Non-disclosure for Distributed Mobile Code

Ana Almeida Matos

INRIA Sophia Antipolis

Abstract. This paper addresses the issue of confidentiality and declassification for global computing in a language-based security perspective. The purpose is to deal with new forms of security leaks, which we call *migration leaks*, introduced by code mobility. We present a generalization of the non-disclosure policy [AB05] to networks, and a type and effect system for enforcing it. We consider an imperative higher-order lambda-calculus with concurrent threads and a flow declaration construct, enriched with a notion of domain and a standard migration primitive.

1 Introduction

Protecting confidentiality of data is a concern of particular relevance in a global computing context. When information and programs move throughout networks, they become exposed to users with different interests and responsibilities. This motivates the search for practical mechanisms that enforce respect for confidentiality of information, while minimizing the need to rely on mutual trust. Access control is important, but it is not enough, since it is not concerned with how information may flow between the different parts of a system. Surprisingly, very little research has been done on the control of information flow in networks. In fact, to the best of our knowledge, this work is the first to address the problem in an imperative setting where mobility of resources plays an explicit role.

This paper is about ensuring confidentiality in networks. More specifically, it is about controlling information flows between subjects that have been given different security clearances, in the context of a distributed setting with code mobility. Clearly, in such a setting, one cannot assume resources to be accessible by all programs at all times. In fact, a network can be seen as a collection of sites where conditions for computation to occur are not guaranteed by one site alone. Could these failures be exploited as covert information flow channels? The answer is *Yes*. New security leaks, that we call *migration leaks*, arise from the fact that execution or suspension of programs now depend on the position of resources over the network, which may in turn depend on secret information.

We take a language based approach [SM03], which means that we restrict our concern to information leaks occurring within computations of programs of a given language. These can be statically prevented by means of a type and effect system [VSI96, LG88], thus allowing rejection of insecure programs before execution. As is standard, we attribute security levels to the objects of our language (memory addresses), and have them organized into a lattice [Den76]. Since confidentiality is the issue, these levels indicate to which subjects the contents of an object are allowed to be disclosed. Consequently, during computation, information contained in objects of "high" security level (let us call them "high objects") should never influence objects of lower or incomparable level. This policy has been widely studied and is commonly referred to as *non-interference* [GM82]. In a more general setting, where the security lattice may vary within a program, *non-disclosure* [AB05] can be used instead.

We consider a calculus for mobility where the notion of location of a program and of a resource has an impact in computations: resources and programs are distributed over computation sites – or *domains* – and can change position during execution; accesses to a resource can only be performed by a program that is located at the same site; remote accesses are suspended until the resources become available. The language of local computations is an imperative λ-calculus with concurrent threads, to which we add a standard migration primitive. We include a flow declaration construct [AB05] for providing the programmer with means to declassify information, that is to explicitly allow certain information leaks to occur in a controlled way (find overviews in [AB05, SS05]). We show that mobility and declassification can be safely combined provided that migrating threads compute according to declared flow policies.

The security properties we have at hand, designed for local computations where the notion of locality does not play a crucial role, are not suitable for treating information flows in a distributed setting with code mobility. In fact, since the location of resources in a network can be itself a source of information leaks, the notion of safe program must take this into account. For this purpose, we extend the usual undistinguishability relation for memories to states that track the positions of programs in a network. Furthermore, it is not reasonable to assume a global security policy that all threads comply to. Admitting that each program has its own security policy raises problems in ensuring that the threads who share resources respect one another's flow policies. For instance, when should one allow "low level" information to be accessed by "high level" readers, if the assignment of the levels 'low' and 'high' were based on different criteria? It turns out that, if the security levels are sets of *principals*, there is a "minimum" security policy that every thread must satisfy, and which we use to conveniently approximate the "intersection" of security policies in any network.

The paper is organized as follows: In the next section we define a distributed calculus with code and resource mobility. In Section 3 we formulate a non-disclosure property that is suitable for a decentralized setting. In Section 4 we develop a type and effect system that only accepts programs satisfying such a property. Finally, we comment on related work and conclude.

2 The Calculus

The design of network models is a whole research area in itself, and there exists a wide spectrum of calculi that focus on different aspects of mobility (see [BCGL02]). We are interested in a general and simple framework that addresses the unreliable nature of resource access in networks, as well as trust concerns that are raised when computational entities follow different security orientations. We then consider a distributed ML-like core language where domains in a network can communicate with each other via mobile threads, in general composed of programs and memory, yet enriched with a flow declaration construct.

In this section we define the syntax and semantics for the calculus at the local and network level. Very briefly, a *network* consists of a number of *domains*, places where local computations occur independently. Threads may execute concurrently inside domains, create other threads, and *migrate* into another domain. They can own and create a *store* that associates values to *references*, which are addresses to memory containers. These stores move together with the thread they belong to, which means that threads and local references are, at all times, located in the same domain. However, a thread need not own a reference in order to access it. Read and write operations on references may be performed if and only if the corresponding memory location is present in the domain (otherwise they are implicitly suspended).

