Distributed Noninterference

Ana Almeida Matos and Jan Cederquist
Instituto de Telecomunicações (SQIG) and Universidade de Lisboa (IST)
Lisbon, Portugal

***Abstract*—Noninterference is the classic information flow property that establishes the absence of illegal information flows. Legality of flows is originally defined with respect to a single security setting that is based on a security lattice that orders security levels according to their confidentiality and/or integrity. This paper proposes a natural generalization of noninterference to a distributed security setting where each computation domain establishes its own local security lattice. Referred to as *distributed noninterference* (DNI), the new security property implies that information flows respect the allowed flow policy of the domains where they are computed. The semantic coherence between DNI and other information flow related properties for distributed settings is established. We present a type and effect system that enforces DNI for an expressive distributed higher-order lambda-calculus with imperative features and code migration.**

***Index Terms*—information flow; noninterference; distribution**

I. Introduction

Today's most powerful and widely used computation technologies run on parallel, distributed, and network-oriented arrangements of processing units, specializing in tasks that range from large scale data processing, to light mobile diffuse services. These technologies are often interconnected and layered into continuously evolving computing platforms that are controlled by parties which might not trust each other, forming a moving and complex target that defy efforts directed at ensuring security. The clarification of how the security principles of confidentiality and integrity apply to computations that involve multiple computation domains is a crucial foundational step towards the understanding of security of real world mobile and distributed applications.

Language-based techniques for ensuring information flow security offer appealing abstractions for achieving high levels of assurance in software applications [13]. Information flow security regards the control of how data of different security levels can influence the observable behavior of program execution. The classic property of *noninterference* establishes that confidentiality is only preserved by programs when information that is labeled with a given security level never influences that of lower or incomparable levels [10]. The relative degree of confidentiality is usually determined with respect to a single security lattice [8]. However, in a distributed scenario, multiple security policies are established independently by different parties. The question of what is a noninterferent program running in a distributed security setting remains open.

Let us consider the example of an organization whose network of computing resources is organized into a grid. Workloads are distributed throughout the connected devices in the form of mobile code for remote evaluation. Each device forms a *computation domain* with specific capabilities and restrictions, and in particular *allowed information flow policies* for protecting data and other computing *threads* that are running concurrently in the same domain. The distribution of tasks must take these policies into account, for domains should not execute code that perform information leaks that do not comply to them. In order to specify this intuitive security property we need a formal counterpart to noninterference that is suitable for distributed settings.

This paper proposes a generalized definition of noninterference that accommodates the reality of that distributed and mobile programs must obey different security settings at different points of their computation, depending on their location. It is formalized for a simple and general network model where computation domains are units of abstract allowed information flow policies. Our main results include: (1) A new information flow property, named *distributed noninterference*, that naturally generalizes classical noninterference to distributed security settings. (2) A study of the semantic coherence between the proposed definition of distributed noninterference and the properties of (local) noninterference, non-disclosure for networks and flow policy confinement. (3) A type and effect system for statically enforcing distributed noninterference for an expressive distributed higher-order imperative λ-calculus.

A full version of this article is available from the authors.

II. Setting

Security Setting: The study of confidentiality traditionally relies on a lattice of security levels [8], corresponding to reading clearances that are associated to information holders in the programming language. The idea is that information pertaining to references labeled with l_2 can influence references labeled with l_1 only if l_1 is at least as confidential as l_2. *Flow policies* can be used for relaxing the basic security lattice, by establishing additional legal flow directions between security levels. When formalized as downward closure operators on security lattices, they collapse security levels of a basic lattice into lower ones [4]. In this view, flow policies can also be ordered into a lattice according to their permissiveness.

