Secure Safe Ambients
and JVM Security

(Abstract)

Michele Bugliesi
Dipartimento di Informatica
Università Ca' Foscari di Venezia
michele@dsi.unive.it
http://www.dsi.unive.it/~michele

Giuseppe Castagna
C.N.R.S.
Laboratoire d'Informatique, ENS
Giuseppe.Castagna@ens.fr
http://www.di.ens.fr/~castagna

1 Background and Motivations

Safe Ambients are a variant of Cardelli and Gordon's *Mobile Ambients* [CG98] proposed by Levi and Sangiorgi in [LS00]. The two calculi differ in the underlying notion of interaction: in Mobile Ambients interaction between ambients are "one-sided", in that one of the two partners in a *move* or *open* action simply undergoes the action. In Safe ambients, instead, the reduction relation requires actions to synchronize with corresponding co-actions.

Secure Safe Ambients (SSA) are a typed variant of Safe Ambients whose type system is so defined as to allow behavioral invariants of ambients to be expressed and verified. The syntax is defined by the following productions:

$$
\begin{array}{lrcl}
\textit{Processes} & P & ::= & \mathbf{0} \;\big|\; \alpha.P \;\big|\; (\boldsymbol{\nu} a\!:\!D)P \;\big|\; P \mid P \;\big|\; a[P] \;\big|\; !P \\
\textit{Capabilities} & \alpha & ::= & \mathtt{in}\,a \;\big|\; \overline{\mathtt{in}}\,a \;\big|\; \mathtt{out}\,a \;\big|\; \overline{\mathtt{out}}\,a \;\big|\; \mathtt{open}\,a \;\big|\; \overline{\mathtt{open}}\,a
\end{array}
$$

The types of (ambient) names, ranged over by D, are *protection domains*, type-level constants used to identify ambients that satisfy the same behavioral invariants and share common security policies: instead of associating such invariants and policies with each ambient we rather define them for domains, and then group ambients in domains.

The core reduction rules for SSA are exactly those defined for *Safe Ambients*:

$$
\begin{array}{rcl}
b[\mathtt{in}\ a.P \mid Q] \mid a[\overline{\mathtt{in}}\ a.R \mid S] & \rightarrow & a[R \mid S \mid b[P \mid Q]] \\
a[\ b[\mathtt{out}\ a.P \mid Q] \mid \overline{\mathtt{out}}\ a.R \mid S] & \rightarrow & b[P \mid Q] \mid a[R \mid S] \\
\mathtt{open}\ a.P \mid a[\overline{\mathtt{open}}\ a.Q \mid R] & \rightarrow & P \mid Q \mid R
\end{array}
$$

The purpose of the type system is twofold: it provides an abstract characterization of the behavior of ambients and processes, and it allows one to define security policies with ambients and to then statically verify those policies are complied with. For instance, one may specify that entering ambients must not enter some protection domain D (by saying that they not exercise the capability $\mathtt{in}\ a$ for any $a : D$) or that it may enter D but not leave it. Similarly one may state that ambients cannot enter a domain twice (this for example if we want to avoid an extensive testing of our selling policy in case of electronic commerce), or that after entering a domain, an ambient should not visit antagonist or hostile domains.

2 Types

Ambient domains provide the type-level unit of abstraction, and the behavior of processes is characterized in terms of the interactions that processes may have with ambients in different domains. To account for this characterization, we introduce type-level capabilities corresponding to the term-level capabilities. We then define process types in terms of type-level capabilities as follows:

$$\begin{array}{llll} \textit{Type Capabilityes} & M & ::= & \mathtt{in}\ D \mid \mathtt{out}\ D \mid \mathtt{open}\ D \mid \overline{\mathtt{in}}\ D \mid \overline{\mathtt{out}}\ D \mid \overline{\mathtt{open}}\ D \\ \textit{Process Types} & \mathsf{P} & ::= & (\mathbf{M}, \mathbf{N}) \qquad\qquad (\mathbf{M}, \mathbf{N} \in 2^M) \end{array}$$