2.1 Syntax

In order to define the syntax of the language we need to introduce the notions of *security level* and of *flow policy* (they are fully explained in Section 3). *Security levels* j, k, l are sets of *principals* (ranged over by $p, q \in \mathcal{P}$). They are apparent in the syntax as they are assigned to references (and reference creators, not to values) and threads (and thread creators). The security level of a reference is to be understood as the set of principals that are allowed to read the information contained in that reference. The security level of a thread is the set of principals that can have information about the location of the thread. We use *flow policies* as in [AB05] for defining a flow declaration construct that enables downgrading computations by encapsulating expressions in a context allowed by the security policy. For now it is enough to know that a flow policy (ranged over by F, G) is a binary relation over $\mathcal{P}$, where a pair $(p, q) \in F$ is denoted $p \prec q$, and is to be understood as "whatever p can read, q can also read".

Names are given to domains ($d \in \mathcal{D}$), threads ($m, n \in \mathcal{N}$) and references (a), which we also call *addresses*. References are lexically associated to the threads that create them: they are of the form $m.u, n.u$, where u is an identifier given by the thread. Thread and reference names can be created at runtime. We add annotations (subscripts) to names: references carry their security level and the type of the values that they can hold (the syntax of types will be defined later, in Section 4), while thread names carry their security level. In the following we may omit these subscripts whenever they are not relevant, following the convention that the same name has always the same subscript.

Threads are named expressions (M^{m_j}), where the syntax of M is given by:

$$\begin{array}{lrcl}
\textit{Expressions} & M, N & ::= & V \mid x \mid (M\ N) \mid (\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2) \\
 & & \mid & (M; N) \mid (\mathsf{ref}_{l,\theta}\ M) \mid (?\ N) \mid (M :=^{?} N) \mid (\varrho x V) \\
 & & \mid & (\mathsf{thread}_l\ M) \mid (\mathsf{goto}\ d) \mid (\mathsf{flow}\ F\ \mathsf{in}\ M) \\
\textit{Values} & V, W & ::= & () \mid m_j.u_{l,\theta} \mid (\lambda x.M) \mid tt \mid ff
\end{array}$$

The language of expressions is an imperative higher-order λ-calculus with thread creation $(\mathsf{thread}_l\ M)$, migration $(\mathsf{goto}\ d)$ and a flow declaration $(\mathsf{flow}\ F\ \mathsf{in}\ M)$. The commands $(?\ N)$ and $(M :=^{?} N)$ correspond to the dereferencing and assignment operations on references, respectively. The different notation is due to the fact that these operations can potentially suspend. The notation follows [Bou04], though here we shall not consider any form of reaction to suspension. The construct $(\varrho x V)$, where x is binded in V, is used to express recursive values.

We define stores S that map *references* to values, and pools (sets) P of *threads* (named expressions) that run concurrently. These two sets are part of domains $d[\,P, S\,]$, which in turn form networks whose syntax is given by:

$$\textit{Networks}\ \ X, Y \ldots \quad ::= \quad d[\,P, S\,] \ \mid\ X \parallel Y$$

Networks are flat juxtapositions of domains, whose names are assumed to be distinct, and where references are assumed to be located in the same domain as the thread that owns them. Notice that networks are in fact just a collection of threads and owned references that are running in parallel, and whose executions depend on their relative location. To keep track of the locations of threads and references it suffices to maintain a mapping from thread names to domain names.

2.2 Semantics

Given a set $\mathcal{D}$ of domain names in a network, and assuming that all threads in a configuration have distinct names, the semantics of the language is operationally defined as a transition system between configurations of the form $\langle T, P, S\rangle$ representing network $d_1[\,P_1, S_1\,] \parallel \cdots \parallel d_n[\,P_n, S_n\,]$ where:

T is a function from thread names with security level to the domains where they appear, given by $T = \{m_l \mapsto d_1 | M^{m_l} \in P_1\} \cup \cdots \cup \{m_l \mapsto d_n | M^{m_l} \in P_n\}$.

P and S are the (disjoint) unions of all the thread pools, respectively stores, that exist in the network, that is $P = P_1 \cup \cdots \cup P_n$ and $S = S_1 \cup \cdots \cup S_n$.

We call the pair (T, S) the *state* of the configuration. We define $\mathsf{dom}(T)$, $\mathsf{dom}(P)$ and $\mathsf{dom}(S)$ as the sets of decorated names of threads and references that are mapped by T, P and S, respectively. We say that a thread or reference name is fresh in T or S if it does not occur, with any subscript, in $\mathsf{dom}(T)$ or $\mathsf{dom}(S)$, respectively. We denote by $\mathsf{tn}(P)$ and $\mathsf{rn}(P)$ the set of decorated thread and reference names, respectively, that occur in the expressions of P (this notation is extended in the obvious way to expressions). Furthermore, we overload tn and define, for a set R of reference names, the set $\mathsf{tn}(R)$ of thread names that are prefixes of some name in R.

