History-based access control for distributed processes

Francisco Martins[1] and Vasco Vasconcelos[2]

[1] Department of Mathematics, University of Azores, Portugal.
[2] Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal.

Abstract. This paper presents a type system to control the migration of code between network nodes in a concurrent distributed framework, using the Dπ language. We express resource access policies as types and enforce policies via a type system. Types describe paths travelled by migrating code, enabling the control of history sensitive access to resources. Sites are logically organised in subnetworks that share the same security policies, statically specified by a network administrator. The type system guarantees that well-typed networks are exempt from security policies violations at runtime.

1 Introduction

The spreading of small, powerful portable machines like PDAs, cellular phones, and laptop computers, equipped with long lasting batteries and wireless communications, is promoting the integration of a broad range of services and encouraging the sharing of resources. Consequently, the protection of personal data and resources from being abusively used is a central concern for the global network participants. This paper proposes a discipline to control the security of resources in a mobile distributed environment.

Take, for example, a typical network architecture for an institution that exposes some of their servers (*e.g.* SSH, HTTP, SMTP, and DNS) to an untrusted network, like the Internet, as described in figure 1 (cf. [23]). The task of the network administrator is to find the correct balance between hiding and revealing the institution's services to the outside world. Some institutions, however, need to give permission to untrusted third parties, for example, to browse their web pages or to download information from their data server, while at the same time need to prevent valuable assets from being defrauded.

One common approach to tackle the problem is to separate the external untrusted network from the institution's network, using a firewall, and to split the inner network into three major areas, offering different levels of security to their components: an *internal network*, protected by an extra firewall, that is not exposed to the outside world at all; a *DMZ*—Demilitarised Zone—that houses servers which are visible to untrusted clients (a semi-protected area); and an *EDMZ*—Extended Demilitarised Zone—hosting internal servers that may be accessed from the DMZ, but not from the external network.

Clustering nodes that share the same security requirements (*e.g.* DMZ, EDMZ, and users subnet) seems a natural method to define security policies for a network. We propose a security model inspired on this notion of clusters, that we name *security groups*, each listing the necessary security requirements. Then we use security groups as building blocks to set up security policies for larger networks, exploiting the policies

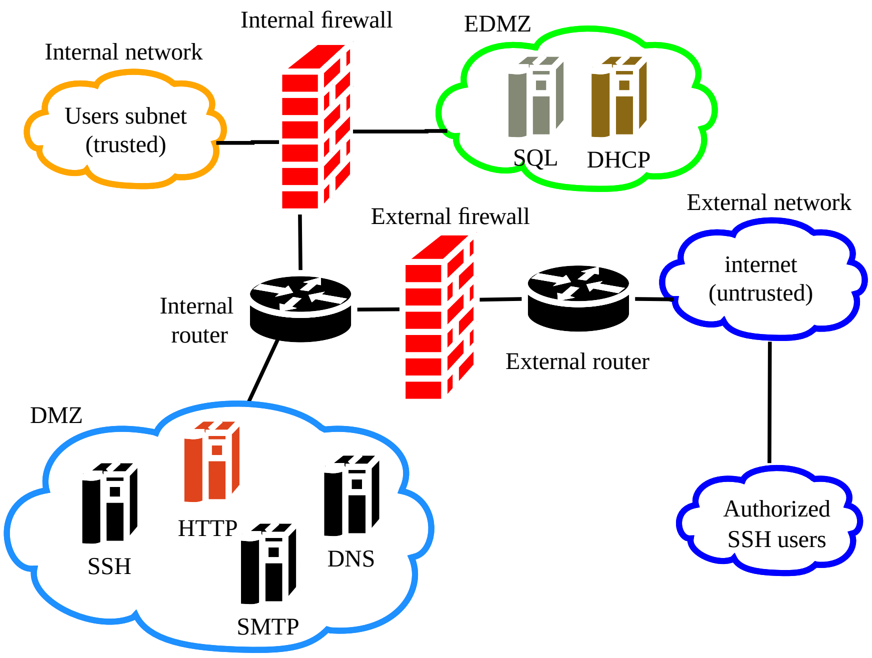


Fig. 1. A two-firewall tiered network architecture.

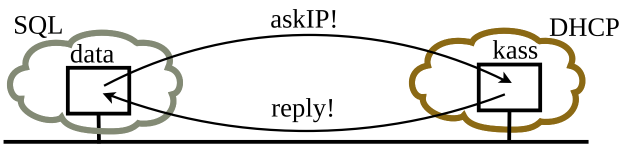
already defined. Each group represents a kind of firewall that dictates the rules and supervises the migration of code that crosses its border. We conceive a model where sites may belong to more than one group and where groups form a hierarchical structure.

We choose Dπ [14, 15] as the underlying calculus and extend it with the notion of *security groups*, an enriched view of the groups Cardelli, Ghelli, and Gordon introduced for the ambient calculus [5–7], thus obtaining a flat computation model (that of Dπ), coupled with a hierarchical organisation from the point of view of security groups, promoting a layered specification of security. Our main motivation is to design a flexible security policy description language, while at the same time, statically guarantee no violation of user-declared security policies.

Sites form a network of computational shells where processes compete for memory, CPU cycles, and other local resources. Communication is local; therefore, the interaction between sites must be programmed explicitly via code migration.

The group's security officer defines a set of rules enumerating what admissible migration paths are allowed to perform what actions. The *migration path*, path for short, is the sequence of groups a piece of code has travelled through, until reaching its current position. We classify actions as *resource usage actions*—the installRes and useRes control attributes describe reading and writing from local channels; *resource allocation actions*—the createRes, createSite, and createGroup attributes enumerate local channel, site, and group creation; *code migration action* is regulated through the tuning of forward control attribute; and finally *management actions*—the inherit attribute enables a group to inherit the policies specified for its parents.

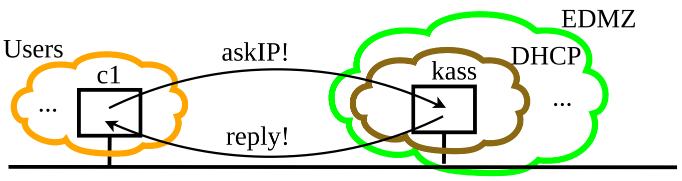
In what follows we explain how to set up security policies using the example in figure 1. As in most real life examples, we take a conservative approach to defining security: all actions are denied unless otherwise stated. Our simple method for writing security rules disables contradictory policies: granting and denying the same privilege.
Groups. The diagram below illustrates an interaction to obtain a valid IP address between a client named *data* from group *SQL* and a server named *kass*, belonging to group *DHCP*. The client runs the process goto $kass.askIP!\langle reply_{@}data\rangle$, and the server replies back running process goto $data.reply!\langle IP\rangle$.



To establish the adequate policies for the network, allowing the code at site *data* and at site *kass* to execute without infringing the security rules, group *DHCP* must allow *SQL*'s code to use local resources (useRes policy) and vice-versa. For each group, we can write down these policies using a simple notation: a pair of sets describing the security policies for the group, and its parent groups.

$$SQL\colon (\{\mathsf{useRes}\colon DHCP\}, \text{-}) \qquad DHCP\colon (\{\mathsf{useRes}\colon SQL\}, \text{-})$$

Subgroups. The notion of subgroups provides for a method to combine group policies. Consider now an IP query from a client *c1* in the *Users* internal network, as depicted in the following diagram.



