Mobile Computations and Hostile Hosts

Jan Vitek[1] & Giuseppe Castagna[2]

1: Object Systems Group, CUI
Université de Genève,
Genève, Suisse

2: C.N.R.S,
Laboratoire d'Informatique
de l'Ecole Normale Supérieure,
45 rue d'Ulm, Paris, France

Abstract

This paper scratches the surface of the problem of classifying the attacks that a mobile computation can be subjected to in an open network. The discussion is based on a simplified version of the Seal calculus. We show how the impact of these attacks on the semantics of the calculus and on the notion of observational equivalence.

1. Introduction

Wide area networks hold great promises in terms of distributed computing. Applications that scale to the size of the Internet and seamlessly provide access to massive amounts of information and value added services are now technically feasible. But programming these new applications is proving more difficult than anticipated. The blame lies in part with current distributed programming paradigms which are too static to accommodate the dynamics of an open systems of the size of the Internet.

Mobile agent programming is a promising alternative to client-server computing based on the simple idea that, instead of moving packet of data between stationary programs, a distributed system should be built by moving programs and allowing mobile programs to interact whenever they are located on the same site. Agents are being used for collaborative applications [16] and data mining where the data space is huge. In such applications, moving an agent to the server reduces network connectivity, eliminates latency and overcomes limitations imposed by firewalls. Other application domains include network management [17] and e-commerce [11].

The main technical, and social, obstacle to mobile agents is security. Not only appropriate security mechanisms must be provided by agent execution environments, but users must become confident enough in these mechanisms to permit foreign programs to migrate freely and execute on their machine.

In the last couple of years a number of process calculi have been designed to model aspects of distributed programming. Several of these calculi, including the Ambient calculus [4], the Join calculus [8], dpi [9] and the Seal calculus [21], advocate programming models based on the notion of mobile computations.

Programming distributed applications over open network raises many security issues. They can be roughly categorized into:

- *protection of hosts* and,

- *protection of mobile computations.*

The former corresponds to protecting the resources of the machine on which mobile computations execute. The latter corresponds to protecting the agents themselves from attacks which may originate from an opponent listening on the network, from other mobile computations, and even from hosts themselves. Some of these problems are well understood. Technologies such as sandboxes and software fault isolation or the type systems of Hennessy and Riely [9] have been successful in protecting host from malicious agents. Results such as that of Abadi, Fournet and Gonthier [1] show that it is possible to implement secure communication channels over an untrusted network. The aspect that remains mostly unexplored is the protection of mobile computations from their execution environment.

The goal of this paper is to discuss informally some points in the design space of agent programming languages. We pay particular attention to the *protection* mechanisms that an agent language can provide, and how they can protect agents from environmental attacks.

The structure of this paper is the following. In Section 2, we outline the threat model and state some assumptions about agent systems. In Section 3, we outline the main features of the Seal calculus, a calculus of mobile computations which will be our motivating example. In Section 4, we discuss a number of attacks and related protection mechanisms.

2. Security

The mobile agent paradigm is an interesting case study for the security community. An agent is a program executing in an environment that it does not fully trust, and which interacts with co-located agents which it also does not fully trust. This mistrust is mutual. The reason for this mistrust is the susceptibility of an agent and the kernel to the following kinds of attack:

- **Unauthorized disclosure**: Secret data stored within an agent, the kernel, or the enclosing environment of the kernel is leaked to an unauthorized agent.
- **Unauthorized modifications**: The code or data of an agent is subvertly modified, thus damaging the agent. When such attacks succeed on the kernel, then the whole platform is paralyzed.
- **Denial of service**: An agent reduces the availability of some shared system resource by consuming an inordinate amount. CPU, memory and network bandwidth are classic targets for these attacks.
- **Introduction of viruses and Trojan horses**: An agent or the kernel is tricked into executing malicious code. This code can cause disclosure, modification and denial of service attacks.

The remarkable thing in the case of agent system is the symmetry of the security concerns: both the agent and their execution environment must be protected. The pendant is that, depending on the application, either one of them could have strong economical incentive to breach the security of the other. An agent may seek to obtain data from the site it is running on or damage the site in some way. An execution environment may have been engineered to steal data or corrupt certain agents that migrate to it.

In the following we will differentiate between **agents** which regard as mobile programs written in an agent language, **agent execution environments** which provide the set of services that the agent may invoke during its execution on a host, and **agent runtime** which is the underlying platform that implements the semantics of the agent language.[1]

There are different threats that must be considered when attempting to secure an agent program.

[1] To relate these concepts to Java programming, we could schematically say that an agent corresponds to an Applet, an environment to the JDK libraries and a runtime to the JVM implementation.

- **Exogenous threats**: These are attacks that occur outside of the agent system, either while the agent is transferred over the network or stored on disk.

- **Endogenous threats**: These threats are specific to an agent system.

  - **Horizontal hostility**: Attacks between agents running on the same host.
  - **Vertical hostility**: Attacks against an agent mounted by the execution environment (the libraries) or by the runtime system of the agent platform.

In general, we refer to vertical hostility — attacks originating from the environment or the runtime — as *hostile host* attacks. The remainder of this paper will study protection mechanisms which can be provided by agent languages and their vulnerability to hostile environment. The goal of the study is to highlight language design issues and foster discussion rather than provide definitive answers.

Hostile hosts have been presented quite pessimistically in the mobile agent literature [6]. Even if security requirements are basically symmetrical — with both agent and hosts looking to protect their respective resources, data and services — control is asymmetrical. The host which executes an agent must see the agent's internal workings and has full control the agent. On the face of it, little can be done to protect a program from the very machine that decodes every single one of the program's instructions and executes that instruction. Given enough time, an attacker will be able to analyze the inner workings of any agent and subvert its intended meaning.