More formally, we assume a *basic security lattice* $\mathcal{L} = \langle \boldsymbol{Lev}, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$ of confidentiality levels $l, j \in \boldsymbol{Lev}$ with the usual meaning, and consider flow policies $A, F \in \boldsymbol{Flo}$ that are downward closure operators $F : \boldsymbol{Lev} \to \boldsymbol{Lev}$ on $\mathcal{L}$ (i.e. they are monotone, idempotent and restrictive). The security lattice that results from the action of F over $\mathcal{L}$, written $\langle \boldsymbol{Lev}, \sqsubseteq^F, \sqcap^F, \sqcup^F, \top^F, \bot^F \rangle$ satisfies: $l_1 \sqsubseteq^F l_2$ if $F(l_1) \sqsubseteq F(l_2)$, where $\sqsubseteq^F$ represents the information flows

that are allowed by F; furthermore, $l_1 \sqcup^F l_2 = l_1 \sqcup l_2$, $l_1 \sqcap^F l_2 = F(l_1 \sqcap l_2)$, $\bot^F = \bot$ and $\top^F = F(\top)$. We refer to information flows that do not comply to $\mathcal{L}$ as information *leaks*, while those that do not comply to a given flow policy A, are considered *illegal with respect to* A.

Flow policies form themselves a lattice $\langle \textbf{\textit{Flo}}, \preccurlyeq, \curlywedge, \curlyvee, \mho, \Omega \rangle$ with the following meaning: $F_1 \preccurlyeq F_2$ means that F_1 is at least as permissive as F_2; 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.

Language Setting: Networks are flat juxtapositions of domains, each containing a store and a pool of threads, which are subjected to the local allowed flow policy of the domain. The basic elements of the language are then references, threads and domains, whose names are drawn from disjoint countable sets $a, b \in \textbf{\textit{Ref}}$, $m, n \in \textbf{\textit{Nam}}$, and $d \in \textbf{\textit{Dom}} \neq \emptyset$, respectively. References are information containers to which values $V \in \textbf{\textit{Val}}$ of the language are assigned by means of *stores* $S : \textbf{\textit{Ref}} \to \textbf{\textit{Val}}$ that map reference names to values. Threads run concurrently in *pools* $P : \textbf{\textit{Nam}} \to \textbf{\textit{Exp}}$, which map thread names to expressions $m, n, \in \textbf{\textit{Exp}}$ (denoted as sets of threads), and their positions in the network are tracked by *position-trackers* $T : \textbf{\textit{Nam}} \to \textbf{\textit{Dom}}$. The *allowed-policy mapping* $W : \textbf{\textit{Dom}} \to \textbf{\textit{Flo}}$ maps the name of each domain to its allowed flow policy, which is considered fixed in this model.

Evaluation is defined over *configurations* $\langle P, T, S \rangle$. We refer to the pairs $\langle S, T \rangle$ as *states*, and pairs $\langle P, T \rangle$ as *thread configurations*. It is parameterized by means of the '$W \vdash$' turnstile, which fixes the allowed flow policy of each domain in the network, and is implicitly parameterized by the *reference labeling* $\Sigma : \textbf{\textit{Ref}} \to \textbf{\textit{Lev}} \times \textbf{\textit{Typ}}$, whose left projection Σ_1 determines the security level of each reference name, and right projection Σ_2 determines the type of values can be assigned to each reference name, and the *thread labeling* $\Upsilon : \textbf{\textit{Nam}} \to \textbf{\textit{Lev}}$, that determines the security level of each thread name. The transitions $\xrightarrow[F]{d}$ are decorated with the name of the domain d where the step takes place and the flow policy F declared by the evaluation context. The semantics does not depend on this information, which is used for the purpose of the security analysis. The relation $\twoheadrightarrow$ denotes the reflexive closure of $\xrightarrow[F]{d}$.

In forthcomming examples, and as a target language for the type system, we use an expressive ML-like language that is extended with basic distribution and code mobility features, such as the one in [3]. Declassification is introduced by means of *flow policy declarations* [1] of the form $(\text{flow } F \text{ in } M)$. They are used to locally weaken the information flow policy that is declared by the evaluation context, by enabling flows that comply to F within its lexical scope M. Programs can inspect the allowed policy of the current domain by means of the *allowed-condition*, written $(\text{allowed } F \text{ then } N_t \text{ else } N_f)$. The construct tests whether F is allowed and chooses branches N_t or N_f accordingly, in practice offering alternative behaviors in case the domain is too restrictive. The remote thread creator $(\text{thread}_l\ M\ \text{at}\ d)$ spawns a new thread with expression M at domain d, to be executed concurrently with other threads. It functions as a migration construct when the destination domain differs from the current one. The confidentiality level l is associated to the position of the new thread in the network.