So, group *EDMZ* must forward code from group *Users* and, furthermore, group *DHCP* must allow group *Users* to use local resources. The types for *EDMZ* and *DHCP* become

$$EDMZ\colon (\{\mathsf{forward}\colon Users\}, \text{-}) \qquad DHCP\colon (\{\mathsf{useRes}\colon Users\}, \{EDMZ\})$$

Notice that group *DHCP* is now a subgroup of *EDMZ* (as specified in the second component of the type for *DHCP*), and that permission to use local resources is only specified at *DHCP*. The point is that each group specifies the policies for the sites that are directly under its control. When a site is under the control of a subgroup, the parent groups only concede the authority for code to cross their boundaries. The remaining policies are "delegated" to the groups where the sites directly belong to, thus avoiding the replication of policies at each group level.

Let us turn our attention to the response from site *kass*, goto $c1.reply!\langle IP\rangle$. Site *kass* is a member of group *DHCP*, which, in turn, is a subgroup of *EDMZ*. So, *kass* may be seen as a member of *DHCP or* as a member of *EDMZ*. Hence, group *Users* may specify security policies addressed specifically at group *DHCP* or at group *EDMZ*.

Suppose that we want to express that group *Users* allows group *EDMZ* to install resources, but that only code from group *DHCP* may use resources. We could set up group *Users* policies as

$$\mathit{Users}\colon (\{\mathsf{useRes}\colon \mathit{DHCP}, \mathsf{installRes}\colon \mathit{EDMZ}\}, \text{-})$$

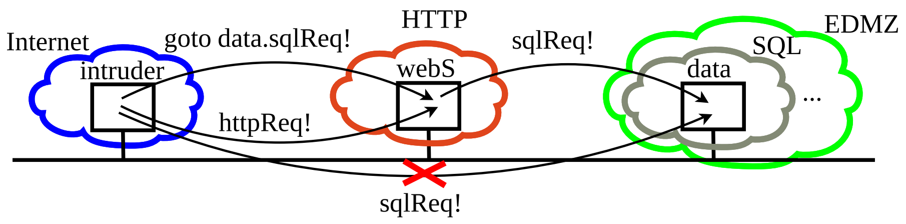
In addition, we may want to be more specific and enable the installation of resources only for group *DHCP* (thus denying code from sites belonging to group *SQL*). So, we could write

$$\mathit{Users}\colon (\{\mathsf{useRes}\colon \mathit{DHCP}, \mathsf{installRes}\colon \mathit{DHCP}\}, \text{-})$$

Policy inheritance. The inheritance of security policies helps in designing and maintaining policies for subgroups. The inheritance is twofold: (a) explicit, via keyword inherit, stating that a subgroup inherits the security policies of its direct parent groups; (b) implicit, adopting the identity of a parent group. Defining security policies for different levels on the grouping hierarchy prevents the enumeration of a myriad of leaf subgroups in the rules.

Migration paths. In addition to indicating a group that may perform some action over the sites of a particular group, we may specify a *path* representing an acceptable sequence of groups that the code must pass through before entering the destination site. The path is specified using a regular expression (figure 5).

Our last example addresses the granting of privileges when the code travels through sites from distinct groups. Consider an intruder browsing web pages that contain data sitting on some data server. We need to give rights to the intruder to view the web pages, but prevent him from gaining access to the data server, either directly from the Internet or via the web server. The network depicted below illustrates this situation



Suppose that the policies for groups *HTTP*, *EDMZ*, and *SQL* are

$$\mathit{HTTP}\colon (\{\mathsf{useRes}\colon \mathit{Internet}\}, \text{-}) \qquad \mathit{EDMZ}\colon (\{\mathsf{forward}\colon \mathit{HTTP}\}, \text{-})$$
$$\mathit{SQL}\colon (\{\mathsf{useRes}\colon \mathit{HTTP}\}, \{\mathit{EDMZ}\})$$

Processes goto $webS.httpReq!\langle\diamond\rangle$ and goto $data.sqlReq!\langle\diamond\rangle$ do not violate the security rules, whereas goto $data.sqlReq!\langle\diamond\rangle$ launched by the intruder breaks the security rules at groups *EDMZ* (forwards code just from *HTTP* group) and *SQL* (only allows the use of resources from *HTTP*). What about process goto $webS$.goto $data.sqlReg!\langle\diamond\rangle$? A trusting relation must not be transitive: although group *HTTP* allows code from group *Internet* to use its resources, and group *SQL* allows *HTTP* the same privileges, that does not imply that *SQL* should allow code migrating from *Internet* to use its resources, either directly or through a site in *HTTP*.

$v ::=$	*Values*	$n ::=$	*Names*
$a_{@s}$	located channel	a, b, c, x	channels
$\mid \diamond$	basic value	$\mid r, s, t, y$	sites
		$\mid f, g, h$	groups
$P, Q ::=$	*Processes*	$N, M ::=$	*Networks*
stop	termination	stop	termination
$\mid (\boldsymbol{\nu} n\colon L_{@s})\ P$	restriction	$\mid (\boldsymbol{\nu} n\colon L)\ N$	restriction
$\mid P \mid Q$	composition	$\mid N \mid M$	composition
$\mid \mathsf{goto}\ s.P$	migration	$\mid s\llbracket P \rrbracket$	site
$\mid a!\langle v \rangle$	output		
$\mid a?(v)\ P$	input	see figure 11 for the syntax of types L	
$\mid a?{*}(v)\ P$	replication		

Fig. 2. Syntax of Dπ.

The security policies set above for *EDMZ* and for *SQL* do not allow migration of code from *intruder* to *webS* and then to *data*, because the path the code travels is *HTTP Internet*, which is not allowed. However, it could be interesting to model a situation in which this migration path is acceptable. Consider the subgroup *SSHusers* in *Internet* accessing to the *data* server. The network administrator may permit that the sites in group *SSH* may be used as proxies for sites in group *SSHusers*, and allow code from and honest agent to migrate through *SSH* to use *SQL*'s resources. The types for *SSH*, *EDMZ*, and *SQL* groups would then become

$$\mathit{SSH}\colon (\{\mathsf{useRes}\colon \mathit{SSHusers}\}, \text{-}) \quad \mathit{EDMZ}\colon (\{\mathsf{forward}\colon \mathit{SSH}\ \mathit{SSHusers}\}, \text{-})$$
$$\mathit{SQL}\colon (\{\mathsf{useRes}\colon \mathit{SSH}\ \mathit{SSHusers}\}, \{\mathit{EDMZ}\})$$

Would the network administrator need to specify that all the code arriving through group *SSH* is welcome to use *SQL*'s resources, it might set *EDMZ* and *SQL* policies as

$$\mathit{EDMZ}\colon (\{\mathsf{forward}\colon \mathit{SSH}\cdot\star\}, \text{-}) \qquad \mathit{SQL}\colon (\{\mathsf{useRes}\colon \mathit{SSH}\cdot\star\}, \{\mathit{EDMZ}\})$$

Other simple path patterns can be specified, like, for instance, code originated at group *SSH* as, $\cdot\star$ *SSH*, or code that passes through group *SSH* as, $\cdot\star$ *SSH* $\cdot\star$.