This pessimistic view may somewhat be offset by practical considerations. Firstly, one must consider the cost of mounting hostile host attacks and secondly, a host may offer some guarantees that make such attacks less likely. The first part refers to the difficulty of modifying an agent system runtime which is a non-trivial undertaking, and, even more significantly, the cost of analyzing the code of agents in order to know how to attack them. If human intervention is required for each different agent type that migrates to a malicious host, than an apt defense may well lie in program obfuscation and transformation techniques. Collberg et al. [7] propose algorithms to automatically generate families of almost identical agents with different code in ways that are resilient to static analysis techniques. As for guarantees, a host may belong to a reputable organization, say Microsoft (we presume), or be certified by some international organization, or be built using with tamperproof hardware [22, 12]. In those cases, one may trust that the semantics of the runtime be respected.

Modifications of the execution environment are easier to achieve than runtime hacking, in the case of Java such attacks often simply hinge on getting a malicious file in the `CLASSPATH` of a user. These attacks are less powerful than runtime modifications, but can still be used to induce an agent into error or subvert information. But at least, in this class of attacks the semantics of the programming language are not violated. Some agents languages allow agents to nest [4, 21], thus each agent runs within the environment provided by its encapsulating agent. In these languages, a reputable host may well contain some shady agents which will entice visitors to come execute within them and then mount execution environment attacks.

We shall conclude this section by remarking that in a world where large numbers of mobile agents move around to carry out their different tasks. The most dangerous attacks are the ones which can be automated and thus do not require human intervention past the initial coding. We also point to intriguing results that may protect agents from the most adverse of conditions: Tschudin and Sander have had success in their work on computing with encrypted functions (protecting the integrity and privacy of agent from the host) [15], Riordan and Schneier has presented the concept of *clueless agents* as agents that are triggered by signals hidden in the environment in ways that make it quite hard for an attacker to provide the appropriate signals [13] and Roth has discussed cooperating agents which work in groups so that it requires several hosts' cooperation to break a group's secrets [14].

3. A mobile calculus

We base our discussion on a calculus of mobile computations called the Seal$^-$ calculus, this calculus belongs to the family of process calculi that descend from Milner's π-calculus, reader familiar with π will find many similarities in the model. The Seal$^-$ calculus is a stripped down variant of the full Seal calculus presented in [21].

The Seal$^-$ calculus unifies several concepts from distributed programming into three abstractions: **locations**, **processes** and **resources**. Locations are meant to stand for physical places such as those delimited by the boundaries of address spaces, host machines, routers, firewalls, local area networks or wide area networks. Locations also embody logical boundaries such as protection domains, sandboxes and applications. The process abstraction stands for any flow of control such as a thread or operating system process. Finally, resources unify physical resources such as memory locations and peripheral device interfaces with services such as those offered by other applications, the operating system or a runtime system.

Names Names may denote two different kinds of computational entities, seals and channels; names are values and as such can be exchanged in communication. The semantics of the calculus ensures that names can not be manufactured or guessed.

Processes In a process calculus every expression denotes a process — a computation — running in parallel with other processes. The calculus includes all usual forms: the inert process, sequential composition of actions, parallel composition of processes and replication. A process can also be a location with a process body, which we call *seal*.

Locations *Seals* are named, hierarchically–structured, locations. The expression $n[P]$ denotes a seal named n running process P. Since a seal is also a process, then a seal can contain a hierarchy of *subseals* of arbitrary depth. A configuration is depicted in Figure 1, an alternate graphical representation is the configuration tree in the same figure; process-labeled vertices represent seals while edges represent seal inclusion.

Resources The only resources in Seal$^-$ calculus are *channels*. Channels are named computational structures used to synchronize concurrent processes. Channels are located in seals, the calculus provides operations to access the channels of another seal. Processes are restricted to use only the channels for which they know the names.

Interactions Since processes (and ressources) are located, process interactions can be either *remote* or *local*. The Seal$^-$ calculus allows only three distinct patterns of interaction. Two of them are *remote*

$$
\begin{array}{l}
\textsc{Network}[\\
\quad\quad \textsc{HostA}[\,P' \mid \\
\quad\quad\quad\quad \textsc{Sandbox}[\,P \mid \textsc{App1}[\,Q\,]\,] \mid \\
\quad\quad\quad\quad \textsc{App2}[Q']\,] \mid \\
\quad\quad \textsc{HostB}[\,P'' \mid \textsc{App3}[\,Q''\,]\,] \\
\quad\quad]
\end{array}
$$

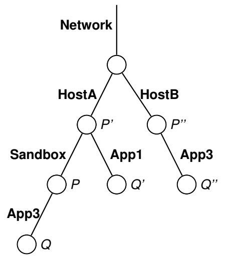
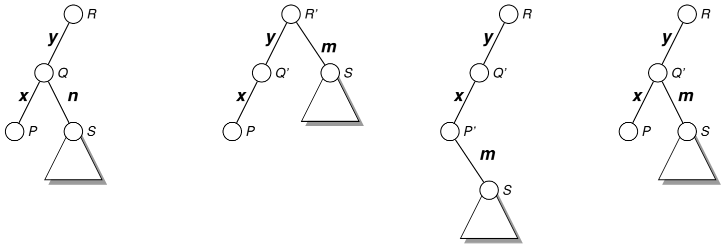


Figure 1: Seal calculus term and configuration tree.

interactions: a process located in the parent synchronizes with a process located in a child on (1) a channel of the child or (2) a channel of the parent. The third one is the *local* interaction: two co-located processes synchronize over a local channel. These interaction patterns are restrictive. For example, they do not allow processes located in sibling seals to communicate directly, and even less those located in arbitrary seals. Communication across a seal configuration must be encoded; in other words every distributed interaction up to packet routing must be programmed. Channel synchronization is used both for communication (the channel is used to pass a name) and for mobility (the channel is used to move a seal), and each of these two forms of interaction corresponds to different actions pairs.