We restrict our attention to *well formed configurations* $\langle T, P, S\rangle$ satisfying the condition for memories $\mathsf{dom}(S) \supseteq \mathsf{rn}(P)$, a similar condition for the values stored in memories obeying $a_{l,\theta} \in \mathsf{dom}(S)$ implies $\mathsf{dom}(S) \supseteq \mathsf{rn}(S(a_{l,\theta}))$, and the corresponding one for thread names $\mathsf{dom}(T) \supseteq \mathsf{dom}(P)$ and $\mathsf{dom}(T) \supseteq \mathsf{tn}(\mathsf{dom}(S))$. We denote by $\{x \mapsto W\}M$ the capture avoiding substitution of W for the free occurrences of x in M. The operation of updating the image of an object z_o to z_i in a mapping Z is denoted $Z[z_o := z_i]$.

In order to define the operational semantics, it is useful to write expressions using evaluation contexts. Intuitively, the expressions that are placed in such contexts are to be executed first.

$$\begin{array}{lll} \textit{Contexts} & \mathbf{E} ::= [] \mid (\mathbf{E}\ N) \mid (V\ \mathbf{E}) \mid (\mathsf{if}\ \mathbf{E}\ \mathsf{then}\ M\ \mathsf{else}\ N) \mid (\mathbf{E}; N) \\ & \quad \mid\ (\mathsf{ref}_{l,\theta}\ \mathbf{E}) \mid (?\ \mathbf{E}) \mid (\mathbf{E} :=^{?} N) \mid (V :=^{?} \mathbf{E}) \mid (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{E}) \end{array}$$

Evaluation is *not* allowed under threads that have not yet been created. We denote by $\lceil \mathbf{E} \rceil$ the flow policy that is permitted by the context $\mathbf{E}$. It collects all the flow policies that are declared using flow declaration constructs into one:

$$\lceil [] \rceil = \emptyset, \qquad \lceil (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{E}) \rceil = F \cup \lceil \mathbf{E} \rceil,$$
$$\lceil \mathbf{E}'[\mathbf{E}] \rceil = \lceil \mathbf{E} \rceil, \ \textit{if}\ \mathbf{E}'\ \textit{does not contain flow declarations}$$

The transitions of our (small step) semantics are defined between configurations, and are decorated with the flow policy of the context that is relevant to the expression being evaluated (it will be used later to formulate the non-disclosure property). We omit the set-brackets for singletons. We start by defining the transitions of a single thread.

The evaluation of expressions might depend on and change the store, the position of references in the network, and the name of the thread of which they are part. However, there are rules that depend only on the expression itself.

$$\begin{array}{rcl} \langle T, \mathsf{E}[((\lambda x.M)\ V)]^{m_j}, S \rangle & \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} & \langle T, \mathsf{E}[\{x \mapsto V\}M]^{m_j}, S \rangle \\ \langle T, \mathsf{E}[(\mathsf{if}\ tt\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)]^{m_j}, S \rangle & \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} & \langle T, \mathsf{E}[N_1]^{m_j}, S \rangle \\ \langle T, \mathsf{E}[(\mathsf{if}\ \mathit{ff}\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)]^{m_j}, S \rangle & \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} & \langle T, \mathsf{E}[N_2]^{m_j}, S \rangle \\ \langle T, \mathsf{E}[(V; N)]^{m_j}, S \rangle & \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} & \langle T, \mathsf{E}[N]^{m_j}, S \rangle \\ \langle T, \mathsf{E}[(\varrho x W)]^{m_j}, S \rangle & \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} & \langle T, \mathsf{E}[(\{x \mapsto (\varrho x W)\}\ W)]^{m_j}, S \rangle \\ \langle T, \mathsf{E}[(\mathsf{flow}\ F\ \mathsf{in}\ V)]^{m_j}, S \rangle & \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} & \langle T, \mathsf{E}[V]^{m_j}, S \rangle \end{array}$$

The name of the thread is relevant to the rules that handle references: when a reference is created, it is named after the parent thread. Accesses to references can only be performed within the same domain.

$$\langle T, \mathsf{E}[(\mathsf{ref}_{l,\theta}\ V)]^{m_j}, S \rangle \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} \langle T, \mathsf{E}[m_j.u_{l,\theta}]^{m_j}, S \cup \{m_j.u_{l,\theta} \mapsto V\} \rangle, \ \textit{if}\ m.u\ \textit{fresh in}\ S$$
$$\langle T, \mathsf{E}[(?\ n_k.u_{l,\theta})]^{m_j}, S \rangle \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} \langle T, \mathsf{E}[V]^{m_j}, S \rangle, \ \textit{if}\ T(n_k) = T(m_j)\ \&\ S(n_k.u_{l,\theta}) = V$$
$$\langle T, \mathsf{E}[(n_k.u_{l,\theta} :=^{?} V)]^{m_j}, S \rangle \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} \langle T, \mathsf{E}[()]^{m_j}, S[n_k.u_{l,\theta} := V] \rangle, \ \textit{if}\ T(n_k) = T(m_j)$$

Fresh names are arbitrarily attributed to threads when they are created. The (goto d) statement is used for sending the executing thread to a domain named d (subjective migration). By simply changing the domain that is associated to the migrating thread's name, both the thread and associated store are subtracted from the emitting domain and integrated into the destination domain.