Outline. The next section briefly introduces the Dπ syntax and its operational semantics. Section 3 introduces our approach to the checking of security policies and present the notion of runtime errors (via a tagged version of Dπ). Section 4 is devoted to the type assignment system and states the results we achieved. The last section presents the related work and states our conclusions.

2 Dπ syntax and operational semantics

This section deals with the syntax and the operational semantics of Dπ, mainly taken from Hennessy and Riely [15].

Syntax. The syntax of the calculus is defined in figure 2. The main difference w.r.t. the original Dπ is the usage of groups, namely the new constructor to create groups. We

1. $((N \mid M) \mid M') \equiv (N \mid (M \mid M'))$ $\quad (M \mid N) \equiv (N \mid M)$ $\quad (N \mid \mathsf{stop}) \equiv N$
2. $(\boldsymbol{\nu} n\colon T)\ N \mid M \equiv (\boldsymbol{\nu} n\colon T)\ (N \mid M)$ $\quad$ if $n \notin \mathrm{fn}(M)$
   $(\boldsymbol{\nu} n\colon T)\ (\boldsymbol{\nu} m\colon T')\ N \equiv (\boldsymbol{\nu} m\colon T')\ (\boldsymbol{\nu} n\colon T)\ N$ $\quad$ if m not in T, and n not in T'
   $(\boldsymbol{\nu} n\colon L@s)\ s[\![P]\!] \equiv s[\![(\boldsymbol{\nu} n\colon L)\ P]\!]$ $\quad$ if $n \neq s$
3. $s[\![P]\!] \mid s[\![Q]\!] \equiv s[\![P \mid Q]\!]$
4. $(\boldsymbol{\nu} n\colon T)\ \mathsf{stop} \equiv \mathsf{stop}$ $\quad (\boldsymbol{\nu} s\colon T)\ s[\![\mathsf{stop}]\!] \equiv \mathsf{stop}$

Fig. 3. Structural congruence.

$$s[\![a!\langle b@r\rangle]\!] \mid s[\![a?(x@y)\ P]\!] \to s[\![P\{r/y\}\{b/x\}]\!] \qquad (\text{COMC}_1)$$

$$s[\![a!\langle\diamond\rangle]\!] \mid s[\![a?(\diamond)\ P]\!] \to s[\![P]\!] \qquad (\text{COMC}_2)$$

$$s[\![a!\langle b@r\rangle]\!] \mid s[\![a?{*}(x@y)\ P]\!] \to s[\![P\{r/y\}\{b/x\}]\!] \mid s[\![a?{*}(x@y)\ P]\!] \qquad (\text{COMR}_1)$$

$$s[\![a!\langle\diamond\rangle]\!] \mid s[\![a?{*}(\diamond)\ P]\!] \to s[\![P]\!] \mid s[\![a?{*}(\diamond)\ P]\!] \qquad (\text{COMR}_2)$$

$$s[\![\mathsf{goto}\ r.P]\!] \to r[\![P]\!] \qquad \frac{N \to M}{(\boldsymbol{\nu} n\colon L@s)\ N \to (\boldsymbol{\nu} n\colon L@s)\ M} \qquad (\text{MIG, RES})$$

$$\frac{N \to N'}{N \mid M \to N' \mid M} \qquad \frac{N \equiv N' \quad N' \to M' \quad M' \equiv M}{N \to M} \qquad (\text{PAR, STR})$$

Fig. 4. Reduction rules.

consider a monadic version of the calculus where only located names can be passed around, since our main focus is the control of migration, not that of communication.

We briefly address the Dπ syntax; the interested reader should refer to [14, 15] for motivations and details. The calculus presents two main syntactic categories: processes and networks. At process level we find the usual asynchronous π-calculus constructs [2, 16]; processes are built from the inactive process, stop, and from the asynchronous output process, $a!\langle v\rangle$, using three constructs: name restriction, $(\boldsymbol{\nu} n\colon T)\ P$, parallel composition, $P \mid Q$, and input, $a?(v)\ P$. We also include a form of replicated input, $a?{*}(v)\ P$. Moreover, Dπ contains an operator that sends a process P to a specific location s: the goto $s.P$ process.

Networks are assembled from the inaction network, stop, and from processes running at specific named locations called sites, $s[\![P]\!]$, using name restriction, $(\boldsymbol{\nu} n\colon T)\ N$, and parallel composition, $N \mid M$.

Operational semantics. The binders of the calculus are the usual in π-calculi like languages: name n is bound in $(\boldsymbol{\nu} n\colon T)\ P$ and in $(\boldsymbol{\nu} n\colon T)\ N$, whereas x and y are both bound in $a?(x@y)\ P$. Networks are taken up to α-congruence in such a way that bound names are different from free names and from each other.

Operational semantics is defined on top of a *structural congruence relation*, $\equiv$, that is the least congruence relation closed under the rules defined in figure 3. It follows closely the structural congruence relation introduced for Dπ.

Reduction in figure 4 is mainly taken from Dπ, except for obvious adjustments to incorporate groups.

3 Security policy

Writing security policies. A security policy ($\mathcal{P}$) consists of a set of rules (π). Effect rules ($\tau\colon S$) describe the set of admissible paths in the group hierarchy that code must

$\mathcal{P} ::=$	*Policies*	$\tau ::=$	*Effects*	$S ::=$	*Paths*
$\{\pi_1 \ldots \pi_n\}$		useRes	output	ε	empty path
		| installRes	input	| g	group
$\pi ::=$	*Security rules*	| createRes	ch. creation	| $\cdot$	any group
$\tau\colon S$	effect rules	| createSite	site creation	| SS	concatenation
| forward: S	code forward	| createGroup	group creation	| $S + S$	alternation
| inherit	inherit policies			| $S\star$	kleene star

Fig. 5. Syntax for security policies.

visit before being able to perform the action the policy protects. Rule forward governs the migration of code. For code migration to succeed, there must be a path all along the group hierarchy that authorises the forwarding of the code to the destination site. The inherit allows a group to import the rules defined for its direct parents.

The *effects*, τ, correspond directly to the actions of the calculus: the input and output actions are related with the installRes and useRes effects, respectively; channel, site, and group creation are associated with createRes, createSite, and createGroup effects, respectively.

A *path pattern*, S, is a regular expression. A group g stands for itself, the symbol $\cdot$ is a wild card that represents any group. Concatenation, alternation, and Kleene closure possess the usual meaning.

Checking security policies. A *typing* Γ is a partial function of finite domain from names to types. For the current section we outline that the type for sites ($s\colon G$) is the set of groups that the site belongs to, and the type for groups ($g\colon (\mathcal{P}, G)$) is a pair: security-rules, parent-groups. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ. When $x \notin \mathrm{dom}(\Gamma)$, we write $\Gamma, x : T$ for the type environment Γ' such that $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = T$ and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$. We use $\widetilde{X}$ to denote a possibly empty sequence of X's. Similarly, $\widetilde{r\colon F}$ and $\widetilde{g \in G}$ denote $r_1\colon F_1, \ldots, r_n\colon F_n$ and $g_1 \in G_1, \ldots g_n \in G_n$, respectively.