Mobility The calculus allows to move seals along with their body process and subseals over channels. The operator for sending a seal on a channel is $\overline{x}^{\star}\langle y\rangle\,.\,P$, and should be read as a process that is waiting to send a child seal y along channel x and then behave like P. Symmetrically, the operator $x^{\star}\langle z\rangle\,.\,P$ denotes a process waiting to receive a seal along channel x and name it z. On the configuration tree, mobility corresponds to a tree rewriting operation. A move disconnects a subtree rooted at some seal y and grafts it either onto the parent of y, onto one of y children, or back onto y itself. The rewriting operation relabel the edge associated to the moved seal. The diagrams below show an initial configuration (a) and all three possible configurations obtained after a move. (b) is obtained by moving n into the parent and renaming it to m. (c) is obtained by moving n in x and renaming it to m. (d) is obtained by renaming n to m.



(a) (b) (c) (d)

We now give a reduction semantics to the calculus.

3.1. Syntax

We assume infinite sets of *names* and *co-names* disjoint and in bijection via $(\bar{\ })$; we declare $\overline{\overline{x}} = x$. Location denotations extend names with symbols $\star$ and $\uparrow$. Bold font variables denote either a name or the corresponding co-name, thus $\mathbf{x}$ may be either x or $\overline{x}$.

$m, n, \ldots, x, y, z$	names		
$\overline{m}, \overline{n}, \ldots, \overline{x}, \overline{y}, \overline{z}$	co-names	$\eta ::= x \mid \uparrow \mid \star$	locations
$\mathbf{x} ::= x \mid \overline{x}$		X	process variables

The set of *processes*, ranged over by P, Q, R, S, is defined by the following grammar:

$$P \;::=\; \mathbf{0} \quad\mid\quad P \mid Q \quad\mid\quad (\boldsymbol{\nu}\, x)P \quad\mid\quad \alpha \,.\, P \quad\mid\quad !\,P \quad\mid\quad x[P] \quad\mid\quad x[X]$$

$\mathbf{0}$ denotes the inert process. $P \mid Q$ denotes parallel composition. $(\boldsymbol{\nu}\, x)P$ is a restriction. $\alpha \,.\, P$ denotes an action α and a continuation P. $!P$ denotes replication. Finally, $x[P]$ and $x[X]$ denote, respectively, a seal named x with body P and a seal x with body a process variable X.

The set of *actions* is defined by:

$$\alpha \;::=\; \overline{x}^{\eta}(\vec{y}) \quad\mid\quad x^{\eta}(\lambda\vec{y}) \quad\mid\quad \overline{x}^{\eta}\langle y\rangle \quad\mid\quad x^{\eta}\langle y\rangle$$

$\overline{x}^{\eta}(\vec{y}) \,.\, P$ denotes a process offering $\vec{y}$ at channel x located in seal η. $x^{\eta}(\lambda\vec{y}) \,.\, P$ is ready to input names $\vec{y}$ at x in η. $\overline{x}^{\eta}\langle y\rangle \,.\, P$ denotes the sender process offering seal y at x in η. $x^{\eta}\langle y\rangle \,.\, P$ is waiting to read a seal at x in η and run it under names y. Location denotations $\star, \uparrow$ and n denote respectively the current seal, the parent seal, a subseal bearing name n.

We abbreviate $x_1 \ldots x_n$ to $\vec{x}$ and $(\boldsymbol{\nu}\, x_1)\ldots(\boldsymbol{\nu}\, x_n)$ are abbreviated to $(\boldsymbol{\nu}\, \vec{x})$. The location $\star$ is often omitted from channels, so for example $x^{\star}(\lambda\vec{y})$ is abbreviated to $x(\lambda\vec{y})$. $P\{\vec{y}/\vec{x}\}$ denotes simultaneous substitutions of distinct names $x_1 \ldots x_n$ by $y_1 \ldots y_n$. σ ranges over substitutions.

3.2. Reduction semantics

The *reduction relation* is defined using by the means of two auxiliary notions: *structural congruence* and *heating*. Structural congruence, $\equiv$, is the least congruence relation on processes satisfying the following axioms and rules:

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad !\,P \equiv P \mid !\,P$$

Structural congruence does not deal with the extrusion of free names. Like in the π-calculus, we need rules to extrude free names, and in particular to extrude them across seal boundaries. Consider:

$$x^n(\lambda y)\,.\,\overline{y}() \mid n[\,(\boldsymbol{\nu}\,z)\overline{x}^{\uparrow}(z)\,]$$

We expect to reduce the term to:

$$(\boldsymbol{\nu}\,z)(\overline{z}() \mid n[\mathbf{0}])$$

Interestingly, name extrusion across seal boundaries has also security implications and thus warrants special treatment. On a more pragmatic level, in a typical implementation, when a seal boundary coincides with an address space boundary, extrusion has a computational signification: it means local names must be promoted to global names. In order to perform this extrusion, we define a *heating* relation on terms. A term is "heated" to allow synchronization. Heating singles out all the $\boldsymbol{\nu}$−abstractions that must be extruded, that is, those that bind arguments of the output action about to be performed. Heating will extrude as few $\boldsymbol{\nu}$-abstractions as possible. So for example the term $x^n(\lambda y)\,.\,\overline{y}() \mid n[(\boldsymbol{\nu}\,w)(\boldsymbol{\nu}\,z)\overline{x}^{\uparrow}(z)]$ reduces to $(\boldsymbol{\nu}\,z)(\overline{z}() \mid n[(\boldsymbol{\nu}\,w)\mathbf{0}])$ rather than to $(\boldsymbol{\nu}\,w)(\boldsymbol{\nu}\,z)(\overline{z}() \mid n[\mathbf{0}])$. A term in heated form is called an *agent*. Agents are written ωP where ω is an agent prefix and P is a process. The set of *agent prefixes* ranged over by ω is defined by the following grammar:

ω	$::=$	$\boldsymbol{\epsilon}$	empty prefix	$\mid$	$(\boldsymbol{\nu}\,\vec{x})\langle\vec{y}\rangle$	name concretion
	$\mid$	$(\boldsymbol{\nu}\,\vec{x})\langle P\rangle$	process concretion	$\mid$	$\langle\lambda\vec{y}\rangle$	name abstraction
	$\mid$	$\langle\lambda X\rangle$	process abstraction			

Free names (*fn*) have a standard definition and alpha-conversions are performed whenever needed.

The heating relation $\prec$ relates a well-formed process to a term of the form $\mathbf{x}^{\eta}\,.\,\omega P$ and is defined as the least relation respecting the following axioms and rules:

Table 1: Heating.

$\overline{x}^{\eta}(\vec{y}).P \prec \overline{x}^{\eta}.\langle\vec{y}\rangle P$

$x^{\eta}(\lambda\vec{y}).P \prec x^{\eta}.\langle\lambda\vec{y}\rangle P$

$\overline{x}^{\eta}\langle y\rangle.P \mid y[Q] \prec \overline{x}^{\eta}.\langle Q\rangle P$

$x^{\eta}\langle y\rangle.P \prec x^{\eta}.\langle\lambda X\rangle(P \mid y[X])$

$y \notin fn(\omega), y \notin \{x,\eta\}, P \prec \mathbf{x}^{\boldsymbol{\eta}}.\omega P' \;\Rightarrow\; (\boldsymbol{\nu}\, y)P \prec \mathbf{x}^{\boldsymbol{\eta}}.\omega(\boldsymbol{\nu}\, y)P'$

$y \in fn(\omega), y \notin \{x,\eta\}, P \prec \mathbf{x}^{\boldsymbol{\eta}}.\omega P' \;\Rightarrow\; (\boldsymbol{\nu}\, y)P \prec \mathbf{x}^{\boldsymbol{\eta}}.(\boldsymbol{\nu}\, y)\omega P'$

$bn(\omega) \cap fn(Q) = \emptyset, P \prec \mathbf{x}^{\boldsymbol{\eta}}.\omega P' \;\Rightarrow\; P \mid Q \prec \mathbf{x}^{\boldsymbol{\eta}}.\omega(P' \mid Q)$

$y \notin bn(\omega), P \prec \mathbf{x}^{\uparrow}.\omega P' \;\Rightarrow\; y[P] \prec \mathbf{x}^{\overline{y}}.\omega\, y[P']$

The first two axioms handle communication. The fourth axiom says that a receive action heats into an abstraction where the process variable X stands for the body of the seals specified by y, after synchronization the residual consists of the continuation P in parallel with the seal where X has been substituted by some process Q. The following two rules select the names that will be extruded. The last rule allows actions originating from a seal y to synchronize with matching actions in the parent, the action label is changed from $x^{\uparrow}$ to $x^{\overline{y}}$ to prevent further propagation.

We define the reduction relation $\rightarrow$ as the least relation on well-formed processes that satisfies:

Table 2: Reduction.

$$\frac{P \rightarrow Q}{(\boldsymbol{\nu}\, x)P \rightarrow (\boldsymbol{\nu}\, x)Q} \qquad \frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R} \qquad \frac{P \rightarrow Q}{x[P] \rightarrow x[Q]}$$

$$\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$$

$$\frac{P \prec x^{\star}.\omega_1 P' \quad Q \prec \overline{x}^{\star}.\omega_2 Q'}{P \mid Q \rightarrow (\omega_1 P') \bullet (\omega_2 Q')} \qquad \frac{P \prec \mathbf{x}^{y}.\omega_1 P' \quad Q \prec \overline{\mathbf{x}}^{\overline{y}}.\omega_2 Q'}{P \mid Q \rightarrow (\omega_1 P') \bullet (\omega_2 Q')}$$

The *pseudoapplication* relation $(\_) \bullet (\_)$ used in the definition of synchronization is a partial *commutative* binary function from agents to processes. Let $\vec{y}, \vec{x}$ be vectors of the same arity and $\vec{x} \notin fn(P)$, then we define pseudoapplication:

$$\begin{array}{lll} 1. & (\langle\lambda\vec{y}\rangle P) \bullet ((\boldsymbol{\nu}\,\vec{x})\langle\vec{z}\rangle Q) & = & (\boldsymbol{\nu}\,\vec{x})(P\{^{\vec{z}}/_{\vec{y}}\} \mid Q) \\ 2. & (\langle\lambda X\rangle P) \bullet ((\boldsymbol{\nu}\,\vec{x})\langle R\rangle Q) & = & (\boldsymbol{\nu}\,\vec{x})(P\{^{R}/_{X}\} \mid Q) \\ 3. & \text{Undefined otherwise.} \end{array}$$

The core of the semantics is given by the last two rules. They describe synchronization on a channel that is respectively local or remote.