III. Security Property

We now formulate Distributed Noninterference in terms of an information flow bisimulation [6], which provides a natural way of relating concurrent programs according to their behavior over "low" parts of the state, and illustrate it with examples.

Low-equality: The notion of low-equality is defined between states. It includes position trackers because the position of a thread in the network can reveal information about the values in the memory (e.g. program III).

Definition 3.1 ($=^{\Sigma,\Upsilon}_{F,l}$): Given reference and thread labelings Σ, Υ, two states $\langle T_1, S_1 \rangle$ and $\langle T_2, S_2 \rangle$ are said to be low-equal with respect to a flow policy F and a security level l, written $\langle T_1, S_1 \rangle =^{\Sigma,\Upsilon}_{F,l} \langle T_2, S_2 \rangle$, if for every reference name $a \in \textbf{\textit{Ref}}$, if $\Sigma_1(a) \sqsubseteq^F l$ then $S_1(a) = S_2(a)$, and for every thread name $n \in \textbf{\textit{Nam}}$, if $\Upsilon(n) \sqsubseteq^F l$ then $T_1(n) = T_2(n)$.
Low-equality is an equivalence relation. Note that if $F_1 \preccurlyeq F_2$, then $\langle T_1, S_1 \rangle =^{\Sigma,\Upsilon}_{F_1,l} \langle T_2, S_2 \rangle$ implies $\langle T_1, S_1 \rangle =^{\Sigma,\Upsilon}_{F_2,l} \langle T_2, S_2 \rangle$.

Store compatibility: When a higher-order language is considered, 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 with the relevant security goals by means of a typability condition. We postpone the choice of the predicate for (W, Σ, Γ)-compatibility to the end of Section V.

Distributed Noninterference: The property is defined as a bisimulation between thread configurations. The location of a thread determines which allowed flow policy it should obey at that point, and is used to place a restriction on the information flows that occur at that step. In the first low-equality of the following definition, $W(d)$ specifies how information that is *read* during that step is allowed to flow in future steps.

Definition 3.2 ($\sim^{\Sigma,\Upsilon}_{\Gamma,l}$): Given an allowed-policy mapping W, reference and thread labelings Σ, Υ, and a typing environment Γ, a $(\Sigma, \Upsilon, \Gamma, l)$-bisimulation is a symmetric relation $\mathcal{R}$ on thread configurations that satisfies, for all P_1, T_1, P_2, T_2, and (W, Σ, Γ)-compatible stores S_1, S_2:

$\langle P_1, T_1 \rangle\ \mathcal{R}\ \langle P_2, T_2 \rangle$ and $W \vdash \langle P_1, T_1, S_1 \rangle \xrightarrow[F]{\boldsymbol{d}} \langle P_1', T_1', S_1' \rangle$ and
$\langle T_1, S_1 \rangle =^{\Sigma,\Upsilon}_{\boldsymbol{W(d)},l} \langle T_2, S_2 \rangle$ and $(\text{dom}(S_1') \backslash \text{dom}(S_1)) \cap \text{dom}(S_2) = \emptyset$
and $(\text{dom}(T_1') \backslash \text{dom}(T_1)) \cap \text{dom}(T_2) = \emptyset$
implies that $\exists P_2', T_2', S_2'$ s.t.: $W \vdash \langle P_2, T_2, S_2 \rangle \twoheadrightarrow \langle P_2', T_2', S_2' \rangle$ and
$\langle T_1', S_1' \rangle =^{\Sigma,\Upsilon}_{\mho,l} \langle T_2', S_2' \rangle$ and $\langle P_1', T_1' \rangle\ \mathcal{R}\ \langle P_2', T_2' \rangle$

Furthermore, S_1', S_2' are still (W, Σ, Γ)-compatible. The largest $(\Sigma, \Upsilon, \Gamma, l)$-bisimulation is denoted $\sim^{\Sigma,\Upsilon,}_{\Gamma,l}$.
For any Σ, Υ, l, the set of pairs of thread configurations where threads are values is an $(\Sigma, \Upsilon, \Gamma, l)$-bisimulation. Furthermore, the union of a family of $(\Sigma, \Upsilon, \Gamma, l)$-bisimulations is a $(\Sigma, \Upsilon, \Gamma, l)$-bisimulation. Consequently, $\sim^{\Sigma,\Upsilon,}_{\Gamma,l}$ exists.