Functions allows and canEnter, defined in figures 6 and 7, perform security checking. Before outlining function allows, we give an overview of function matches defined in [19]. A formula $\widetilde{g}$ matches S means that a path $\widetilde{g}$ fits in path pattern S. The rules for most path constructs are straightforward; we present the non-standard ones: a group matches itself or any group in its hierarchy.

$$\Gamma \vdash g \;\mathsf{matches}\; g \qquad\qquad \frac{\Gamma, g\colon (\mathcal{P}, G) \vdash h \;\mathsf{matches}\; f \qquad h \in G}{\Gamma, g\colon (\mathcal{P}, G) \vdash g \;\mathsf{matches}\; f}$$

A formula g allows $\widetilde{f} : \pi$ (figure 6) says that group g allows code that travelled through path $\widetilde{f}$ to perform action π; path $\widetilde{f}$ is matched against the path pattern associated with policy π using function matches. We use A in the rules to denote a set of effects τ. The createGroup effect receives special treatment, since the creation of a group establishes a new node in the group hierarchy, and the groups above must accept its new member. Since a group may identify itself as any of its parents, the creation of a subgroup must collect the acceptance of the whole hierarchy. Forwarding code requires that at least one branch in the hierarchy grants the forward policy to the path the code

$$\frac{\begin{array}{c}\mathit{\Gamma}(g) = (\mathcal{P} \cup \{\tau\colon S\}, G)\\ \mathit{\Gamma} \vdash \widetilde{f} \ \mathsf{matches}\ S\\ \tau \neq \mathsf{createGroup}\end{array}}{\mathit{\Gamma} \vdash g\ \mathsf{allows}\ \widetilde{f}\colon \tau}
\qquad
\frac{\begin{array}{c}\mathit{\Gamma}(g) = (\mathcal{P} \cup \{\mathsf{createGroup}\colon S\}, G)\\ \mathit{\Gamma} \vdash \widetilde{f}\ \mathsf{matches}\ S\\ \forall h \in G \quad \mathit{\Gamma} \vdash h\ \mathsf{allows}\ \widetilde{f}\colon \{\mathsf{createGroup}\}\end{array}}{\mathit{\Gamma} \vdash g\ \mathsf{allows}\ \widetilde{f}\colon \{\mathsf{createGroup}\}}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}(g) = (\mathcal{P} \cup \{\mathsf{forward}\colon S\}, \emptyset)\\ \mathit{\Gamma} \vdash \widetilde{f}\ \mathsf{matches}\ S\end{array}}{\mathit{\Gamma} \vdash g\ \mathsf{allows}\ \widetilde{f}\colon \{\mathsf{forward}\}}
\qquad
\frac{\begin{array}{c}\mathit{\Gamma}(g) = (\mathcal{P} \cup \{\mathsf{forward}\colon S\}, \{h\} \cup G)\\ \mathit{\Gamma} \vdash \widetilde{f}\ \mathsf{matches}\ S \quad \mathit{\Gamma} \vdash h\ \mathsf{allows}\ \widetilde{f}\colon \{\mathsf{forward}\}\end{array}}{\mathit{\Gamma} \vdash g\ \mathsf{allows}\ \widetilde{f}\colon \{\mathsf{forward}\}}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}(g) = (\{\mathsf{inherit}\} \cup \mathcal{P}, \{h\} \cup G)\\ \mathit{\Gamma} \vdash h\ \mathsf{allows}\ \widetilde{f}\colon A\end{array}}{\mathit{\Gamma} \vdash g\ \mathsf{allows}\ \widetilde{f}\colon A}
\qquad
\frac{\forall g \in G, \forall \widetilde{f \in F}, \forall \pi \in A,\ \mathit{\Gamma} \vdash g\ \mathsf{allows}\ \widetilde{f}\colon \pi}{\mathit{\Gamma} \vdash G\ \mathsf{allows}\ \widetilde{F}\colon A}$$

$$\frac{\mathit{\Gamma}, s\colon G, \widetilde{r\colon F} \vdash G\ \mathsf{allows}\ \widetilde{F}\colon A}{\mathit{\Gamma}, s\colon G, \widetilde{r\colon F} \vdash s\ \mathsf{allows}\ \widetilde{r}\colon A}
\qquad
\frac{}{\mathit{\Gamma} \vdash s\ \mathsf{allows}\ s\colon A}$$

Fig. 6. allows relation.

$$\frac{\mathit{\Gamma}(f) = (\mathcal{P}, \emptyset)}{\mathit{\Gamma} \vdash \widetilde{g}\ \mathsf{canEnter}\ f}
\qquad
\frac{\begin{array}{c}\mathit{\Gamma}(f) = (\mathcal{P}, \{h\} \cup G)\\ \mathit{\Gamma} \vdash h\ \mathsf{allows}\ \widetilde{g}\colon\ \mathsf{forward}\end{array}}{\mathit{\Gamma} \vdash \widetilde{g}\ \mathsf{canEnter}\ f}
\qquad
\frac{}{\mathit{\Gamma} \vdash s\ \mathsf{canEnter}\ s}$$

$$\frac{\forall \widetilde{g \in G}, \forall f \in F, \mathit{\Gamma} \vdash \widetilde{g}\ \mathsf{canEnter}\ f}{\mathit{\Gamma} \vdash \widetilde{G}\ \mathsf{canEnter}\ F}
\qquad
\frac{\mathit{\Gamma}, \widetilde{s\colon G}, r\colon F \vdash \widetilde{G}\ \mathsf{canEnter}\ H}{\mathit{\Gamma}, \widetilde{s\colon G}, r\colon F \vdash \widetilde{s}\ \mathsf{canEnter}\ r}$$

Fig. 7. canEnter relation.

travelled. When the inherit policy keyword is set for a group, the security policies for the direct parent groups are considered as a part of the security specification for the group.

Function canEnter (figure 7) checks whether code that travels through a given path has permission to enter a target group. A formula $\widetilde{g}$ canEnter f means that group f accepts code that has travelled through path $\widetilde{g}$. This privilege is controlled using the forward policy. Code that went all along path $\widetilde{g}$ is able to enter the frontier of group f, if there exists a path through f's hierarchy granting, at each group in the path (except for the target group), the forward right to $\widetilde{g}$.

Tagged language. To precise the notion of runtime errors we need to make explicit in the syntax the path the code travelled and the group policies (cf. [15]). Therefore, we present a tagged version for the language introduced in figure 2 and the corresponding tagged semantics.

Tagged syntax. Figure 8 (syntax) summarises the syntactic changes: (a) we append to the site constructor the *sequence of sites* the code has visited and the set of assumptions needed to check security policies for the processes it runs; (b) we add to name restriction the sequence of sites followed by the code before creating the name, since, by scope extrusion, a name may appear at network level and we need this information to formalise runtime errors. All the remaining syntax is left unchanged.