3.3. Equivalences

A notion of observational equivalence based on a bisimulation may be defined for the Seal calculus. We give a commitment semantics (defined in the appendix) that relates a process P to a label ℓ and an agent ωQ (written: $P \xrightarrow{\ell} \omega Q$) and can be proved equivalent to the reduction semantics (see [21]) and choose bisimilarity as it is a widely adopted notion of process equivalence [10]. Note, though, that our notion is higher order as we are sending processes over channels, the definition is thus related to Thomsen's Higher Order Applicative Bisimulation [18, 3].

If $\mathcal{R}$ is a relation on well formed processes, then we define the relation $\mathcal{R}_{[]}$ on processes such that $P\ \mathcal{R}_{[]}\ P'$ iff for any x we have $x[P]\ \mathcal{R}\ x[P']$, and we also define the relation $\overline{\mathcal{R}}$ on agents:

$$\begin{array}{rcll} \boldsymbol{\epsilon} P & \overline{\mathcal{R}} & \boldsymbol{\epsilon} P' & \text{iff} \quad P\ \mathcal{R}\ P' \\ \langle\lambda\vec{x}\rangle P & \overline{\mathcal{R}} & \langle\lambda\vec{x}\rangle P' & \text{iff} \quad \forall\sigma.dom(\sigma) = \vec{x} \wedge P\sigma\ \mathcal{R}\ P'\sigma \\ (\boldsymbol{\nu}\,\vec{y})\langle\vec{x}\rangle P & \overline{\mathcal{R}} & (\boldsymbol{\nu}\,\vec{y})\langle\vec{x}\rangle P' & \text{iff} \quad P\ \mathcal{R}\ P' \\ \langle\lambda X\rangle P & \overline{\mathcal{R}} & \langle\lambda X\rangle P' & \text{iff} \quad \forall Q.P\{^{Q}/_{X}\}\ \mathcal{R}\ P'\{^{Q}/_{X}\} \\ (\boldsymbol{\nu}\,\vec{x})\langle Q\rangle P & \overline{\mathcal{R}} & (\boldsymbol{\nu}\,\vec{x})\langle Q\rangle P' & \text{iff} \quad P\ \mathcal{R}\ P' \wedge Q\ \mathcal{R}_{[]}\ Q' \end{array}$$

A relation $\mathcal{R}$ is a *strong simulation* if $P\ \mathcal{R}\ P'$ and $P \xrightarrow{\ell} \omega Q$ implies that there exists $\omega' Q'$ such that $P' \xrightarrow{\ell} \omega' Q'$ and $\omega Q\ \overline{\mathcal{R}}\ \omega' Q'$. A relation $\mathcal{R}$ on processes is a *strong bisimulation if* $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong simulations. *Strong bisimilarity* $\sim$ is the greatest strong bisimulation. Let $P \overset{\ell}{\Longrightarrow} \omega Q$ if either $P \xrightarrow{\ell} \omega Q$ holds or there exists a P' such that $P \xrightarrow{\tau} \boldsymbol{\epsilon} P'$ and $P' \overset{\ell}{\Longrightarrow} \omega Q$. A relation $\mathcal{R}$ is a *weak simulation* if $P\ \mathcal{R}\ P'$ and $P \xrightarrow{\ell} \omega Q$ implies there exists $\omega' Q'$ such that $P' \overset{\ell}{\Longrightarrow} \omega' Q'$ and $\omega Q\ \overline{\mathcal{R}}\ \omega' Q'$. A relation $\mathcal{R}$ on processes is a *bisimulation* if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations. Bisimilarity $\approx$ is the greatest weak bisimilarity.

4. Protection

The Seal$^-$ calculus has been designed to provide protection for agents against attacks from other agents and attacks from the agent execution environment (in our case, enclosing seals).

This section presents three protection mechanisms and then gives three examples of attacks and their impact on language design.

4.1. Names and secrets

The calculus emphasizes the role of names, they are used to name seals and to name channels of communication. Without knowing the name on which a process wishes to interact, no communication is possible.

Thus, restrictions, terms of the form $(\boldsymbol{\nu}\, x)P$, can be viewed as language enforced protection mechanisms. The semantics of the calculus guarantees that no other process than P may guess x. In practice this means that x may be some large, randomly selected, number which has a high probability of being unique.

The semantics allow these secrets to be exchanged between processes, so the term $z(\lambda y).P \mid (\boldsymbol{\nu}\, x)\overline{z}(x).Q$ reduces in one step to $(\boldsymbol{\nu}\, x)\ (P\{^x/_y\} \mid Q)$.

Alternatively, x can be regarded as a shared cryptographic key, similarly to Abadi's and Gordon's spi calculus [2]. Assuming a value z and a shared key x, we can model the ciphertext $K_x(z)$ by the seal

$$y[!\, \overline{x}^{\uparrow}(z)]$$

It is only by knowing x that an interlocutor may learn z.

Names can be thus used to test an environment, if the environment knows certain names than a degree of trust can be established.

4.2. Encapsulation boundaries

The process running within a seal are always protected by their seal's boundary. They can not escape to wreak havoc, but nor can the environment peek and poke in the seal's internals.

The interactions with the environment are limited and clearly identified to communication of the form $x^{\uparrow}(\lambda y)$, $\overline{x}^{\uparrow}(y)$, $x^{\uparrow}\langle y\rangle$ and $\overline{x}^{\uparrow}\langle y\rangle$. The environment can not interfere with any other term. Thus for example, the seal $w[{}^z(\lambda y).P \mid (\boldsymbol{\nu}\, x){}^z(x).Q]$ will always reduce in one step to $w[(\boldsymbol{\nu}\, x)(P\{^x/_y\} \mid Q)]$ irregardless of the execution context.