Information flows that take place at each step are captured by quantifying over all possible pairs of stores that coincide in the observable level. This enables compositionality of the property, accounting for the possible changes to the store that are induced by external threads. Position trackers of configurations are fixed across steps within the bisimulation game, reflecting the assumption that changes in the position of a thread can only be induced by the thread itself (*subjective* migration). Note that the above relation is not reflexive. In fact, only secure programs are related to themselves:

Definition 3.3 (Distributed Noninterference): A pool P satisfies Distributed Noninterference with respect to an allowed-policy mapping W, reference labelings Σ, Υ and typing environment Γ, if $\langle P, T_1\rangle \sim_{\Gamma,l}^{\Sigma,\Upsilon} \langle P, T_2\rangle$ for all security levels l and position trackers T_1, T_2 s.t. $\mathrm{dom}(P) = \mathrm{dom}(T_1) = \mathrm{dom}(T_2)$ and $T_1 =_{\mho,l}^{\Sigma,\Upsilon} T_2$. We then write $P \in \mathcal{DNI}(W, \Sigma, \Upsilon, \Gamma)$.
Distributed noninterference (DNI) is compositional by set union of pools of threads, up to disjoint naming of threads.

Noninterference: As expected, local noninterference follows from Definition 3.3 when networks are collapsed into a single domain d^*, i.e. when $\boldsymbol{Dom} = \{d^*\}$. The resulting property is parameterized by $W(d^*)$, and coincides with the view of noninterference as the *absence of information leaks* (in the sense defined in Section II) when $W(d^*) = \mho$. However, here we adopt the view of noninterference as the *absence of illegal flows*, which is relative to the particular allowed flow policy. This point is further discussed in Section VI.

Examples: Programs that violate noninterference, such as the direct leak $(b := (!\ a))$ when running at d such that $\Sigma_1(a) \not\sqsubseteq^{W(d)} \Sigma_1(b)$ are also insecure with respect to DNI. The same holds for indirect leaks via control and termination.

In the following program, information regarding reference a flows to b via observation of the destination domain:

$$\begin{array}{l}(\text{if } (!\ a) \text{ else } (\text{thread}_l\ (\text{allowed } F \text{ then } (b := 0) \text{ else } ()) \text{ at } d_2)\\ \quad \text{then } (\text{thread}_l\ (\text{allowed } F \text{ then } (b := 1) \text{ else } ()) \text{ at } d_1))\end{array} \quad (1)$$

In fact, since the position of threads in the network is part of the observable state, a *migration leak* [2] from level $\Sigma_1(a)$ to level l occurs as soon as the new thread is created. The information is read before migration, so the leak is only secure if it is allowed by the policy of the thread's initial domain.

Let us now consider the simpler program to be executed at d such that $\Sigma_1(a) \sqsubseteq^{W(d)} l$ (where the migration leak is allowed):

$$\begin{array}{l}(\text{if } (!\ a) \text{ then}\\ \quad (\text{thread}_l\ (b := 0) \text{ at } d_1) \text{ else } (\text{thread}_l\ (b := 1) \text{ at } d_1))\end{array} \quad (2)$$

Since the two threads produce different changes at the level of b, then the program is again secure only if the first domain d allows it, i.e. $\Sigma_1(a) \sqsubseteq^{W(d)} \Sigma_1(b)$. The policy of the new domain d_1 only rules over what happens from the point where the thread enters it. Since the behavior of the code that actually migrates does not depend on the location of the corresponding threads, no leak is taking place at this point of the program.

Notice that the declassification operations are transparent to the property. In particular, program

$$(\text{thread}_l\ (\text{flow } F \text{ in } (b := (!\ a))) \text{ at } d) \quad (3)$$

violates DNI, regardless of F, if $\Sigma_1(a) \not\sqsubseteq^{W(d)} \Sigma_1(b)$.