$$\langle T, \{\mathsf{E}[(\mathsf{thread}_l\ N)]^{m_j}\}, S \rangle \xrightarrow[\lceil \mathbf{E} \rceil]{N^{n_l}} \langle T \cup \{n_l \mapsto T(m_j)\}, \{\mathsf{E}[()]^{m_j}\}, S \rangle, \ \textit{if}\ n\ \textit{fresh in}\ T$$
$$\langle T, \{\mathsf{E}[\mathsf{goto}\ d]^{m_j}\}, S \rangle \xrightarrow[\lceil \mathbf{E} \rceil]{\emptyset} \langle T[m_j := d], \{\mathsf{E}[()]\}, S \rangle$$

Finally, the execution of threads in a network is compositional. The following three rules gather the threads that are spawned into a pool of threads.

$$\frac{\langle T, M^{m_j}, S\rangle \xrightarrow[\lceil \mathbf{E} \rceil]{()} \langle T', M'^{m_j}, S'\rangle}{\langle T, M^{m_j}, S\rangle \xrightarrow[\lceil \mathbf{E} \rceil]{} \langle T', M'^{m_j}, S'\rangle} \qquad \frac{\langle T, M^{m_j}, S\rangle \xrightarrow[\lceil \mathbf{E} \rceil]{N^{n_l}} \langle T', M'^{m_j}, S'\rangle \quad \textit{if } N^{n_l} \neq ()}{\langle T, M^{m_j}, S\rangle \xrightarrow[\lceil \mathbf{E} \rceil]{} \langle T', \{M'^{m_j}, N^{n_l}\}, S'\rangle}$$

$$\frac{\langle T, P, S\rangle \xrightarrow[\lceil \mathbf{E} \rceil]{} \langle T', P', S'\rangle \quad \langle T, P \cup Q, S\rangle \textit{ is well formed}}{\langle T, P \cup Q, S\rangle \xrightarrow[\lceil \mathbf{E} \rceil]{} \langle T', P' \cup Q, S'\rangle}$$

One can prove that the above rules preserve well-formedness of configurations, and that the language of expressions is deterministic up to choice of new names.

3 Decentralized Non-disclosure Policies

We begin this section with the definition of an extended flow relation. A discussion on the implementation and meaning of multiple flow policies follows. We then define the non-disclosure policy for networks and prove soundness of our type system with respect to that property.

3.1 From Flow Relations to Security Lattices

We have mentioned that security levels are sets of principals representing read-access rights to references. Our aim is to insure that information contained in a reference a_{l_1} (omitting the type annotation) does not leak to another reference b_{l_2} that gives a read access to an unauthorized principal p, i.e., such that $p \in l_2$ but $p \notin l_1$. Reverse inclusion defines the allowed flows between security levels, allowing information to flow from level l_1 to level l_2 if and only if $l_1 \supseteq l_2$.

Given a security level l and a flow policy F, the upward closure of l w.r.t. F is the set $\{p \mid \exists q \in l \,.\, (q, p) \in F\}$ and is denoted by $l \uparrow_F$. Now denoting the reflexive and transitive closure of F by F^*, we can derive (as in [ML98, AB05]) a more permissive flow relation:

$$l_1 \preceq_F l_2 \overset{\text{def}}{\Leftrightarrow} \forall q \in l_2 . \exists p \in l_1 \,.\, p\ F^*\ q \;\Leftrightarrow\; (l_1 \uparrow_F) \supseteq (l_2 \uparrow_F)$$

This relation defines a lattice of security levels, where meet ($\sqcap_F$) and join ($\sqcup_F$) are given respectively by the union of the security levels and intersection of their upward closures with respect to F:

$$l_1 \sqcap_F l_2 = l_1 \cup l_2 \qquad l_1 \sqcup_F l_2 = (l_1 \uparrow_F) \cap (l_2 \uparrow_F)$$

Notice that $\preceq_F$ extends $\supseteq$ in the sense that $\preceq_F$ is larger than $\supseteq$ and that $\preceq_\emptyset = \supseteq$. We will use this mechanism of extending the flow relation with a flow policy F in the following way: the information flows that are allowed to occur in an expression M placed in a context $\mathsf{E}[]$ must satisfy the flow relation $\preceq_{\lceil \mathbf{E} \rceil}$.

3.2 The Non-disclosure Policy

In this section we define the non-disclosure policy for networks, which is based on a notion of bisimulation for sets of threads P with respect to a "low" security level. As usual, the bisimulation expresses the requirement that P_1 and P_2 are to be related if, when running over memories that coincide in their low part, they perform the same low changes. Then, if P is shown to be bisimilar to itself, one can conclude that the high part of the memory has not interfered with the low part, i.e., no security leak has occurred. Using the flow policies that were presented earlier, the notion of "being low" can be extended as in [AB05], thus weakening the condition on the behavior of the threads.