4.3. Linear movement

Seals in our calculus, just like the Ambients of Cardelli and Gordon, move in a linear fashion. That is, when a seal is sent along a channel it ceases to exist at the source and materializes at the destination. For example the following term moves *joe* inside a *bus* agent through a channel named *door*:

$$\overline{door}^{bus}\langle joe\rangle \mid joe[P] \mid bus[door^{\uparrow}\langle joe\rangle] \quad \rightarrow \quad bus[joe[P]]$$

This allows some patterns of programming such as using seals to hold linear resources. For instance, if we write the term

$$y[\ \overline{x}^{\uparrow}(v)\ ]$$

we know that seal will send a *single* message along x. As another example, a seal that decrypts the encrypted seal $(c[\mathbf{!}\, x \uparrow v])$ only once is coded as:

$$decrypt[door^{\uparrow}\langle c\rangle \,.\, x^{c}(\lambda z) \,.\, \overline{door}^{\uparrow}\langle c\rangle \,.\, \overline{door}^{\uparrow}(z)]$$

Here the seal receives the encrypted value from its parent and names it c. Then it uses the key x to decrypt c and obtain the secret z. Finally it sends back c and z to the parent. All these communications are performed through a channel named *door*.

The semantics of the calculus guarantee that this seal can be given freely and that it will only perform its service once.

4.4. Denial of service attacks

The Seal^{-} calculus requires synchronous agreement of senders and receivers before a message exchange is allowed to go through. Thus for example $\overline{x}^{y}(z) \mid y[x^{\uparrow}(\lambda w)]$ may reduce since there are matching output and input offers.

Another calculus design which we are considering for seals is to have asynchronous message passing. Thus we would have rules of the form:

$$\overline{x}^{y}(z) \mid y[P] \rightarrow y[\overline{\overline{x}^{\uparrow}}(z) \mid P]$$

where $\overline{\overline{\mathbf{x}}^{\uparrow}}$ synchronizes with $\mathbf{x}^{\uparrow}$ Such a formulation simplifies the semantics but at the cost of allowing a denial of service attack which can be mounted by a parent flooding a seal with messages. (In an actual implementation [20] the cost of holding and managing large amounts of message is great, further when the seal migrates it will require larger amounts of bandwidth.)

4.5. Trojan horse attacks

Trojan horse attacks against seals can be mounted by a malicious execution environment whenever a seal accepts to input another seal. The new subseal may have been engineered to cause some damage by missbehaving. Still those attacks are somewhat limited in scope since the seal boundaries still protect the victim.

A much more serious attack can be mounted by a malicious runtime system that would change the heating rules to inject arbitrary processes into incoming seals:

$$x^{\eta}\langle y\rangle + Q\,.\,P \;\prec\; x^{\eta}.\big\langle \lambda X\big\rangle(P\,|\,y[X \mid Q])$$

The process Q is put in parallel with the unsuspecting seal and allowed to interfere with the insides of the seal.

4.6. Replay attacks

A replay attack occurs if a host is able to duplicate a seal, i.e., it requires an operation such as COPY x AS y so that the following reduction be possible:

$$\text{COPY } x \text{ AS } y \mid x[\,\overline{k}^{\uparrow}(v)] \quad\rightarrow\quad x[\,\overline{k}^{\uparrow}(v)] \mid y[\,\overline{k}^{\uparrow}(v)]$$

With such an operation, a host may try to automatically break the security of incoming agents (in this case the cryptographic key k) by making multiple copies and playing several pre-recorded interaction sequences. Simple observation of the response of an agent to messages may be enough to understand an agent's encoding.[2] Clearly, replay attacks forbid using seals to implement linear resources.

We now show how to extends the calculus to replay attacks into account. In fact the extension is modest. The only thing we must change is the seal receive action which becomes:

$$x^{\eta}\langle\vec{y}\rangle$$

(instead of $x^{\eta}\langle y\rangle$) that is, instead of creating a single instance of the seal, we allow the creation of several instances. Thus the term $x\langle n\,m\rangle$ means receive a seal along channel x and instantiate two identical copies under names n and m.

[2] A slightly naive example involves a shopping agent sent to buy plane tickets on the behalf of the user. A platform may try to offer the same ticket at several different prices in order to learn how much the agent is ready to pay. The interesting point here is that the attack is trivial to program and can be played out automatically without fear of detection.

The changes to the semantics are equally modest. In the case of the reduction semantics, we simply change one rule in the definition of heating:

$$x^{\eta}\langle y_1 \ldots y_n \rangle \,.\, P \prec x^{\eta} \,.\, \big\langle \lambda X \big\rangle (P \,|\, y_1[X] \mid \ldots \mid y_n[X])$$

and perform a similar change for the commitment semantics. The definition of bisimulation is not affected by this change.

In such an extended calculus, COPY is a derived operation defined as follows.

$$\mathrm{COPY}\, x \,\mathrm{AS}\, z \quad \equiv \quad (\boldsymbol{\nu}\, y)(\, \overline{y}^{\star}\langle x \rangle \;\mid\; y^{\star}\langle x\, z \rangle \,.\, P\,)$$

Intuitively the result of $\mathrm{COPY}\, n \,\mathrm{AS}\, m \mid n[P]$ reduces to $m[P] \mid n[P]$. More precisely, the operation first creates a brand new channel name y to prevent any other process running in parallel from interfering with the protocol. Then, the subprocess on the right tries to move n on the local channel, while the one on the left waits to receive the seal and instantiate two copies of it, one named n and the other named m.

In the full Seal calculus, we have chosen to incorporate copying in the semantics; we feel that there are legitimate uses of copying such as the ability to replicate services dynamically for fault tolerance or to checkpoint seals and provide transparent persistence. Further, it may be very hard for an implementation of the calculus to prevent copies from arising due to for instance machine failures and subsequent restart and duplication on the transport layer.