IV. Semantic Coherence

This section shows that Distributed Noninterference is coherent with the properties of Noninterference, Non-Disclosure for Networks (NDN) and Flow Policy Confinement (FPC).

Non-Disclosure for Networks: Non-disclosure states that, at each step performed by a program, information flows respect the flow policy that is declared by the current evaluation context. The property is naturally defined by means of a bisimulation that relates the outcomes of each possible step that is performed over fresh states that coincide on their "low" region, where the notion of "low" is customized with the currently declared flow policy [1], [2]. Similarly to the definition of DNI, when migration is subjective, resetting the position tracker arbitrarily is unnecessary. The following program is intuitively secure (regarding NDN) if $W(d) \preccurlyeq F$

$$(\text{thread}_l\ (\text{allowed } F \text{ then } () \text{ else } M_{\mathit{insec}}) \text{ at } d) \quad (4)$$

as the body of the thread is known to be executed at domain d. However, if M_{insec} violates NDN, it is considered insecure by the definition in [2], as it covers the execution of the allowed condition at "fresh" locations where F is possibly *not* allowed. It is then reasonable to relax the power of the attacker, by focusing on the behavior of threads when coupled with their possible locations on the network.

Definition 4.1 ($\approx_{\Gamma,l}^{\Sigma,\Upsilon}$): Consider an allowed-policy mapping W, reference and thread labelings Σ, Υ, and a typing environment Γ. A $(\Sigma, \Upsilon, \Gamma, l)$-bisimulation is a symmetric relation $\mathcal{R}$ on thread configurations that satisfies, for all P_1, T_1, P_2, T_2, and (W, Σ, Γ)-compatible stores S_1, S_2:

$\langle P_1, T_1\rangle\ \mathcal{R}\ \langle P_2, T_2\rangle$ and $W \vdash \langle P_1, T_1, S_1\rangle \xrightarrow[\boldsymbol{F}]{d} \langle P_1', T_1', S_1'\rangle$ and
$\langle T_1, S_1\rangle =_{\boldsymbol{F},l}^{\Sigma,\Upsilon} \langle T_2, S_2\rangle$ and $(\mathrm{dom}(S_1') \backslash \mathrm{dom}(S_1)) \cap \mathrm{dom}(S_2) = \emptyset$
and $(\mathrm{dom}(T_1') \backslash \mathrm{dom}(T_1)) \cap \mathrm{dom}(T_2) = \emptyset$
implies that $\exists P_2', T_2', S_2'$ s.t.: $W \vdash \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle$ and
$\langle T_1', S_1'\rangle =_{\mho,l}^{\Sigma,\Upsilon} \langle T_2', S_2'\rangle$ and $\langle P_1', T_1'\rangle\ \mathcal{R}\ \langle P_2', T_2'\rangle$

Furthermore, S_1', S_2' are still (W, Σ, Γ)-compatible. The largest $(W, \Sigma, \Upsilon, \Gamma, l)$-bisimulation is denoted $\approx_{\Gamma,l}^{\Sigma,\Upsilon}$.
For analogous reasons as for Definition 3.2, $\approx_{\Gamma,l}^{\Sigma,\Upsilon}$ exists.

We now present a weakened version of non-disclosure for networks, that is defined over thread configurations.

Definition 4.2 (Non-disclosure for Networks): A pool P satisfies the Non-disclosure for Networks property with respect to an allowed-policy mapping W, a reference labeling Σ, a thread labeling Υ and a typing environment Γ, if it satisfies $\langle P, T_1\rangle \approx_{\Gamma,l}^{\Sigma,\Upsilon} \langle P, T_2\rangle$ for all security levels l and position trackers T_1, T_2 such that $\mathrm{dom}(P) = \mathrm{dom}(T_1) = \mathrm{dom}(T_2)$ and $T_1 =_{\mho,l}^{\Sigma,\Upsilon} T_2$. We then write $P \in \mathcal{NDN}(W, \Sigma, \Upsilon, \Gamma)$.

Definition 4.2 is strictly weaker than the old thread pool-based definition, as is illustrated by Example 4 above.

Notice that this property concerns only the match between flow declarations and the leaks that are encoded in the program. It does not restrict the usage of flow declarations.