Syntax (all rules from figure 2, replacing site *and* restriction *by the following rules)*

$$s[P]^{\tilde{r}}_{\Gamma} \qquad\qquad (\boldsymbol{\nu}_{\tilde{t}}\, n\colon T)\ N$$

Structural congruence (group 1. from figure 3, plus the following rules)

2. $(\boldsymbol{\nu}_{\tilde{t}}\, n\colon L@s)\ s[P]^{\tilde{t}}_{\Gamma \sqcap n\colon L@s} \equiv_T s[(\boldsymbol{\nu} n\colon L)\ P]^{\tilde{t}}_{\Gamma}$ if $n \notin \mathrm{dom}(\Gamma) \cup \{s\}$
3. $s[P]^{\tilde{t}}_{\Gamma} \mid s[Q]^{\tilde{t}}_{\Gamma} \equiv_T s[P \mid Q]^{\tilde{t}}_{\Gamma}$

Fig. 8. The tagged language—syntax and structural congruence.

$$s[a!\langle b@r\rangle]^{\tilde{t}}_{\Gamma, r\colon G} \mid s[a?(x@y)\ P]^{\tilde{u}}_{\Delta} \mapsto s[P\{r/y\}\{b/x\}]^{\tilde{u}}_{\Delta \sqcap r\colon G} \qquad \text{(T-COMC}_1\text{)}$$

$$s[a!\langle\diamond\rangle]^{\tilde{t}}_{\Gamma} \mid s[a?(\diamond)\ P]^{\tilde{u}}_{\Delta} \mapsto s[P]^{\tilde{u}}_{\Delta} \qquad \text{(T-COMC}_2\text{)}$$

$$s[a!\langle b@r\rangle]^{\tilde{t}}_{\Gamma, r\colon G} \mid s[a?{*}(x@y)\ P]^{\tilde{u}}_{\Delta} \mapsto s[P\{r/y\}\{b/x\}]^{\tilde{u}}_{\Delta \sqcap r\colon G} \mid s[a?{*}(x@y)\ P]^{\tilde{u}}_{\Delta} \qquad \text{(T-COMR}_1\text{)}$$

$$s[a!\langle\diamond\rangle]^{\tilde{t}}_{\Gamma} \mid s[a?{*}(\diamond)\ P]^{\tilde{u}}_{\Delta} \mapsto s[P]^{\tilde{u}}_{\Delta} \mid s[a?{*}(\diamond)\ P]^{\tilde{u}}_{\Delta} \qquad \text{(T-COMR}_2\text{)}$$

$$s[\mathsf{goto}\ r.P]^{\tilde{t}}_{\Gamma} \mapsto r[P]^{s\tilde{t}}_{\Gamma} \qquad \frac{N \mapsto M}{(\boldsymbol{\nu}_{\tilde{t}}\, n\colon T)\ N \mapsto (\boldsymbol{\nu}_{\tilde{t}}\, n\colon T)\ M} \qquad \text{(T-MIG, T-RES)}$$

(*plus rules* PAR, *and* STR *from figure 4*)

Fig. 9. The tagged language—reduction.

Tagged structural congruence. Name extrusion (rule 2, figure 8) records the code journey leading to name creation, whereas restricting it to a site is only possible when the code running at the site followed the same migration path as that of the name creation. Notice that the set of assumptions at the left-hand side of the congruence relation enlarges with the name declared at network level, announcing the creation of the name. The meet operator, $\sqcap$, (cf. [14, 15]), used to combine the new name with that in type assumptions is defined in [19]. Merging sites (rule 3, figure 8) is only viable when the tagged information agree: it is not possible to merge sites that execute code travelling through different locations or that are governed by distinct security policies. The remaining structural congruence rules reflect the syntactic adjustments and are omitted.

Tagged reduction (figure 9). The main differences w.r.t. untagged reduction (figure 4) concern communication and code migration. Communication may occur under dissimilar views of the security policies of a site, in particular, the input process may not use or even have knowledge of the value it is going to receive (except for its type). Therefore, communication updates the typing assumptions of the receiving process with information from the emitting process. Rule T-COMC$_1$ updates the resulting process with type information for the communicated site r. When site r is not mentioned in process P, the type information is just appended to typing Δ, otherwise we compute the least common supertype of the type figuring in Δ and the one communicated. The subject reduction theorem (page 14) guarantees that for well-typed processes the meet operation is always defined. The remaining rules are either similar to the above (T-COMR$_1$) or close to their untagged version (T-COMC$_2$ and T-COMR$_2$).

As for code migration, rule T-MIG, we append the name of the source site to the migration path. This information is fundamental to reason about security. We check the

R-OUT	$s[a!\langle v\rangle]^{\widetilde{r}}_{\Gamma} \xmapsto{err}$	if $\Gamma \nvdash s$ allows $\widetilde{r}$: useRes
R-INP	$s[a?(v)\ P]^{\widetilde{r}}_{\Gamma} \xmapsto{err}$	if $\Gamma \nvdash s$ allows $\widetilde{r}$: installRes
R-MIG	$s[\mathsf{goto}\ t.P]^{\widetilde{r}}_{\Gamma} \xmapsto{err}$	if $\Gamma \nvdash s\widetilde{r}$ canEnter t
R-RES$_1$	$s[(\boldsymbol{\nu} a\colon L)\ P]^{\widetilde{r}}_{\Gamma} \xmapsto{err}$	if $\Gamma \nvdash s$ allows $\widetilde{r}$: createRes
R-RES$_2$	$(\boldsymbol{\nu}_{\widetilde{r}}\, a\colon L@s)\ N \xmapsto{err}$	if $\Gamma \nvdash s$ allows $\widetilde{r}$: createRes

Fig. 10. Runtime errors.

security policies considering the sites that migration code visits because this information is important to express the trust between the destination group and the rest of the network. Rule T-RES results from the syntax update.

Runtime errors. The unary relation, $\xmapsto{err}$, defined in figure 10, identifies processes that break some security policy during reduction.

The output (input) process fails, R-OUT (R-INP), if the site that sent the code, r, has no permission to use (install) resources. We omit the rule for replicated input, since it is similar to rule R-INP. For code migration, rule R-MIG states that a goto process incurs in a runtime error if it cannot enter the border of the groups where the target site resides. Notice the role of typing Γ—a placeholder for security policies—, and the need to talk about the site where the code is, s, the sequence of sites visited by the code, $\widetilde{r}$, and the site where the code is migrating to, t. Rule R-RES$_1$ says that the channel creation operation fails if the current site does not allow the site that sent the code to create channels. Rule R-RES$_2$ is similar.

4 Typing system

In this section we present two type and effect systems (for the tagged and for the untagged languages) that check whether networks respect the security policies defined for groups. The type systems are based on a subtyping relation *à la* Sangiorgi and Pierce [21], and are parametric in two functions that are used to check the security policies, namely, the allows and the canEnter functions.

Types. The syntax for types is depicted in figure 11. We assign types to channels, to sites, and to groups. *Types*, T, may be local or global: *local types*, L, are used when creating names at a given site; *global* (or *located types*), $L@s$, are assigned to names when declared at network level.

Name types, L, aggregate *channel types*, C, that trace the type of the values that are communicated along the channel, as well as its usage (input, output, or both); *site types*, G, that simply record the set of groups the site belongs to; and *group types*, $(\mathcal{P}, G)$, which is a central notion in our work: it is at group level that we record information for security, namely, (a) the set of *security rules*, $\mathcal{P}$, that govern the interaction with the network, and (b) the *set*, G, *of the parent groups* of the group.

