N}_f) : s \curlywedge s_t \curlywedge s_f, \tau_t \curlywedge \tau_f}$$

$$[\textsc{App}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \xrightarrow{s'} \sigma \quad \Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s'', \tau'' \quad \tau \preccurlyeq \tau''}{\Gamma \vdash^{\Sigma} (M\ N) \hookrightarrow (\hat{M}\ \hat{N}) : s \curlywedge s' \curlywedge s'', \sigma}$$

$$[\textbf{Flow}_\textbf{I}]\ \frac{\Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s, \tau}{\Gamma \vdash^{\Sigma} (\text{flow } F \text{ in } N) \hookrightarrow (\text{flow } F \text{ in } \hat{N}) : s \curlywedge F, \tau}$$

$$[\textbf{Allow}_\textbf{I}]\ \frac{\begin{array}{c}\Gamma \vdash^{\Sigma} N_t \hookrightarrow \hat{N}_t : s_t, \tau_t \\ \Gamma \vdash^{\Sigma} N_f \hookrightarrow \hat{N}_f : s_f, \tau_f\end{array} \quad \tau_t \approx \tau_f}{\Gamma \vdash^{\Sigma} (\text{allowed } F \text{ then } N_t \text{ else } N_f) \hookrightarrow (\text{allowed } F \text{ then } \hat{N}_t \text{ else } \hat{N}_f) : s_t \smile F \curlywedge s_f, \tau_t \curlywedge \tau_f}$$

$$[\textbf{Mig}_\textbf{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \mathsf{unit}}{\Gamma \vdash^{\Sigma} (\text{thread } M \text{ at } d) \hookrightarrow (\text{thread}^s\ \hat{M} \text{ at } d) : \mho, \mathsf{unit}}$$

Figure 4. Informative Type and Effect System for obtaining the Declassification Effect

program that is annotated with the relevant information for deciding, at runtime, whether its migrating threads can be considered safe by the destination domain. The aim is to bring the overhead of the runtime check to static time.

The typing judgments of the type system in Figure 4 have the form:

$$\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \tag{10}$$

Comparing with the typing judgments of Subsection 5.2, while the flow policy allowed by the context parameter is omitted from the turnstile '$\vdash$', the security effect s represents a flow policy which corresponds to the *declassification effect*: a lower bound to the flow policies that are declared in the typed expression. The second expression $\hat{M}$ is the result of annotating M. The syntax of annotated expressions differs only in the thread creation construct, that has an additional flow policy F as parameter, written (thread^F M at d). The syntax of types is the same as the ones used in Subsections 5.1 and 5.2.

It is possible to relax the type system by matching types that have the same structure, even if they differ in flow policies pertaining to them. We achieve this by overloading $\preccurlyeq$ to relate types where certain latent effects in the first are at least as permissive as the corresponding ones in the second. The more general relation $\approx$ matches types where certain latent effects differ: Finally, we define an operation $\curlywedge$ between two types τ and τ' such that $\tau \approx \tau'$:

$$\begin{gathered}\tau \preccurlyeq \tau' \text{ iff } \tau = \tau',\ \text{or } \tau = \theta \xrightarrow{F} \sigma \text{ and } \tau' = \theta \xrightarrow{F'} \sigma' \text{ with } F \preccurlyeq F' \text{ and } \sigma \preccurlyeq \sigma' \\ \tau \approx \tau' \text{ iff } \tau = \tau',\ \text{or } \tau = \theta \xrightarrow{F} \sigma \text{ and } \tau' = \theta \xrightarrow{F'} \sigma' \text{ with } \sigma \approx \sigma' \qquad (11) \\ \tau \curlywedge ! \tau' = \tau,\ \text{if } \tau = \tau',\ \text{or } \theta \xrightarrow{F \curlywedge F'} \sigma \curlywedge \sigma',\ \text{if } \tau = \theta \xrightarrow{F} \sigma \text{ and } \tau' = \theta \xrightarrow{F'} \sigma'\end{gathered}$$

The $\preccurlyeq$ relation is used in rules Ref$_\text{I}$, Ass$_\text{I}$ andApp$_\text{I}$, in practice enabling to associate to references and variables (by reference creation, assignment and application) expressions with types that contain stricter policies than required by the declared types. The relation $\approx$ is used in rules Cond$_\text{I}$ andAllow$_\text{I}$ in order to accept that two branches of the same test construct can differ regarding some of their policies. Then, the type of the test construct is constructed from both using $\curlywedge$, thus reflecting the flow policies in both branches.