The intuitive reading of a process type is as follows. If $P : (\mathsf{P}^\uparrow, \mathsf{P}^\downarrow)$ [1], the type capabilities in $\mathsf{P}^\uparrow$ describe the interactions ambients enclosing P may have with ambients of other domains, while $\mathsf{P}^\downarrow$ describes the interactions that P itself may have with (ambients of) its surrounding domains. The distinction in two levels reflects the different effect of exercising a capability. For instance, if $\mathtt{in}\ a.P : (\mathsf{P}^\uparrow, \mathsf{P}^\downarrow)$, and $a : D$, then $\mathtt{in}\ D \in \mathsf{P}^\uparrow$, as the capability $\mathtt{in}\ a$ is exercised on behalf of the ambient enclosing P. Instead, if $a[\mathtt{in}\ b] : (\mathsf{P}^\uparrow, \mathsf{P}^\downarrow)$ with $b : D$, then $\mathtt{in}\ D \in \mathsf{P}^\downarrow$, as it is now $a[\mathtt{in}\ b]$ that directly exercises the capability $\mathtt{in}\ b$. Also, the two levels provide the type system with finer control over process behavior. Take for instance the two processes $P = a[b[\mathtt{in}\ c]]$ and $Q = a[\mathtt{in}\ c]$. These two processes have rather different behaviors, as it is easily verified by considering the composition of P and Q a third process R: the fact that P exercises $\mathtt{in}\ c$ in $P \mid R$ has no effect on R, whereas if Q exercises $\mathtt{in}\ c$ in $Q \mid R$, R is affected if it contains an ambient c willing to be entered. Distinguishing processes such as P and Q at the type level is thus important for reasoning about equivalences and other behavioral properties (for instance, interferences [LS00]).

Having characterized process behavior, we then connect process and ambient types by resorting to *behavior environments* associated with ambient domains.

$$\textit{Behavior Environments} \qquad \Pi : \textit{Ambient Domains} \rightarrow \textit{Process Types}$$

The purpose of behavior environments is twofold. Firstly, they convey information about the interactions that each domain may have with it surrounding domains. Secondly, they ensure a form of coherence between the behavior of ambients belonging to the same domain. Intuitively, given any ambient a of a given domain D, the process type $\Pi(D)$ associated to D by Π defines an upper bound on the capabilities that may be exercised by any process P enclosed in a as in $a[P]$.

[1] We often use the following convention: if $\mathsf{P} = (\mathbf{M}, \mathbf{N})$ is a process type, we write $\mathsf{P}^\uparrow$ for $\mathbf{M}$ and $\mathsf{P}^\downarrow$ for $\mathbf{N}$. Also, abusing this notation, we use $\mathsf{P}^\uparrow$ and $\mathsf{P}^\downarrow$ both as projections of the type P, and directly as sets, as in $P : (\mathsf{P}^\uparrow, \mathsf{P}^\downarrow)$.

Behavior environments are included naturally in the type system: the judgments derived by the typing rules have the form $\Pi, E \vdash P : \mathsf{P}$, where E is a type environment mapping ambient names to domains, Π is a behavior environment, and $\mathsf{Img}(E) \subseteq \mathsf{Dom}(\Pi)$.

2.1 Behavioral Soundness

The operational significance of the type system is established by showing that process types provide a safe approximation of process behavior. To formalize this notion, we introduce a labeled transition system that derives assertions of the form $P \Downarrow \alpha$ whenever process P may (after any number of reduction steps) exercise the capability α. Then we connect the type system with the labeled transition system by a soundness result that may be stated informally as follows (where $\mathtt{cap} \in \{\mathtt{in}\,, \mathtt{out}\,, \mathtt{open}\,, \overline{\mathtt{in}}\,, \overline{\mathtt{out}}\,, \overline{\mathtt{open}}\,\}$):

if $\Pi, E \vdash P : \mathsf{P}$, *and* $P \Downarrow \mathtt{cap}\ a$ *with* $\Pi, E \vdash a : D$, *then* $\mathtt{cap}\ D \in \mathsf{P}$

In other words, if a process P may exercise a capability on a name a, and P is well-typed, then the corresponding type capability is traced by the type of P: thus, no action goes untraced by the type system.

3 Security Environments

Given the characterization of process behaviors provided above, the type system can be extended to establish security policies, and to statically verify that processes comply with those policies. New environments help associate security policies with ambient domains:

Security Environments Σ : *Ambient Domains* $\rightarrow$ *Security Constraints*

The role of security environments is to constrain the behavior of domains. For each domain D, $\Sigma(D)$ controls the possible interactions that ambients of any domain D' may have with ambients of domain D. Security constraints may have different forms depending on the sort of security policy one wishes to express. We discuss two options below: others are possible.