Channels can carry other channels, as well as basic values, as described by *value types*, V. The *type for channel values* assumes the form $C@G$, where C is the type of the channels that can be carried, and G is the set of groups hosting the communicated channels. The subtype relation characterising *channel tags* I is introduced in figure 12.

$T ::=$	*Types*	$C ::=$	*Local channel types*	$V ::=$	*Value types*
L	local type	$\langle V \rangle^I$	local channel	$C@G$	channel
$\mid L@s$	global type			$\mid \mathsf{unit}$	basic type
$L ::=$	*Name types*	$I ::=$	*Tags*	G	*set of groups*
C	local channel	r	input		
$\mid G$	site type	$\mid \mathsf{w}$	output		
$\mid (\mathcal{P}, G)$	group type	$\mid \mathsf{rw}$	input/output		

Fig. 11. Syntax of types.

$$\mathsf{unit} <: \mathsf{unit} \qquad \frac{C_1 <: C_2 \quad G_1 \subseteq G_2}{C_1@G_1 <: C_2@G_2} \qquad (\textit{Value subtyping})$$

$$\frac{i = \mathsf{r}, \mathsf{rw} \quad V_1 <: V_2}{\langle V_1 \rangle^i <: \langle V_2 \rangle^{\mathsf{r}}} \qquad \frac{i = \mathsf{w}, \mathsf{rw} \quad V_2 <: V_1}{\langle V_1 \rangle^i <: \langle V_2 \rangle^{\mathsf{w}}} \qquad \frac{V_1 <: V_2 \quad V_2 <: V_1}{\langle V_1 \rangle^{\mathsf{rw}} <: \langle V_2 \rangle^{\mathsf{rw}}}$$

(*Local channel subtyping*)

$$\frac{C_1 <: C_2}{C_1@s <: C_2@s} \qquad (\textit{Global channel subtyping})$$

Fig. 12. Subtyping relation.

Subtyping. The *subtyping relation*, $<:$, is defined as the least preorder relation on types that satisfies the rules in figure 12 where channels are tagged according to their usage: input (r), output (w), and input/output (rw). We extend the subtyping relation to deal with types involving groups. The original intuitions remain unchanged, namely that the subtyping relation is covariant for inputs, contravariant for outputs, and invariant if the channels are used both for input and for output purposes. The subtyping rules are straightforward. Notice the set inclusion to handle groups in value subtyping and the last subtyping rule that relates located channels.

Typing the untagged language. The type system collects the effects of the actions performed by processes. For instance, process $a!\langle b@r \rangle$ running at site s has effect useRes. Then, following to a goto action we check whether the path travelled by the code has the right privileges to perform the intended action, in the present case an *output*. At network level there is nothing to be checked, since there is no computation taking place. Also, we do not check code running at its host site—code that is not in the continuation part of a goto process—, since we assume that there is no need to grant specific privileges in such circumstances.

The type system, described in figures 13–15, includes three kinds of judgements: (a) judgement $\Gamma \vdash \mathrm{env}$ asserts that Γ is a well-formed environment; (b) judgement $\Gamma \vdash_{s\tilde{t}} P : A$ means that process P is running at site s has travelled through the sequence of sites $\tilde{t}$, has the effects enumerated in set of actions A, and is well typed under typing assumptions Γ; and (c) judgement $\Gamma \vdash N$ denotes that network N is well typed under typing assumptions Γ.

Well-formed environment rules, figure 13, guarantee that group structures are not circular. Rule E-GROUP ensures that when we enlarge a typing with a new group definition, its parent groups are already in the typing. The remaining rules are simple.

$$\textsc{E-Unit}\quad \diamond\colon \mathsf{unit} \vdash \mathrm{env} \qquad \textsc{E-Channel}\ \frac{\varGamma \vdash \mathrm{env}}{\varGamma, a\colon C@s \vdash \mathrm{env}}$$

$$\textsc{E-Site}\ \frac{\varGamma \vdash \mathrm{env}}{\varGamma, s\colon G \vdash \mathrm{env}} \qquad \textsc{E-Group}\ \frac{\varGamma \vdash \mathrm{env} \qquad G \subseteq \mathrm{dom}(\varGamma)}{\varGamma, g\colon (\mathcal{P}, G) \vdash \mathrm{env}}$$

Fig. 13. Well-formed environments.

$$\textsc{P-OutB}\ \frac{\varGamma \vdash \mathrm{env} \qquad \varGamma(a) <: \langle \mathsf{unit}\rangle^{\mathsf{w}}@s}{\varGamma \vdash_{s\tilde{t}} a!\langle\diamond\rangle\colon \{\mathsf{useRes}\}} \qquad \textsc{P-OutC}\ \frac{\begin{array}{c}\varGamma \vdash \mathrm{env} \qquad \varGamma(a) <: \langle C@G\rangle^{\mathsf{w}}@s \\ \varGamma(r) = G \qquad\quad \varGamma(b) = C@r\end{array}}{\varGamma \vdash_{s\tilde{t}} a!\langle b@r\rangle\colon \{\mathsf{useRes}\}}$$

$$\textsc{P-InpB}\ \frac{\varGamma \vdash_{s\tilde{t}} P\colon A \qquad \varGamma(a) <: \langle \mathsf{unit}\rangle^{\mathsf{r}}@s}{\varGamma \vdash_{s\tilde{t}} a?(\diamond)\ P\colon A \cup \{\mathsf{installRes}\}}$$

$$\textsc{P-InpC}\ \frac{\varGamma, x\colon C@y, y\colon G \vdash_{s\tilde{t}} P\colon A \qquad \varGamma(a) <: \langle C@G\rangle^{\mathsf{r}}@s \qquad y \text{ not in } \varGamma}{\varGamma \vdash_{s\tilde{t}} a?(x@y)\ P\colon A \cup \{\mathsf{installRes}\}}$$

$$\textsc{P-InpR}\ \frac{\varGamma \vdash_{\tilde{t}} a?(v)\ P\colon A}{\varGamma \vdash_{\tilde{t}} a?{*}(v)\ P\colon A} \qquad \textsc{P-Par}\ \frac{\varGamma \vdash_{\tilde{t}} P\colon A_1 \qquad \varGamma \vdash_{\tilde{t}} Q\colon A_2}{\varGamma \vdash_{\tilde{t}} P \mid Q\colon A_1 \cup A_2}$$

$$\textsc{P-ResS}\ \frac{\varGamma, r\colon G@s \vdash_{s\tilde{t}} P\colon A \qquad r \text{ not in } \varGamma}{\varGamma \vdash_{s\tilde{t}} (\boldsymbol{\nu} r\colon G)\ P\colon A \cup \{\mathsf{createSite}\}}$$

$$\textsc{P-ResC}\ \frac{\varGamma, a\colon C@s \vdash_{s\tilde{t}} P\colon A}{\varGamma \vdash_{s\tilde{t}} (\boldsymbol{\nu} a\colon C)\ P\colon A \cup \{\mathsf{createRes}\}}$$

$$\textsc{P-ResG}\ \frac{\varGamma, g\colon (\mathcal{P}, G)@s \vdash_{s\tilde{t}} P\colon A \qquad g \text{ not in } \varGamma}{\varGamma \vdash_{s\tilde{t}} (\boldsymbol{\nu} g\colon (\mathcal{P}, G))\ P\colon A \cup \{\mathsf{createGroup}\}} \qquad \textsc{P-Nil}\ \frac{\varGamma \vdash \mathrm{env}}{\varGamma \vdash_{\tilde{t}} \mathsf{stop}\colon \emptyset}$$

$$\textsc{P-Mig}\ \frac{\varGamma \vdash_{r\tilde{t}} P\colon A \qquad \varGamma \vdash r \mathrel{\mathsf{allows}} \tilde{t}\colon A \qquad \varGamma \vdash \tilde{t} \mathrel{\mathsf{canEnter}} r}{\varGamma \vdash_{\tilde{t}} \mathsf{goto}\ r.P\colon \emptyset}$$

Fig. 14. Typing processes.