The declassification effect is constructed by aggregating (using the meet operation) all relevant flow policies that are declared within the program. The effect is updated in rule Flow$_\text{I}$, each time a flow declaration is performed, and "grows" as the declassification effects of sub-expressions are met in order to form that of the parent command. However, when a part of the program is "protected" by an allowed condition, some of the information in the declassification effect can be discarded. This happens in rule Allow$_\text{I}$, where the declassification effect of the first branch is not used entirely: the part that will be tested during execution by the allowed-condition is omitted. In rule Mig$_\text{I}$, the declassification effect of migrating threads is also not recorded in the effect of the parent program, as they will be executed (and tested) elsewhere. That information is however used to annotate the migration instruction.

One can show that the type system is deterministic, in the sense that it assigns to a non-annotated expression a single annotated version of it, a single declassification effect, and a single type.

Modified operational semantics, revisited. By executing annotated programs, the type check that conditions the migration instruction can be replaced by a simple declassification effect inspection. The new migration rule is similar to the one in Subsection 5.2, but now makes use of the declassification effect (n fresh in T):

$$\frac{W(d) \preccurlyeq s}{W \vdash^{\Sigma} \langle \{\mathrm{E}[(\text{thread}^{s}\ N \text{ at } d)]^{m}\}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{n} \langle \{\mathrm{E}[()]^{m}, N^{n}\}, [n := d]T, S\rangle} \qquad (12)$$

In the remaining rules of the operational semantics the annotations are ignored.

We refer to the mechanism that consists of statically annotating all threads in a network according to the type and effect system of Figure 4, assuming that each thread's declassification effect is allowed by its initial domain, using the semantics of Figure 2 modified according to Rule (12), as *Enforcement mechanism III.*

Soundness. We will now see that the declassification effect can be used for enforcing confinement. The (W, Σ, Γ)-compatibility predicate that is used to define confinement here requires all references $a \in \text{dom}(S)$ to store a value that results from annotating some other value V according to $\Gamma \vdash^{\Sigma}_{\mho} V \hookrightarrow S(a) : \Sigma(a)$.

Theorem 3 (Soundness of Enforcement Mechanism III). *Consider a fixed allowed-policy mapping W, a given reference labeling Σ and typing environment Γ, and a thread configuration $\langle P, T\rangle$ such that for all $M^m \in P$ there exist $\hat{M}$, s and τ such that $\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau$ and $W(T(m)) \preccurlyeq s$. Then $\langle \hat{P}, T\rangle$, formed by annotating the threads in $\langle P, T\rangle$, is (W, Σ, Γ)-confined.*

Proof. By showing that the following is a set of (Σ, Γ)-confined located threads

$$\{\langle d, \hat{M}^m\rangle \mid m \in \textit{\textbf{Nam}} \textit{ and } \exists M, s, \tau \;.\; \Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \text{ and } W(d) \preccurlyeq s\} \quad (13)$$

using induction on the inference of $\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau$.

Precision and efficiency. The relaxed type system of Subsection 5.2 for checking confinement, and its informative counterpart of Figure 4, are strongly related. The following result states that typability according to latter type system is at least as precise as the former. It is proven by induction on the inference of $\Gamma \vdash^{\Sigma}_{A} M : \tau$.