As we will see in Section 4, the position of a thread in the network can reveal information about the values in the memory. For this reason, we must use a notion of "low-equality" that is extended to states $\langle T, S\rangle$. The intuition is that a thread can access a reference if and only if it is located at the same domain as the thread that owns it. Threads that own low references can then be seen as "low threads". We are interested in states where low threads are co-located. Low-equality on states is defined pointwise, for a security level l that is considered as low:

$$\begin{aligned} S_1 =^{F,l} S_2 &\overset{\text{def}}{\Leftrightarrow} \forall a_{k,\theta} \in \mathsf{dom}(S_1) \cup \mathsf{dom}(S_2) \;.\; k \preceq_F l \Rightarrow \\ &\qquad a_{k,\theta} \in \mathsf{dom}(S_1) \cap \mathsf{dom}(S_2) \;\&\; S_1(a_{k,\theta}) = S_2(a_{k,\theta}) \\ T_1 =^{F,l} T_2 &\overset{\text{def}}{\Leftrightarrow} \forall n_k \in \mathsf{dom}(T_1) \cup \mathsf{dom}(T_2) \;.\; k \preceq_F l \Rightarrow \\ &\qquad n_k \in \mathsf{dom}(T_1) \cap \mathsf{dom}(T_2) \;\&\; T_1(n_k) = T_2(n_k) \end{aligned}$$

We say that two states are low-equal if they coincide in their low part (including their domains). This relation is transitive, reflexive and symmetric.

Now we define a bisimulation for networks, which can be used to relate networks with the same behavior over low parts of the states. In the following we denote by $\twoheadrightarrow$ the reflexive closure of the union of the transitions $\underset{F}{\longrightarrow}$, for all F.

Definition 1 (l-Bisimulation and $\approx_l$). *Given a security level l, we define an l-bisimulation as a symmetric relation $\mathcal{R}$ on sets of threads such that*

$$P_1 \;\mathcal{R}\; P_2 \;\&\; \langle T_1, P_1, S_1\rangle \underset{F}{\longrightarrow} \langle T_1', P_1', S_1'\rangle \;\&\; \langle T_1, S_1\rangle =^{F,l} \langle T_2, S_2\rangle \;\&\; (*) \textit{ implies:}$$
$$\exists T_2', P_2', S_2' : \langle T_2, P_2, S_2\rangle \twoheadrightarrow \langle T_2', P_2', S_2'\rangle \;\&\; \langle T_1', S_1'\rangle =^{\emptyset,l} \langle T_2', S_2'\rangle \;\&\; P_1' \;\mathcal{R}\; P_2'$$

where $(*) = \mathsf{dom}({S_1}' - S_1) \cap \mathsf{dom}(S_2) = \emptyset$ *and* $\mathsf{dom}({T_1}' - T_1) \cap \mathsf{dom}(T_2) = \emptyset$. *The relation $\approx_l$ is the greatest l-bisimulation.*

Intuitively, our security property states that, at each computation step performed by some thread in a network, the information flow that occurs respects the basic flow relation (empty flow policy), extended with the flow policy (F) that is declared by the context where the command is executed.

Definition 2 (Non-disclosure for Networks). *A set P of threads satisfies the* non-disclosure policy *if it satisfies $P \approx_l P$ for all security levels l.*

The non-disclosure definition differs from that of [AB05] in two points: first, the position of the low threads is treated as "low information"; second, for being independent from a single flow policy – as we have seen, each thread in the network may have its own flow policy.

4 The Type and Effect System

The type and effect system that we present here selects secure threads by ensuring the compliance of all information flows to the flow relation that rules in each point of the program. To achieve this, it constructively determines the *effects* of each expression, which contain information on the security levels of the references that the expression reads and writes, as well as the level of the references on which termination or non-termination of the computations might depend.

A key observation is that non-termination of a computation might arise from an attempt to access a *foreign* reference. In order to distinguish the threads that own each expression and reference, we associate unique identifiers $\bar{m}, \bar{n} \in \bar{\mathcal{N}}$ to names of already existing threads, as well as to the unknown thread name '?' for those that are created at runtime. It should be clear that information on which the position of a thread n might depend can leak to another that simply attempts to access one of n's references. For this reason, we associate to each thread a security level representing its "visibility" level, since just by owning a low reference, the position of a thread can be detected by "low observers".

Judgments have the form $\Sigma, \Gamma \vdash_G^{\bar{n}_l} M : s, \tau$, where Σ is a partial injective mapping from the set of decorated thread names extended with '?', and the set of decorated thread identifiers. The typing context Γ assigns types to variables. The expression M belongs to the thread that is statically identified by $\bar{n}_l$. The security level l is a lower bound to the references that the thread can own. The flow policy G is the one that is enforced by the context in which M is evaluated. The security effect s has the form $\langle s.r, s.w, s.t \rangle$, where $s.r$ is an upper bound on the security levels of the references that are read by M, $s.w$ is a lower bound on the that are written by M, and $s.t$ is an upper bound on those levels of the references on which the termination of expression M might depend. Finally, τ is the type of the expression, whose syntax is as follows, for any type variable t:

$$\tau, \sigma, \theta \quad ::= \quad t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathsf{ref}_{l, \bar{n}_k} \mid \tau \xrightarrow[\bar{n}_k, G]{s} \sigma$$

As expected, the reference type shows the reference's security level l and the type θ of the value that it points to; now we also add the identifier $\bar{n}$ and security level k of the thread that owns the reference. As for the function type, we have the usual latent parameters that are needed to type the body of the function.