4.7. Breach of privacy attacks

Name passing calculi such the π-calculus, Ambients, the Seal^{-} calculus and Join strongly rely on static scoping of names. In the π-calculus, we expect the following equivalence to hold $(\boldsymbol{\nu}\, x)\overline{x}(y) \;\sim\; \mathbf{0}$, that is, a process trying to output on a restricted name can not be distinguished with the inert process. In the ambient calculus, we have the so-called, perfect firewall equation, $x \notin fn(P) \;\Rightarrow\; (\boldsymbol{\nu}\, x)x[P] \;\approx\; \mathbf{0}$, which states that an ambient whose name is restricted is bisimilar to the inert process provided the restricted name does not occur free in the ambient's body, in our calculus this condition is not required as discussed in [5].

Static scoping is preserved by the semantics of these calculi which guarantee that names can not be guess or counterfeited. This guarantee can also be used to model cryptography by names as in the spi calculus [2]. Static scoping has an added advantage, it simplifies analysis of system, as we have the guarantee that the initial interaction with the outside must take place on free names.

As we have also seen in the semantics, scoping is static but the position of restrictions may vary in the course of computation due to the phenomenon of scope extrusion. For example, the expression $(\boldsymbol{\nu}\, y)\overline{x}(y)\,.\,P$ is ready to send a $\boldsymbol{\nu}-$abstracted name y along a channel x, after synchronization we expect that the scope of the restriction will encompass the residual of the above term along with the receiver. The difficulty with scope extrusion is that as long as the scope of a name remains within a trusted environment, then we can be sure that static scoping is preserved. On the other hand, as soon as restriction spans several sites the guarantees become more difficult to ensure.

Breach of privacy attacks occur if an environment is able to guess names that occur free within an agent. For example, assume a $\text{FN}\; x\langle\lambda y\rangle\,.\,P$ operation which returns some free name y occurring within seal x. This would allow the following reduction:

$$\text{FN}\; x\langle\lambda y\rangle\,.\,P \;\mid\; x[\overline{K}^{\uparrow}(v)] \quad\to\quad P\{{}^{K}\!/\!_{y}\} \;\mid\; x[\overline{K}^{\uparrow}(v)]$$

To obtain this in the calculus we may add FN as an action and add a rule which, for the commitment semantics, does the following:

$$\text{FN}\; n\langle\lambda y\rangle\,.\,P \mid n[Q] \xrightarrow{\tau} P\{{}^{x}\!/\!_{y}\} \mid n[Q] \qquad\qquad \text{for } x \in \mathit{fn}(Q)$$

A side effect of this rule is that the perfect firewall equation does not hold anymore and that it is therefore not possible to rely on privacy of names. Consider the following terms: $x[(\boldsymbol{\nu}\, y)y[\overline{K}^{\uparrow}(v)]] \;\approx\; x[\mathbf{0}]$ in the Seal^{-} calculus they are bisimilar, but with the addition of FN we can distinguish them in the following context:

$$\text{FN}\; x\langle\lambda z\rangle\,.\,P \;\mid\; x[(\boldsymbol{\nu}\, y)y[\overline{K}^{\uparrow}(v)]] \quad\not\approx\quad \text{FN}\; x\langle\lambda z\rangle\,.\,P \;\mid\; x[\mathbf{0}]$$

because $\text{FN}\; x\langle\lambda z\rangle\,.\,P \;\mid\; x[(\boldsymbol{\nu}\, y)y[\overline{K}^{\uparrow}(v)]]$ reduces to $P\{{}^{K}\!/\!_{z}\} \;\mid\; x[(\boldsymbol{\nu}\, y)y[\overline{K}^{\uparrow}(v)]]$.

In fact, we must take a finer notion of bisimulation, let's call it $\approx_f$ which we define as follows:

$$P \approx_f Q \text{ if and only if } P \approx Q \wedge \mathit{fn}(P) = \mathit{fn}(Q)$$

We conjecture that $\approx_f$ is a congruence.

While we do not want to add FN as a regular operator to the calculus, it has no legitimate use, a more sensible variant is *fn* which performs a free variable test. That is, it does not reveal names that were not known beforehand, it merely checks for the presence of a given free name. With such an operator we could check, before allowing a seal to migrate away, whether the seal contains some restricted names (such as a key). So for example we could write:

$$\mathit{fn}\, x\ K\,.\,P \mid x[(\boldsymbol{\nu}\, y)y[\overline{K}^{\uparrow}(v)]] \quad \rightarrow \quad P \mid x[(\boldsymbol{\nu}\, y)y[\overline{K}^{\uparrow}(v)]]$$

The corresponding rule in the commitment semantics would be:

$$\mathit{fn}\, n\ x\,.\,P \mid n[Q] \xrightarrow{\tau} P \mid n[Q] \qquad\qquad \text{for } x \in \mathit{fn}(Q)$$

We conclude by remarking that while $\approx_f$ is strictly finer than $\approx$ it only differentiates on *unreachable* free names, that is, free names that can never be part of normal reduction, $\approx$ does already differentiates on (reachable) free names, thus $x(\lambda z) \not\approx y(\lambda z)$.

5. Conclusions

Mobile computations are an exiting paradigm for structuring distributed applications; their main drawback comes from a perceived lack of security. This paper has discussed some of the protection mechanisms that can be provided by agent languages and discussed their implications.

References

[1] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. 1998. Long version (Draft).

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich, April 1997*. ACM Press, 1997.

[3] L. Cardelli and A. D. Gordon. Mobile Ambients–annex–. Manuscript, Microsoft Research, 1997.

[4] L. Cardelli and A. D. Gordon. Mobile Ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, number 1378 in LNCE, pages 140—155. Springer-Verlag, 1998.