Proposition 1. *Consider a given a typing environment Γ and reference labeling Σ. If there exist A, τ such that $\Gamma \vdash^{\Sigma}_{A} M : \tau$, then there exist $\hat{M}$, τ' and s such that $\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau'$ and $A \preccurlyeq s$ with $\tau \preccurlyeq \tau'$.*

The converse direction is not true, i.e. enforcement mechanism III accepts strictly more programs than enforcement mechanism II. This can be seen by considering the secure program where, $\theta_1 = \tau \xrightarrow{F_1} \sigma$ and $\theta_2 = \tau \xrightarrow{F_2} \sigma$:

$$(\text{if } (!\ a) \text{ then } (!\ (\text{ref}_{\theta_1}\ M_1)) \text{ else } (!\ (\text{ref}_{\theta_2}\ M_2))) \quad (14)$$

This program is not accepted by the type system of Section 5.2 because it cannot give the same type to both branches of the conditional (the type of the dereference of a reference of type θ is precisely θ). However, since the two types satisfy $\theta_1 \approx \theta_2$, the informative type system can accept it and give it the type $\theta_1 \curlywedge \theta_2$.

A more fundamental difference between the two enforcement mechanisms lays in the timing of the computation overhead that is required by each mechanism. While mechanism II requires heavy runtime type checks to occur each time a thread migrates, in III the typability analysis is anticipated to static time, leaving only a comparison between two flow policies to be performed at migration time.

The complexity of this comparison depends on the concrete representation of flow policies. In the worst case, that of flow policies as general downward closure operators (see Section 2), it is linear on the number of security levels that are considered. When flow policies are flow relations, then it consists on a subset relation check, which is polynomial on the size of the flow policies.

6 Related Work

Controlling declassification. Most previous mechanisms for controlling declassification [12] target flexible versions of an information flow property. Departing from this approach, the work by Boudol and Kolundzija [13] on combining access control and declassification is the first to treat declassification control separately from the underlying information flow problem. In [13], standard access control primitives are used to control the access level of programs that perform declassifications in the setting of a local language, ensuring that a program can only declassify information that it has the right to read.

Controlling code mobility. A wide variety of distributed network models have been designed with the purpose of studying mechanisms for controlling code mobility. These range from type systems for statically controlling migration as an access control mechanism [5,14], to runtime mechanisms that are based on the concept of programmable domain. In the latter, computing power is explicitly associated to the *membranes* of computation domains, and can be used for controlling boundary transposition. This control can be performed by processes that interact with external and internal programs [15,16,4], or by more specific automatic verification mechanisms [17]. In the present work we abstract away from the particular machinery that implements the migration control checks, and express declaratively, via the language semantics, the condition that must be satisfied for the boundary transposition to be allowed.

Checking the validity of the declassification effect as a certificate is not simpler than checking the program against a concrete allowed policy (as presented in Subsection 5.2), meaning that it does not consist of a case of Proof Carrying Code. The concept of trust can be used to lift the checking requirements of code whose history of visited domains provides enough reassurance [17,5]. These ideas could be applied to the present work, assisting the decision of trusting the declassification effect, otherwise leading to a full type check of the code.

Hybrid mechanisms. The use of hybrid mechanisms for enforcing information flow policies is currently an active research area (see [18] for a review of related work). The closest to ours is perhaps the study of securing information release for a simple language with dynamic code evaluation in the form of a string eval command, which includes an on-the-fly information flow static analysis [19].

Focusing on declassification control, the idea of using a notion of declassification effect for building a runtime migration control mechanism was put forward in [6] for a similar language with local thread creator and a basic goto migration instruction. In spite of the restrictions that are pointed out in Subsection 5.1 for a static analysis, the type system presented as part of Enforcement Mechanism I is more refined than the proof-of-concept presented earlier. Indeed, in the previous work, migration was not taken into account when analyzing the declassifications occurring within the migrating code. So while there the following program would be rejected if F was not allowed by $W(d_1)$

$$(\text{thread } (\text{thread } (\text{flow } F \text{ in } M) \text{ at } d_2) \text{ at } d_1) \tag{15}$$

the type system of Figure 3 only rejects it if F is not allowed by $W(d_2)$. Enforcement Mechanism II adopts part of the idea in [6] of performing a runtime type analysis to migrating programs, but uses a more permissive "checking" type system. Enforcement Mechanism III explores a mechanism that allows to take advantage of the efficiency of flow policy comparisons. It uses a type and effect system for calculating declassification effects that is substantially more precise than previous ones, thanks to the matching relations and operations that it uses.