The rules of the type system are shown in Figure 1. Whenever we have $\Sigma; \Gamma \vdash_G^{\bar{n}} M : \langle \bot, \top, \bot \rangle, \tau$, for all $\bar{n}, G$, we simply write $\Sigma; \Gamma \vdash M : \tau$. We also abbreviate meet and join with respect to the empty flow relation ($\sqcap_\emptyset$, $\sqcup_\emptyset$) by $\sqcap$, $\sqcup$, and $\langle s.r \sqcup s'.r, s.w \sqcap s'.w, s.t \sqcup s'.t \rangle$ by $s \sqcup s'$. We must now convince ourselves that the type system indeed selects only safe threads, according to the security notion defined in the previous section. We refer the reader to [AB05] for explanations regarding the use of flow policies in the typing rules. The usual intuitions on treating termination leaks can be useful to understand the new conditions regarding migration leaks. In fact, suspension of a thread on an access to an absent reference can be seen as a non-terminating computation that can be unblocked by migration of concurrent threads.

In rule Loc, the identifier of the thread name that owns the reference is obtained by applying Σ to the prefix of the address. In rule Ref, the reference

$$[\text{Nil}]\ \Sigma;\Gamma\vdash ():\mathsf{unit}\qquad [\text{Flow}]\ \frac{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G\cup F} M:s,\tau}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(\mathsf{flow}\ F\ \mathsf{in}\ M):s,\tau}$$

$$[\text{Abs}]\ \frac{\Sigma;\Gamma,x:\tau\vdash^{\bar{m}_j}_{G} M:s,\sigma}{\Sigma;\Gamma\vdash(\lambda x.M):\tau\xrightarrow[\bar{m}_j,G]{s}\sigma}\qquad [\text{Rec}]\ \frac{\Sigma;\Gamma,x:\tau\vdash W:\tau}{\Sigma;\Gamma\vdash(\varrho xW):\tau}$$

$$[\text{Var}]\ \Sigma;\Gamma,x:\tau\vdash x:\tau\qquad [\text{Loc}]\ \Sigma;\Gamma\vdash n_k.u_{l,\theta}:\theta\ \mathsf{ref}_{l,\Sigma(n_k)}$$

$$[\text{Ref}]\ \frac{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G} M:s,\theta\quad \begin{array}{r}j\preceq\ \ l\\ s.r,s.t\preceq_G l\end{array}}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(\mathsf{ref}_{l,\theta}\ M):s,\theta\ \mathsf{ref}_{l,\bar{m}_j}}\qquad [\text{Der}]\ \frac{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G} M:s,\theta\ \mathsf{ref}_{l,\bar{n}_k}}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(?\ M):s\sqcup\langle l,\top,\bar{t}\rangle,\theta}\ (*)$$

$$[\text{Ass}]\ \frac{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G} M:s,\theta\ \mathsf{ref}_{l,\bar{n}_k}\quad \Sigma;\Gamma\vdash^{\bar{m}_j}_{G} N:s',\theta\quad \begin{array}{r}s.t\preceq_G s'.w\\ s.r,s'.r,s.t,s'.t,j\preceq_G l\end{array}}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(M:=^{?}N):s\sqcup s'\sqcup\langle\bot,l,\bar{t}\rangle,\mathsf{unit}}\ (*)$$

$$(*)\ \textit{where}\ \bar{t}=(\mathsf{if}\ \bar{m}\neq\bar{n}\ \mathsf{then}\ k\sqcup j\ \mathsf{else}\ \bot)$$

$$[\text{BoolT}]\ \Sigma;\Gamma\vdash tt:\mathsf{bool}\qquad [\text{BoolF}]\ \Sigma;\Gamma\vdash f\!f:\mathsf{bool}$$

$$[\text{Cond}]\ \frac{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G} M:s,\mathsf{bool}\quad \Sigma;\Gamma\vdash^{\bar{m}_j}_{G} N_i:s_i,\tau\quad s.r\sqcup s.t\preceq_G s_1.w\sqcup s_2.w}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2):s\sqcup s_1\sqcup s_2\sqcup\langle\bot,\top,s.r\rangle,\tau}$$

$$[\text{App}]\ \frac{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G} M:s,\tau\xrightarrow[\bar{m}_j,G]{s'}\sigma\quad \Sigma;\Gamma\vdash^{\bar{m}_j}_{G} N:s'',\tau\quad \begin{array}{r}s.t\preceq_G s''.w\\ s.r,s''.r,s.t,s''.t\preceq_G s'.w\end{array}}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(M\ N):s\sqcup s'\sqcup s''\sqcup\langle\bot,\top,s.r\sqcup s''.r\rangle,\sigma}$$