[5] G. Castagna and J. Vitek. Confinement and commitment for the seal calculus. Nov. 1998.

[6] D. M. Chess. Security issues in mobile code systems. In *Mobile agents and security* [19].

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, San Diego, California, 19–21 Jan. 1998.

[8] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR96*, pages 406–421, 1996.

[9] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Proceedings of the Workshop on Internet Programming Languages, (WIPL).* Chicago, Ill., 1998.

[10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.

[11] J.-H. Morin. HyperNews: a Hyper–Media Electronic Newspaper based on Agents. In *Proceedings of HICSS-31, Hawai International Conference on System Sciences*, pages 58–67, Kona, Hawaii, January 1998.

[12] E. R. Palmer. Introduction to citadel: a secure crypto coprocessor for workstations. In Technical Committee TC 11 of the International Federation for Information Processing IFIP, editor, *10th International Information Security Conference IFIP SEC'94*, Curacao, Dutch Antilles, May 1994.

[13] J. Riordan and B. Schneier. Environmental key generation towards clueless agents. In *Mobile agents and security* [19].

[14] V. Roth. Mutual protection of cooperating agents. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects.* Jan Vitek and Christian Jensen (Eds.), Springer Verlag, 1999.

[15] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile agents and security* [19].

[16] D. Spoonhower, G. Czajkowski, C. Hawblitzel, C.-C. Chang, D. Hu, and T. von Eicken. Design and evaluation of an extensible web & telephony server based on the J-kernel. Technical Report TR98-1715, Cornell University, Computer Science, Nov. 4, 1998.

[17] D. Tennenhouse. Active networks. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, Berkeley, CA, USA, Oct. 1996. USENIX.

[18] B. Thomsen. Plain CHOCS. A second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, 1993.

[19] G. Vigna. *Mobile agents and security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1998.

[20] J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal: or How to make Java safe for agents. In D. Tsichritzis, editor, *Electronic Commerce Objects*. University of Geneva, 1998.

[21] J. Vitek and G. Castagna. A calculus of secure mobile computations. In *WIPL98 — Workshop on Internet Programming Languages*. Chicago, Ill., Spinger-Verlag, March 1998.

[22] B. S. Yee. A sanctuary for mobile agents. Technical Report CS97-537, UC San Diego, Department of Computer Science and Engineering, Apr. 1997.

Appendix: Commitment semantics

Commitment is indexed by a set of *labels*

$$\ell \quad ::= \quad \tau \quad | \quad x\square \quad | \quad \mathbf{x}^{\eta} \quad | \quad \overline{x}^{\eta}\langle y\rangle$$

and is the smallest relation satisfying following groups of rules.

Structure:

$$\frac{P \equiv P' \quad P' \xrightarrow{\ell} \omega Q}{P \xrightarrow{\ell} \omega Q} \qquad \frac{P \xrightarrow{\ell} \omega Q}{P \mid R \xrightarrow{\ell} \omega(Q \mid R)}\; bn(\omega)\cap fn(R)=\emptyset$$

$$\frac{P \xrightarrow{\ell} \omega Q}{(\boldsymbol{\nu}\, x)P \xrightarrow{\ell} \omega(\boldsymbol{\nu}\, x)Q}\; x\notin fn(\ell)\cup fn(\omega) \qquad \frac{P \xrightarrow{\ell} \omega Q}{(\boldsymbol{\nu}\, x)P \xrightarrow{\ell} (\boldsymbol{\nu}\, x)\omega Q}\; x\notin fn(\ell)\wedge x\in fn(\omega)$$

Seals:

$$\frac{P \xrightarrow{\mathbf{x}^{\uparrow}} \omega Q}{y[P] \xrightarrow{\mathbf{x}^{y\square}} \omega\, y[Q]}\; y\notin bn(\omega) \qquad \frac{}{x[P] \xrightarrow{x\square} \langle P\rangle \mathbf{0}} \qquad \frac{P \xrightarrow{\tau} \boldsymbol{\epsilon} Q}{x[P] \xrightarrow{\tau} \boldsymbol{\epsilon}\, x[Q]}$$

Communication:

$$\frac{}{\overline{x}^{\eta}(\vec{y})\,.\,P \xrightarrow{\overline{x}^{\eta}} \langle\vec{y}\rangle P} \qquad \frac{}{x^{\eta}(\lambda\vec{y})\,.\,P \xrightarrow{x^{\eta}} \langle\lambda\vec{y}\rangle P}$$

$$\frac{P \xrightarrow{x^{\star}} \omega_1 P' \quad Q \xrightarrow{\overline{x}^{\star}} \omega_2 Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\epsilon}\ (\omega_1 P') \bullet (\omega_2 Q')} \qquad \frac{P \xrightarrow{x^{y}} \omega_1 P' \quad Q \xrightarrow{\overline{x}^{y}} \omega_2 Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\epsilon}\ (\omega_1 P') \bullet (\omega_2 Q')}$$

Mobility:

$$\frac{}{\overline{x}^{\eta}\langle y\rangle\,.\,P \xrightarrow{\overline{x}^{\eta}\langle y\rangle} \boldsymbol{\epsilon} P} \qquad \frac{}{x^{\eta}\langle y\rangle\,.\,P \xrightarrow{x^{\eta}} \langle\lambda X\rangle(P \mid y[X])}$$

$$\frac{P \xrightarrow{\overline{x}^{\eta}\langle y\rangle} \boldsymbol{\epsilon} P' \quad Q \xrightarrow{y\Box} \omega Q'}{P \mid Q \xrightarrow{\overline{x}^{\eta}} \omega(P' \mid Q')}\ bn(\omega)\cap fn(P')=\emptyset$$