The concept of informative type and effect system was introduced in [8], where a different notion of declassification effect was defined and applied to the problem of dealing with dynamic updates to a local allowed flow policy.

7 Conclusion

We have considered an instance of the problem of enforcing compliance of declassifications to a dynamically changing allowed flow policy. In our setting, changes in the allowed flow policy result from the migration of programs during execution. We approach the problem from a migration control perspective. To this end, we chose a network model that abstracts away the details of the migration control architecture. This allows us to prove soundness of a concrete network level security property, guaranteeing that programs can roam over the network, never performing declassifications that violate the network confinement property.

While our results are formulated for a particular security property – flow policy confinement – we expect that similar ideas can be used for other properties. One could add expressiveness to the property by taking into account the history of domains that a thread has visited when defining secure code migrations. For instance, one might want to forbid threads from moving to domains with more favorable allowed flow policies. This would be easily achieved by introducing a condition on the allowed flow policies of origin and destination domains.

By performing comparisons between three related enforcement mechanisms, we have argued that the concept of declassification effect offers a good balance between precision and efficiency. We believe that similar mechanisms can be applied in other contexts. For future work, we plan to study others instances of enabling dynamic changing allowed flow policies.

Acknowledgments. This work benefits from the authors' visits to the Indes Team at INRIA Sophia Antipolis and the Language Based Security Group at Chalmers University of Technology, partially supported by the Portuguese Foundation for Science and Technology. Thanks are due to José Santos for insightful feedback.

References

1. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
2. J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.
3. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society, 2005.
4. G. Boudol. A generic membrane model. In Corrado Priami and Paola Quaglia, editors, *IST/FET International Workshop on Global Computing*, volume 3267 of *LNCS*, pages 208–222. Springer, 2005.
5. F. Martins and V.T. Vasconcelos. History-based access control for distributed processes. In *Proceedings of TGC'05*, LNCS. Springer-Verlag, 2005.
6. A. Almeida Matos. Flow-policy awareness for distributed mobile code. In *Proceedings of CONCUR 2009 - Concurrency Theory*, volume 5710 of *Lecture Notes in Computer Science*. Springer, 2009.
7. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
8. A. Almeida Matos and J. Fragoso Santos. Typing illegal information flows as program effects. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, pages 1:1–1:12. ACM, 2012.
9. A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. *Journal of Computer Security*, 17(5):549–597, 2009.
10. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th ACM Symp. on Principles of Programming Languages*, pages 47–57. ACM Press, 1988.
11. H. G. Mairson. Deciding ml typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 382–401. ACM, 1990.
12. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.
13. G. Boudol and M. Kolundzija. Access Control and Declassification. In *Computer Network Security*, volume 1 of *CCIS*, pages 85–98. Springer-Verlag, 2007.
14. M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *Proc. of the 6th Int. Conf. on Foundations of Software Science and Computation Structures and European Conf. on Theory and Practice of Software*, pages 282–298. Springer-Verlag, 2003.
15. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT ymposium on Principles of Programming Languages*, pages 352–364, New York, NY, USA, 2000. ACM.
16. A. Schmitt and J.-B. Stefani. The m-calculus: a higher-order distributed process calculus. *SIGPLAN Not.*, 38(1):50–61, 2003.
17. D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. In *Foundations of Global Ubiquitous Computing, FGUC 2004*, ENTCS, pages 23–42. Elsevier, 2005.
18. S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 146–160. IEEE Computer Society, 2011.
19. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 43–59. IEEE Computer Society, 2009.