$$[\text{Seq}]\ \frac{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G} M:s,\tau\quad \Sigma;\Gamma\vdash^{\bar{m}_j}_{G} N:s',\sigma\quad s.t\preceq_G s'.w}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(M;N):s\sqcup s',\sigma}$$

$$[\text{Thr}]\ \frac{\begin{array}{c}j\preceq_G l\\ \bar{n}\ \textit{fresh in}\ \Sigma\end{array}\quad \Sigma,?:\bar{n}_l;\Gamma\vdash^{\bar{n}_l}_{G} M:s,\mathsf{unit}}{\Sigma;\Gamma\vdash^{\bar{m}_j}_{G}(\mathsf{thread}_l\ M):\langle\bot,s.w\sqcap l,\bot\rangle,\mathsf{unit}}$$

$$[\text{Mig}]\ \Sigma;\Gamma\vdash^{\bar{m}_j}_{G}\mathsf{goto}\ d:\langle\bot,j,\bot\rangle,\mathsf{unit}$$

Fig. 1. Type system

that is created belongs to the thread identified by the superscript of the '$\vdash$'. We check that the security level that is declared for the new reference is greater than the level of the thread. In rule Thr, a fresh identifier – image of an unknown thread name represented by '?' – is used to type the thread that is created. The new thread's security level must preserve that of the parent thread.

In rule Mig we add the security level of the thread to the write effect to prevent migrations of threads owning low references to depend on high information. The motivation for this is that the mere arrival of a thread and its references to another domain might trigger the execution of other threads that were suspended on an access to a low reference, as in the following program:

$$d_1[\{(\mathsf{if}\ (?\ n_1.x_H)\ \mathsf{then\ goto}\ d_2\ \mathsf{else}\ ())^{n_1}\},\{n_1.y_L\mapsto 1\}]\ \|\ d_2[\{(n_1.y_L:=^{?}0)^{n_2}\},\emptyset]$$

Notice that the rule Cond rejects thread n in a standard manner, since $H\not\preceq L$.

In rules DER and ASS, the termination effect is updated with the level of the thread that owns the foreign reference we want to access. Without this restriction, suspension on an access to an absent reference could be unblocked by other threads, as is illustrated by the following example:

$$\begin{gathered} d[\,(\mathsf{if}\ a_H\ \mathsf{then}\ (\mathsf{goto}\ d_1)\ \mathsf{else}\ (\mathsf{goto}\ d_2))^{m_j}, \{m_j.x_\top \mapsto 42\}\,] \parallel \\ \parallel d_1[\,((m_j.x_\top :=^? 0); (n_{1k_1}.y_L :=^? 0))^{n_{1k_1}}, S_1\,] \parallel \\ \parallel d_2[\,((m_j.x_\top :=^? 0); (n_{2k_2}.y_L :=^? 0))^{n_{2k_2}}, S_2\,] \end{gathered}$$

Then, depending on the value of the high reference a, different low assignments would occur to the low references $n_1.y_L$ and $n_2.y_L$. To see why we can take j for preventing the leak from a_H to $n_1.y_L, n_2.y_L$, notice that (by MIG and COND) $H \preceq j$. The same example can show a potential leak of information about the positions of the threads n_1 and n_2 via their own low variables $n_1.y_L, n_2.y_L$. This also accounts for updating the termination level of the assignments $(m_j.x_\top :=^? 0)$ and $(m_j.x_\top :=^? 0)$ with the security levels k_1 and k_2, respectively.

The previous example shows how migration of a thread can result in an information leak from a high variable to a lower one via an "observer" thread. It is the ability of the observer thread to detect the presence of the first thread that allows the leak. However, one must also prevent the case where it is the thread itself that reveals that information, like in the following simple example:

$$d[\,(n.u_L :=^? 0)^{m_j}, \emptyset\,]$$

This program is insecure if $j \not\preceq L$, and it is rejected by the condition $j \preceq_G l$ in rule ASS. Notice that, in the typing rule, for the cases where $m = n$ the condition is satisfied anyway due to the meaning of j.

We now give a safety property of our type system:

Theorem 1 (Subject reduction). *If $\Sigma; \Gamma \vdash_G^{\Sigma(m_j)} M : s, \tau$ and $\langle T, M^{m_j}, S\rangle \xrightarrow[F]{N^{n_l}} \langle T', M'^{m_j}, S'\rangle$, then $\exists s'$ such that $\Sigma; \Gamma \vdash_G^{\Sigma(m_j)} M' : s', \tau$, where $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$. Furthermore, if $N^{n_l} \neq ()$, then $\exists \bar{n}, s''$ such that $\Sigma, ? : \bar{n}_l; \Gamma \vdash_G^{\bar{n}_l} N : s'', \mathsf{unit}$ where $\bar{n}$ is fresh in Σ, $j \preceq_G l$ and $s.w \preceq s''.w$.*

This result states that computation preserves the type of threads, and that as the effects of an expression are performed, the security effects of the thread "weaken". We now state the main result of the paper, saying that our type system only accepts threads that can securely run in a network with other typable threads.