Flow Policy Confinement: Flow Policy Confinement states that the declassifications that are declared by a program at each computation step complies to the allowed policy of the domain where the step is performed. Similarly to [3] we

define the property co-inductively, on thread configurations. 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.

Definition 4.3 ((W,Σ,Γ)-Confined Thread Configurations): Given an allowed-policy mapping W, a reference labeling Σ, and a typing environment Γ, a set $\mathcal{C}$ of thread configurations is a set of (W,Σ,Γ)-*confined thread configurations* if it satisfies, for all P, T, and (W,Σ,Γ)-compatible stores S:

$$\langle P,T\rangle \in \mathcal{C} \text{ and } W \vdash \langle P,T,S\rangle \xrightarrow[F]{d} \langle P',T',S'\rangle \text{ implies}$$
$$W(d) \preccurlyeq F \text{ and } \langle P',T'\rangle \in \mathcal{C}$$

Furthermore, S' is still (W,Σ,Γ)-compatible. The largest set of (W,Σ,Γ)-confined thread configurations is denoted $\mathcal{C}^{\Sigma,\Gamma}_W$. For analogous reasons as for Definition 3.2, $\mathcal{C}^{\Sigma,\Gamma}_W$ exists.

Definition 4.4 (Flow Policy Confinement): A pool of threads P satisfies Flow Policy Confinement with respect to an allowed-policy mapping W, a reference labeling Σ and a typing environment Γ, if all thread configurations satisfy $\langle P,T\rangle \in \mathcal{C}^{\Sigma,\Gamma}_W$. We then write $P \in \mathcal{FPC}(W,\Sigma,\Gamma)$.

Notice that this property speaks strictly about what *flow declarations* a thread can do *while* it is at a specific domain. It does not deal with information flows.

Combined: While NDN establishes a match between the leaks that are performed by a program and the declassifications that are declared in its code, FPC requires that the declassifications comply to the allowed flow policy of the domain where they occur. It is thus expected that a notion of DNI, which ensures that leaks respect the relevant allowed flow policies, should follow from the combination of the other two.

Theorem 4.5:
$\mathcal{NDN}(W,\Sigma,\Upsilon,\Gamma) \cap \mathcal{FPC}(W,\Sigma,\Gamma) \subseteq \mathcal{DNI}(W,\Sigma,\Upsilon,\Gamma)$.
To see that the sets are not equal, consider again program 3. While it is secure regarding DNI if $H \sqsubseteq^{W(d)} L$:

- if $F = \mho$ it violates NDN, but respects FPC, and
- if $F = \Omega$ it respects NDN, but violates FPC if $W(d) \not\preccurlyeq F$.

V. Type and effect system

It is clear that DNI is guaranteed by combining enforcement mechanisms for NDN and FPC. We argue in Section VI for the practical advantages of using that indirect approach. This section presents a type and effect system [11] that checks DNI, guaranteeing that information flows always comply to the current domain's allowed flow policy. Seen as a set of syntax-based rules that ensure security in a program, it contributes to clarifying the meaning of security of programs.

In a setting where code can migrate at runtime, the imposed allowed flow policy might change dynamically. This can happen in particular within the branch of an allowed condition:

$$(\text{allowed } F \text{ then } (\text{thread}_l\ M_1 \text{ at } d) \text{ else } M_2) \qquad (5)$$

Although M_1 is chosen only when F is allowed by the initial domain, by the time it is executed its location might be different. Analyzing the information leaks that occur in an expression then requires tracking the possible locations where threads might be at each point.

$$\text{[FLOW]}\ \frac{W;\Gamma \vdash^{\Sigma}_{j,A} N : s,\tau}{W;\Gamma \vdash^{\Sigma}_{j,A} (\text{flow } F \text{ in } N) : s,\tau}$$

$$\text{[ALLOW]}\ \frac{\begin{array}{c} W;\Gamma \vdash^{\Sigma}_{j,A\sqcap F} N_t : s_t,\tau \\ W;\Gamma \vdash^{\Sigma}_{j,A} N_f : s_f,\tau \end{array} \qquad j \sqsubseteq^A s_t.w, s_f.w}{W;\Gamma \vdash^{\Sigma}_{j,A} (\text{allowed } F \text{ then } N_t \text{ else } N_f) : s_t \sqcup s_f \sqcup \langle \bot,\top,j\rangle,\tau}$$