As for processes (figure 14), rule P-OutB enforces that typing $\varGamma$ is well formed, and that channel a is a write or a read-write channel located at the site where the process is running and is capable of carrying unit values. Rule P-OutC types an output process that carries another channel, rather than the `unit` value. The difference w.r.t P-OutB regards the type for channel a: it must carry channels of the type of b and must be located at the groups of site r. To type an input process, $a?(x@y)\ P$, channel a must be a read or a read-write channel. The continuation process, P, must be well typed in a typing augmented with x and y. Notice that channel x is located at y and that y is defined as a site member of the groups that channel a can carry. Hence, we guarantee that the privileges for the actions involving x and y are correctly checked, since we verify policies against all groups in G. The subtyping rule is covariant for inputs, which means that, if the type of channel a is a subtype of $\langle C@G\rangle^{\mathsf{r}}@s$, then a carries channels located at a subset of G. Finally, the effect of the input action—$\mathsf{installRes}$—is appended to the set of actions. Rules P-InpB and P-InpR follow a pattern similar to the one just described.

The parallel composition of processes, $P \mid Q$, combines the set of actions gathered when typing the individual processes. We split name restriction over three rules, P-ResS, P-ResC, and P-ResG, since there is a specific effect associated to each creation action.

It is at code migration, goto $r.P$, that all the security checking takes place. When we reach a goto process we have all the information necessary to check security polices,

$$\textsc{N-Site}\ \frac{\Gamma \vdash_s P \colon A}{\Gamma \vdash s[P]} \qquad \textsc{N-Res}\ \frac{\Gamma, n \colon L@s \vdash N \qquad n \text{ not in } \Gamma}{\Gamma \vdash (\boldsymbol{\nu} n \colon (L, @)s)\ N}$$

$$\textsc{N-Par}\ \frac{\Gamma \vdash N \qquad \Gamma \vdash M}{\Gamma \vdash N \mid M} \qquad \textsc{N-Nil}\ \frac{\Gamma \vdash \text{env}}{\Gamma \vdash \mathsf{stop}}$$

Fig. 15. Typing networks.

$$\textsc{T-Site}\ \frac{\begin{array}{cc}\Delta \vdash_{s\widetilde{t}} P \colon A & \Delta \vdash s\ \mathsf{allows}\ \widetilde{t} \colon A \\ \Gamma <\colon \Delta & \Delta \vdash \widetilde{t}\ \mathsf{canEnter}\ s\end{array}}{\Gamma \Vdash s[P]^{\widetilde{t}}_{\Delta}} \qquad \textsc{T-Resc}\ \frac{\begin{array}{c}\Gamma, a \colon C@s \Vdash N \\ \Gamma \vdash s\ \mathsf{allows}\ \widetilde{t} \colon\ \mathsf{createRes}\end{array}}{\Gamma \Vdash (\boldsymbol{\nu}_{\widetilde{t}}\, a \colon C@s)\ N}$$

$$\textsc{T-Ress}\ \frac{\begin{array}{c}\Gamma, s \colon G@s \Vdash N \\ \Gamma \vdash s\ \mathsf{allows}\ \widetilde{t} \colon\ \mathsf{createSite}\end{array}}{\Gamma \Vdash (\boldsymbol{\nu}_{\widetilde{t}}\, s \colon G)\ N} \qquad \textsc{T-Resg}\ \frac{\begin{array}{c}\Gamma, g \colon (\mathcal{P}, G)@s \Vdash N \qquad g \text{ not in } \Gamma \\ \Gamma \vdash s\ \mathsf{allows}\ \widetilde{t} \colon\ \mathsf{createGroup}\end{array}}{\Gamma \Vdash (\boldsymbol{\nu}_{\widetilde{t}}\, g \colon (\mathcal{P}, G))\ N}$$

(*plus all rules in figures 14, 15, except* N-SITE, *and* N-RES)

Fig. 16. The tagged language—typing networks.

namely, we know the sequence of sites visited by code (annotated under the turnstile), the target site (indicated in the syntax of the goto process), as well as the actions performed by process P (the set A in the typing for P) that need to be checked in the typing. Therefore, the typing of a goto process checks if the continuation process is well typed and ensures that the target site allows code that travels through that sequence of sites to perform the actions of the continuation process. There is no effect associated with code migration, since the actions that P performs are checked at this level, and so, there is no point in including the actions executed by P to be checked again at outermost levels.

Figure 15 describes network typing. Security policies are not checked at network level, since no computation takes place at this level. Therefore, the only interesting fact to stress (refer to rule N-SITE) is that when code is installed at a certain site, the actions that are not the continuation of a goto process are not checked. Indeed, since functions allows and canEnter are reflexive, there is no point in checking policies at this level.

The type system we present preserves typings during reduction.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash N$ *and* $N \to M$*, then* $\Gamma \vdash M$.

Typing the tagged language. The changes imposed on the type system by tagging are described in figure 16. The typing rules for processes are left unchanged, but at network level we propose substantially different rules. Rule T-SITE checks, among other things, that the tagged typing assumptions are enough to type the process. Moreover, the conclusion's typing environment, Γ, is a relaxed version of the one used for tagging, Δ. Hence, it is possible to consider the minimum security requirements to type a process, and then enlarge the set of security properties at network level.

Notice that we now verify security policies in the rule, since the inclusion of the path travelled by the code $(\widetilde{t})$ represents an implicit goto process. To understand the need to check security policies at site level, consider, for instance, sequent $\Delta \vdash_{s\widetilde{t}} P \colon A$. It is always possible to run a process at its host site: $\Gamma \Vdash s[P]^{s}_{\Delta}$, for $\Gamma <\colon \Delta$. But if we want to indicate that the code migrated from a site, say t, $s[P]^{t}_{\Delta}$, then it is only possible if site s allows t to execute actions in set A and t is able to enter s border.

We check also the declaration of new names, since we are again in presence of an implicit goto. From the network declaration $(\boldsymbol{\nu}_{\tilde{t}}\, n\colon L_{@}s)\ N$ we infer that there was a code migration all along the sequence of sites $\tilde{t}$ to site s, and then afterwards the creation of n took place. The remaining typing rules result from syntax modifications.

Theorem 2 (Tagged subject reduction). *If $\Gamma \Vdash N$ and $N \mapsto M$, then $\Gamma \Vdash M$.*

Type safety. The tagged and untagged reduction relations are closely related. We define a tag function $\mathsf{tag}_{\Gamma}(N)$ in [19] that takes an untagged network N and yields the set of tagged networks obtained from it using Γ.