Domain Constraints yield rather coarse protection policies whereby one can identify *trusted* and *untrusted* domains and, for each domain, allows interactions only with trusted domains. These security constraints may be expressed by tables of the following shape, where $\mathscr{D}$ denotes a set of domain names:

$$\mathsf{S} = \langle \mathsf{in} = \mathscr{D}_{\mathsf{in}},\ \mathsf{out} = \mathscr{D}_{\mathsf{out}},\ \mathsf{open} = \mathscr{D}_{\mathsf{open}} \rangle$$

Each of the entries in the table defines a set of *trusted* domains: if D is a domain with $\Sigma(D) = \mathsf{S}$, then only ambients of domain in $\mathscr{D}_{\mathsf{in}}$ (respectively $\mathscr{D}_{\mathsf{out}}$, and $\mathscr{D}_{\mathsf{open}}$) may enter (resp., exit and open) ambients of D.

Capability Constraints lead to finer protection policies that identify the type-level capabilities that entering, exiting and opening ambients may or may not exercise. These constraints may be expressed by the following tables, whose entries are process types.

$$\mathsf{S} = \langle \mathsf{in} = \mathsf{P}_{\mathsf{in}},\ \mathsf{out} = \mathsf{P}_{\mathsf{out}},\ \mathsf{open} = \mathsf{P}_{\mathsf{open}} \rangle$$

If D is a domain, and $\Sigma(D) = \mathsf{S}$ then:

- P_{in} defines the capabilities that a process entering ambients of domain D must *not* not exercise: the two sets $\mathsf{P}^{\downarrow}_{\mathsf{in}}$ and $\mathsf{P}^{\uparrow}_{\mathsf{in}}$ specify the capabilities that cannot be exercised, respectively, at the level of the entering process, and at the level of the enclosing ambient.

- $\mathsf{P}_{\mathsf{out}}$ is the table defining the capabilities that a process exiting out of ambients of domain D must *not* exercise, with the two entries $\mathsf{P}^{\uparrow}_{\mathsf{out}}$ and $\mathsf{P}^{\downarrow}_{\mathsf{out}}$ defined as above.

- $\mathsf{P}_{\mathsf{open}}$ is the table defining the capabilities that a process opening ambients of domain D must *not* exercise.

3.1 Security Soundness

Having introduced security environments, it is now possible, and operationally meaningful, to define a notion of security soundness. Intuitively, a system of ambients and processes in sound with respect to the security policies defined by a security environment, if no process or ambient in the system exercises a capability that violates the security policy established by some domain.

This intuition can be formalized by parameterizing typing judgments over security environments, as in $\Pi, E \vdash_{\Sigma} P : \mathsf{P}$, where $\mathsf{Dom}(\Sigma) = \mathsf{Dom}(\Pi)$. This judgment is derivable if $\Pi, E \vdash P : \mathsf{P}$ is derivable and both P and Π are *consistent* with Σ: the notion of consistency guarantees that neither P nor any ambient domain in Π violate a security constraints defined by Σ. If $\Pi, E \vdash_{\Sigma} P : \mathsf{P}$ is derivable, then we say that P is *secure* with respect to Σ: as a corollary of behavioral soundness, one may then conclude that secure processes will never violate any security assumption of any domain.

4 Distributed version and JVM security

While the type system we have outlined has interesting operational impact, in a highly distributed system it is unrealistic to assume that the typing security assumptions are globally available to all the components of the system. If P is a process over a wide area distributed system, a proof for the judgment $\Pi, E \vdash P : \mathsf{P}$ would imply that Π and E contain assumptions for all the ambients running over the system, and those ambients' groups: this is clearly unrealistic.

As a remedy, we are studying a distributed variant of SSA and of the type system, where each ambient carries along the type, behavior, and security environments used to prove the ambient itself well-typed and secure. The idea is simple: when an ambient, say $b[P]_{(\Sigma,\Pi,E)}$, crosses ambient a, a

first checks that the assumptions carried along by b are compatible with its own assumptions, and then verifies that that b is indeed well-typed and secure under those assumptions.

In fact, a closer look shows that the properties that need to be checked for entering or exiting ambients correspond precisely to the security checks performed by the three components of the security architecture of the first versions of the JVM: the *Class Loader*, the *Bytecode Verifier*, and the *Security Manager* [LY97].