$$\text{[MIG]}\ \frac{\boldsymbol{W;\Gamma \vdash^{\Sigma}_{l,W(d)} M : s, \mathsf{unit}}}{\boldsymbol{W;\Gamma \vdash^{\Sigma}_{j,A} (\mathbf{thread}_l\, M \text{ at } d) : \langle \bot, l \sqcup s.w, \bot\rangle, \mathsf{unit}}}$$

Figure 1. Type and effect system for Distributed Noninterference (fragment)

Figure 1 presents a fragment of a new type and effect system for enforcing distributed noninterference over a migrating program. It guarantees that when information flows are performed by a thread, they comply to the allowed flow policy of the current domain. The typing judgments have the form

$$W;\Gamma \vdash^{\Sigma}_{j,A} M : s,\tau$$

meaning that expression M is typable with type $\tau \in \textbf{\textit{Typ}}$ and security effect s in typing context $\Gamma : \textbf{\textit{Var}} \to \textbf{\textit{Typ}}$, which assigns types to variables. In addition to the mapping W of domain names to allowed flow policies, the turnstile has as parameter the reference mapping Σ, the allowed flow policy A of the domain where the expression is to be executed, and the confidentiality level j of the location of M's thread. The security effect s is composed of three security levels: $s.r$ is an upper-bound on the levels of the references that are *read* by M; $s.w$ is a lower bound on the levels of the references that are *written* by M; $s.t$ is an upper bound on the level of the references on which the *termination* of M might depend. Accordingly, the reading and termination effects are composed in a covariant way, whereas the writing effect is contravariant.

Our type and effect system requires compliance of all information flows to the flow relation $\sqsubseteq^A$ that is determined by the allowed flow policy A of the current domain (see Section II). The conditions enforce standard syntactic rules of the kind "no low writes should depend on high reads", both with respect to the values that are read, and to termination behaviors that might be derived. We present only the rules that differ from the type system for NDN, and refer the reader to [2] for further explanations. While the FLOW rule is transparent in this type system, MIG sets the allowed policy parameter that is used for typing the new thread with that of the destination domain. The restriction in ALLOW ensures that the security level associated to the position of the thread in the network is at least as permissive as the writing effects of the branches, in order to prevent migration leaks. The requirements on the typability of the first branch are weakened with the tested policy F which is known to be valid in that case.

Soundness: The above type and effect system guarantees security of networks with respect to DNI:

Theorem 5.1 (Soundness): Consider a fixed allowed-policy mapping W, reference and thread labelings Σ,Υ, a typing environment Γ, and a thread configuration $\langle P,T\rangle$ s.t. for all $M^m \in P$ there exists s,τ such that $W;\Gamma \vdash^{\Sigma}_{\Upsilon(m),W(T(m))} M : s,\tau$. Then $P \in \mathcal{DNI}(W,\Sigma,\Upsilon,\Gamma)$.

We can now specify a (W, Σ, Γ)-compatibility predicate that is sufficient for our analysis as including a store S if for every reference $a \in \mathrm{dom}(S)$ its value $S(a)$ satisfies $\Gamma \vdash^{\Sigma}_{\bot,\Omega} S(a) : \bot, \Sigma_2(a)$ and also typability with respect to type systems that enforce NDN [2] and FPC [3].

Precision: The proposed type and effect system is considerably more precise than the combination of the corresponding type and effect systems for NDN and for FPC. While the enforcement of NDN and FPC regard the more refined programming discipline surrounding declassification, distributed noninterference ensures a minimum security goal of not breaking the allowed flow policy of each domain.

VI. Related Work and Conclusions

Migration vs. declassification: When seen as a means for running code under a different allowed flow policy, migration exhibits similarities to the notion of declassification as a flow declaration. However, while a domain's allowed flow policy represents a limit that should not be crossed, a declared flow represents intentions of performing leaks that are rejected by the strictest baseline security lattice. At a more technical level, migration has expression in the semantics of the language, with a potential impact on the observable state, while the flow declaration is used merely with annotation purposes. In the considered language setting, domains cannot be nested into different shades of allowed flow policies. More importantly, code cannot move in and out of a flow declaration.