The following results ensure that types are preserved both by the tagging function and by the tagged reduction.

Theorem 3 (Tagging preserves types). *If $\Gamma \vdash N$, and $M \in \mathsf{tag}_{\Gamma}(N)$, then $\Gamma \Vdash M$.*

Theorem 4 (Operational correspondence between tagged and untagged languagues).

(i) If $N \to N'$, then $\exists M \in \mathsf{tag}_{\Gamma}(N)$ s.t. $M \mapsto M' \in \mathsf{tag}_{\Gamma}(N')$.
(ii) If $M \mapsto M'$, then $\exists N \in \mathsf{tag}_{\Gamma}(N), N' \in \mathsf{tag}_{\Gamma}(M')$ s.t. $N \to N'$.

The type safety result states that well-typed networks do not incur in runtime errors.

Theorem 5 (Type safety). *If $\Gamma \Vdash M$, then $M \overset{err}{\not\mapsto}$.*

5 Conclusions and related work

Summary. We present an approach to express and control history-based access to resources using types. We use Dπ as the underlying calculus and, on top of it, define a hierarchical structure of security groups. The security model we propose is based on the notion of security group that delimits a region of the network with the same security requirements. Security groups may be understood as a firewall that dictates and supervises the sites under its control. We use a type system as the security mechanism to enforce that networks respect the security policies defined by groups and claim a type safety result.

Ongoing work comprise the refinement of the type system to enforce a fine grained control of resources' security and the study of how to change policies dynamically.

Related work. Refer to [3] for a general survey on concurrent mobile calculi, type systems, and security policies. As far as we known, our security model is the first to mix group policies, to record history-based access to resources, and to use a group hierarchy for helping the writing of security rules and the reusing of existent ones.

Cardelli, Ghelli, and Gordon introduced the notion of groups for the Ambient calculus [5–7] to control the movement and the opening of ambients. They use groups to combine ambients in clusters, but specify the security properties for each ambient regardless the group the ambient belongs to. Instead, we use groups to specify security policies shared by the sites that compose each group.

Lhoussaine and Sassone [17] use dependent types as an alternative to groups. The type system is far more complex and the calculus does not facilitates the writing of policies.

The work on Dπ has proposed advanced type systems [14, 15] to control resource access. The control of policies is based on a subtype relation that permits the delivery of different types of the same channel to distinguished parties. Code mobility is controlled with the **mig** keyword. If a process "sees" the **mig** keyword as part of the type of a site, then it may migrate code to that site. The subtype relation, together with the capability to communicate site names, allows for a site to tailor the information (*e.g.* resource names, control keywords) that the target site is able to use. This approach spreads security annotations along the code and it makes difficult to understand what actions are really allowed to execute.

Martins and Ravara [18] presented a type system to control migration in $lsd\pi$ [22] with no site creation. The paper discusses an earlier stage of development of the current work, where there is no notion of groups, nor history-based access control to resources. The works of Abadi and Fournet [1] and of Edjlali, Anurag, and Vipin [11] present a practical application of history based access control to resources. Both works are deeply committed with the frameworks they select to make their security experiments, namely, the Java language, and by this reason, are difficult to compare to the current work. Chothia and Stark [8] present a notion of *local areas* that resemble our group hierarchy, but they just use local areas to ensure that channels are used in the appropriate domain.

The decentralised label model of Myers and Liskov [20] use the notions of *labels* and *principals* to control information flow. These are related with our idea of groups and policies. Labels, assigned to variables, define the information flow policy: the sequence of principals that can read information for each owner. Principals may *act* for other principals, thus forming a hierarchy similar to our group hierarchy. However, we assign policies to groups and manage other policies besides the read policy. Bugliesi, Colazzo and Crafa [4] also work with groups to control information flow for the π-calculus. Channel types record the information carried by channels, as well as the path the channel travels. We use a similar mechanism to keep track of code mobility, but since we control code migration instead of information flow, the assignment of migration paths is to mobile threads rather than to channels.

Finally, KLAIM [9, 10, 12, 13] uses a capability type system to control operations on tuple spaces. The KLAIM approach uses a notion of security policies, which are declared at site level, but differs substantially from our approach in what concerns how policies are programmed and checked. One main distinction is the place where the security policies are defined: security policies in KLAIM talk about what operations a site may perform on other sites, whereas in our framework each security group talks about what actions it allows others to perform on it. From the administrator's point of view this looks more adequate. Recent type systems proposed for μKLAIM tackle the compilation of open systems, using a kind of partial compilation mechanism that marks parts of the processes that cannot be checked statically to be analysed at runtime.

Acknowledgements. The authors would like to acknowledge the financial support of the EU Global Computing project Mikado, and thank Matthew Hennessey and António Ravara for fruitful discussions. The first author would like to thank Neva Slani for the innumerous suggestions and comments given on this work. We also wish to acknowledge the hospitality of the School of Cognitive and Computing Sciences, University of Sussex, during the academic year of 2002/2003.

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of NDSSS'03*, pages 107–121, 2003.
2. G. Boudol. Asynchrony and the π-calculus. Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.
3. G. Boudol, I. Castellani, F. Germain, and M. Lacoste. Models of distribution and mobility: State of the art. Mikado Deliverable D1.1.1, 2002.
4. M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *Proceedings of CONCUR'04*, volume 3170 of *LNCS*, pages 225–239. Springer-Verlag, 2004.
5. L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer-Verlag, 1999.
6. L. Cardelli, G. Ghelli, and A. Gordon. Ambient groups and mobility types. In *Proceedings of TCS'00*, volume 1872 of *LNCS*, pages 333–347. Springer-Verlag, 2000.
7. L. Cardelli and A. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
8. T. Chothia and I. Stark. A distributed pi-calculus with local areas of communication. In *ENTCS*, volume 41.
9. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and mobility. *IEEE Trans. in Software Engineering*, 24(5):315–330, 1998.
10. R. De Nicola, G. Ferrari, R. Pugliese, and B. Veneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.
11. G. Edjlali, A. Anurag, and C. Vipin. History-based access-control for mobile code. In *Proceedings of CCS'98*.
12. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *Proceedings of ICALP'03*, volume 2719 of *LNCS*, pages 119–132. Springer-Verlag, 2003.
13. D. Gorla and R. Pugliese. Controlling data movement in global computing applications. In *Proceedings of SAC'04*. ACM Press, 2004.
14. M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 2003.
15. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.
16. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.
17. C. Lhoussaine and V. Sassone. A dependently typed ambient calculus. In *Proceedings of ESOP'03*, LNCS. Springer-Verlag, 2003.
18. F. Martins and A. Ravara. Typing migration control in $lsd\pi$. In Andrei Sabelfield, editor, *Proceedings of FCS'04*. TUCS, 2004.
19. F. Martins and V. Vasconcelos. Controlling security policies in a distributed environment. DI/FCUL TR 04–01, 2004.
20. A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
21. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
22. A. Ravara, A. Matos, V. Vasconcelos, and L. Lopes. Lexically scoping distribution: what you see is what you get. In *FGC: Foundations of Global Computing*, volume 85(1) of *ENTCS*.
23. E. Zwicky, S. Cooper, and D. Chapman. *Building Internet Firewalls, Second Edition*. OReilly & Associates, 2000.