We illustrate the idea by showing the (now typed) reduction for the `in` capability, in the distributed variant of SSA. We use the letter Γ to denote triples of the form (Σ, Π, E). If $\Gamma = (\Sigma, \Pi, E)$ and $\Gamma' = (\Sigma', \Pi', E')$, then we use the notation $\Gamma \vdash J$ for the judgment $\Pi, E \vdash_{\Sigma} J$ and $\Gamma' \cdot \Gamma = (\Sigma' \cdot \Sigma, \Pi' \cdot \Pi, E' \cdot E)$ for the extension of Γ with Γ'. Note that in $\Gamma' \cdot \Gamma$, the bindings in Γ hide the corresponding bindings in Γ'. We can now write the distributed version of the rule for entering. Let $\Gamma,\ \Pi,\ E$, and Σ respectively denote $\Gamma_b \cdot \Gamma_a,\ \Pi_b \cdot \Pi_a,\ E_b \cdot E_a$, and $\Sigma_b \cdot \Sigma_a$, then

$$\textbf{(in)} \qquad b[\mathtt{in}\ a.P \mid Q]_{\Gamma_b} \mid a[\overline{\mathtt{in}}\ a.R \mid S]_{\Gamma_a} \rightarrow a[R \mid S \mid b[P \mid Q]_{\Gamma}]_{\Gamma} \qquad (\Gamma \vdash b[\mathtt{in}\ a.P \mid Q] : \Pi(E(b)))$$

Let us consider the condition fot applying the rule: $\Gamma \vdash b[\mathtt{in}\ a.P \mid Q] : \Pi(E(b))$. It says that (1) we must consider the environment $\Gamma_b \cdot \Gamma_a$; (2) in this environment $b[\mathtt{in}\ a.P \mid Q] : \Pi(E(b))$ must hold; (3) this type is *consistent* with Σ. If we closely look at these three points it is easy to see how each of them corresponds, from the a's viewpoint, to a function implemented by one of the three component of the JVM security system.

$\Gamma = \Gamma_b \cdot \Gamma_a$ **:** The local (of a) type environment is appended to the one of the entering ambient. Therefore an item is first searched in the local ambient and then in the remote one: thanks to this, b cannot spoof a definition of a. This is the security policy implemented by the JVM Class Loader, which provides name spaces separation and prevents type confusion attacks for spoofing.

$b[\mathtt{in}\ a.P \mid Q] : \Pi(E(b))$ **:** The agent a checks that the entering ambient b has the type it declares to have ($b \notin \mathsf{Dom}(E_a)$), or that a expects it to have ($b \in \mathsf{Dom}(E_a)$). This is the security policy enforced by the bytecode verifier.

$\Pi(E(b))$ **is consistent with** Σ **:** The agent a checks that the entering ambient does not perform actions that clash with the security constraints defined by Σ, and Σ_a in particular. This is the security policy enforced by the Security Manager, with the only difference that while the Security Manager performs this check dynamically (when the agent is already entered and requires to perform the action), our system is so elemental that it can do it when the agent requires to enter, that is, from a's viewpoint, at load time.

Finally in order to have a perfect analogy between ambients such as $a[P]_{(\Pi,E,\Sigma)}$ and Java `class` files we must have that $\mathsf{Dom}(E) = \mathit{fn}(a[P])$ and $\mathsf{Dom}(\Pi) = \mathsf{Dom}(\Sigma) = \mathsf{Img}(E)$ since in a Java class file type informations are about just the identifiers occurring in the bytecode. Therefore we must modify the rule above as follows.

Let us write $(\Pi, E, \Sigma)\big|_P$ to denote the environment obtained by deleting from Π all the assignments of names that are not free in P, and from Π and Σ all assignments to domainsdeleted from the image of E.

(in) $$b[\mathtt{in}\ a.P \mid Q]_{\Gamma_b} \mid a[\overline{\mathtt{in}}\ a.R \mid S]_{\Gamma_a} \rightarrow a[R \mid S \mid b[P \mid Q]_{\Gamma|_{b[P \mid Q]}}]_{\Gamma} \qquad (*)$$

$(*)$ $\quad (\, \Gamma \vdash b[\mathtt{in}\ a.P \mid Q] : \Pi(E(b)) \,)$

The interpretation of the rule we gave above still holds. The remaining rules require similar modifications.

References

[CG98] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL'98*. ACM Press, 1998.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL 2000*, pages 352–364. ACM Press, 2000.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.