Noninterference and declassification: Noninterference[10] is often seen as the strictest information flow property of which declassification enabling properties are a weakening. This conservativity principle was proposed [15] as a sanity check for declassification mechanisms. Parameterizing noninterference with respect to a downward closure operator on a basic security lattice [4] offers another perspective on how noninterference relates to declassification. The parameter can be used as an abstract allowed flow policy that determines the particular security lattice which flows should strictly respect. Then, considering the lattice of noninterference properties that results from this parameterization, noninterference is not necessarily the strictest element, but can represent a level of tolerance to declassification operations – a lower bound to finer grained declassification properties. In this sense, parameterized noninterference doesn't necessarily imply non-disclosure.

Information flow in distributed security settings: Domains' security assurances can be represented as security levels. Zdancewic et. al [17] propose in Jif/Split a technique for automatically partitioning programs by placing code and data onto hosts in accordance with DLM labels [12] in the source code. Jif/Split ensures that if a host is subverted, the only data whose confidentiality or integrity is threatened during execution of a part of the program, is data owned by principals that trust that host. Chong et. al [7] present Swift as specialization of this idea for Web applications. Fournet et. al [9] present a compiler that produces distributed code where communications are implemented using cryptographic mechanisms, and ensures that all confidentiality and integrity properties are preserved, despite the presence of active adversaries. In [18], Zheng and Myers address the issue of how availability of hosts might affect information flows in a distributed computation. To our knowledge, the only work on information flow assuming distributed allowed flow policies is on FPC [3]. Its relation to DNI can be understood in combination with the NDN property [2], as is discussed in detail in Section IV.

Conclusion: We have proposed a generalization of the noninterference property to a distributed setting where computation domains are units of allowed flow policies. The property is formalized using a standard bisimulation for information flow [6], [14]. This allows to prove soundness of the proposed enforcement mechanism by means of a known proof method [1]. Bisimulations are a well studied [16] tool for inspecting the behavior of concurrent programs, and are amenable to formal verification [5]. We have supported the adequacy of the formal property by means of examples and its relation with other relevant properties.

Acknowledgments: This work was partially supported by the Portuguese FCT, during our visits to the Indes Team at INRIA and the Language Based Security Group at Chalmers University of Technology. We thank in particular David Sands, Niklas Broberg and Bart van Delft for fruitful discussions.

References

[1] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. *Journal of Computer Security*, 17(5):549–597, 2009.
[2] A. Almeida Matos and J. Cederquist. Non-disclosure for distributed mobile code. *Mathematical Structures in Computer Science*, 21(6), 2011.
[3] A. Almeida Matos and J. Cederquist. Informative types and effects for hybrid migration control. In *Runtime Verification - 4th International Conference. Proceedings*, volume 8174 of *LNCS*. Springer, 2013.
[4] 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*. ACM, 2012.
[5] G. Barthe and L. Prensa Nieto. Secure information flow for a concurrent language with scheduling. *J. Comput. Secur.*, 15(6), 2007.
[6] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1–2), 2002.
[7] S. Chong, J. Liu, A. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. of 21st ACM Symposium on Operating Systems Principles*. ACM, 2007.
[8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
[9] C. Fournet, G. Le Guernic, and T. Rezk. A security-preserving compiler for distributed programs. In *Proc. of the 16th ACM Conf. on Computer and Communications Security*. ACM, 2009.
[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*. IEEE Computer Society, 1982.
[11] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th ACM Symp. on Princ. of Programming Languages*. ACM Press, 1988.
[12] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Soft. Eng. and Methodology*, 9(4), 2000.
[13] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Communications*, 21(1), 2003.
[14] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW'00: 13th IEEE Computer Security Foundations Workshop*, pages 200–215. IEEE Computer Society, 2000.
[15] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.
[16] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.
[17] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3), 2002.
[18] L. Zheng and A. C. Myers. Making distributed computation trustworthy by construction. Technical Report TR2006-2040, Cornell Univ., 2006.