Theorem 2 (Soundness). *Consider a set of threads P and an injective mapping Σ from decorated thread names to decorated thread identifiers, such that $\mathsf{dom}(\Sigma) = \mathsf{tn}(P)$. If for all $M^{m_j} \in P$ we have that $\exists \Gamma, s, \tau\ .\ \Sigma; \Gamma \vdash_\emptyset^{\Sigma(m_j)} M : s, \tau$, then P satisfies the non-disclosure policy for networks.*

This result is compositional, in the sense that it is enough to verify the typability of each thread separately in order to ensure non-disclosure for the whole network. Having the empty set as the flow policy of the context means that there is no global flow policy that encompasses the whole network. One could easily prove non-disclosure with respect to a certain global flow policy G by requiring the typability of all the threads with respect to G. However, by choosing the empty global flow policy we stress the decentralized nature of our setting.

5 Conclusion and Related Work

To the best of our knowledge, this paper is the first to study insecure information flows that are introduced by mobility in the context of a distributed language with states. We have identified a new form of security leaks, the *migration leaks*, and provided a sound type system for rejecting them. The discussion on related work will focus on type-based approaches for enforcing information flow control policies in settings with concurrency, distribution or mobility.

A first step towards the study of confidentiality for distributed systems is to study a language with concurrency. Smith and Volpano [SV98] proved non-interference for an imperative multi-threaded language. They identified the *termination leaks* that appear in concurrent contexts but that are not problematic in sequential settings. This line of study was pursued by considering increasingly expressive languages and refined type systems [Smi01, BC02, AB05, Bou05]. In the setting of synchronous concurrent systems, new kinds of termination leaks – the *suspension leaks* – are to be handled. A few representative studies include [Sab01, ABC04]. Already in a distributed setting, but restricting interaction between domains to the exchange of values (no code mobility), Mantel and Sabelfeld [SM02] provided a type system for preserving confidentiality for different kinds of channels over a publicly observable medium.

Progressing rather independently we find a field of work on mobile calculi based on purely functional concurrent languages. To mention a few representative papers, we have Honda *et al.*'s work on for π-calculus [HVY00], and Hennessy and Riely's study for the security π-calculus [HR00]. The closest to the present work is the one by Bugliesi *et al.* [CBC02], for Boxed Ambients [BCC01], a purely functional calculus based on the mobility of *ambients*. Since ambient names correspond simultaneously to places of computation, subjects of migration, and channels for passing values, it is hard to establish a precise correspondence between the two type systems. Nevertheless, it is clear that the knowledge of the position of an ambient of level l is considered as l-level information, and that migration is also identified as a way of leaking the positions of ambients, though the dangerous usages of migration are rejected rather differently.

Sharing our aim of studying the distribution of code under decentralized security policies, Zdancewic *et al.* [ZZNM02] have however set the problem in a very different manner. They have considered a distributed system of potentially corrupted hosts and of principals that have different levels of trust on these hosts. They then proposed a way of partitioning the program and distributing the resulting parts over hosts that are trusted by the concerned principals.

Acknowledgments

I would like to thank Gérard Boudol, Ilaria Castellani, Jan Cederquist, Matthew Hennessy, Tamara Rezk and the anonymous referees for insightful comments at different stages of this work. This research was partially funded by the PhD scholarship POSI/SFRH/BD/7100/2001 and by the French ACI Project CRISS.

References

[AB05] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *CSFW*, 2005.

[ABC04] A. Almeida Matos, G. Boudol, and I. Castellani. Typing noninterference for reactive programs. In *FCS*, volume 31 of *TUCS General Publications*, 2004.

[BC02] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.

[BCC01] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS*, volume 2215 of *LNCS*, 2001.

[BCGL02] G. Boudol, I. Castellani, F. Germain, and M. Lacoste. Analysis of formal models of distribution and mobility: state of the art. Mikado D1.1.1, 2002.

[Bou04] G. Boudol. ULM, a core programming model for global computing. In *ESOP*, volume 2986 of *LNCS*, 2004.

[Bou05] G. Boudol. On typing information flow. In *ICTAC*, LNCS, 2005.

[CBC02] S. Crafa, M. Bugliesi, and G. Castagna. Information flow security for boxed ambients. In *F-WAN*, volume 66(3) of *ENTCS*, 2002.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy*, 1982.

[HR00] M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous pi-calculus. In *ICALP'00*, volume 1853 of LNCS, 2000.

[HVY00] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP*, volume 1782 of LNCS, 2000.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.

[ML98] A. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Symposium on Security and Privacy*, 1998.

[Sab01] A. Sabelfeld. The impact of synchronization on secure information flow in concurrent programs. In *Andrei Ershov International Conference on Perspectives of System Informatics*, 2001.

[SM02] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *SAS*, volume 2477 of LNCS, 2002.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *Journal on Selected Areas in Communications, 21(1)*, 2003.

[Smi01] Geoffrey Smith. A new type system for secure information flow. In *CSFW*, 2001.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW*, 2005.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, 1998.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 1996.

[ZZNM02] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions in Computer Systems*, 20(3):283–328, 2002.