The JavaSeal Mobile Agent Kernel

Vitek, Jan; Bryce, Ciaran

The JavaSeal Mobile Agent Kernel

Jan Vitek
Ciarán Bryce

Abstract

JavaSeal is a secure mobile agent kernel that provides a small and coherent set of abstractions for constructing agent applications. This paper describes the design of these abstractions and their implementation. We address the limitations of the Java security model and present a medium-sized e-commerce application that runs over JavaSeal.

1 Introduction

Mobile agent systems come in all shapes and sizes. In fact, there is little consensus over the services that an agent system should offer, or on the exact nature of mobile agents for that matter. Recent standardization efforts notwithstanding [28, 13], most agent systems are hardly comparable and even less compatible. While variety fosters new ideas, most projects end up having to solve similar problems and leave some of the same key questions unanswered:

- **Structure:** What software structuring principles are appropriate for mobile agents? The distinction between mobile and immobile software components must be clarified. Further, which services should be provided in the agent platform and which can be coded at user-level?
- **Security:** Execution guarantees are essential. There is a wide consensus on the need for security, but few agent systems provide clear statements of their meaning of security. Even less provide strong security guarantees.

This paper reports on our experience in implementing and using a Java-based agent kernel. The JavaSeal system has been designed to support a minimal set of abstractions needed for building mobile agent applications. We chose to focus on providing a clear way to structure mobile programs and to enforce specific security constraints. JavaSeal abstractions can be categorized into three groups:

1. Software units called *seals* which are nested, encapsulated, programs. Seals are used for mobile agents and for local services.
2. State capture mechanisms and a custom archive format for seals. This is used for wrapping up seals for mobility and persistence.
3. Secure communication primitives for seal communication.

*To appear in the Proceedings of the Joint Symposium ASA/MA'99, First International Symposium on Agent Systems and Applications (ASA'99) and Third International Symposium on Mobile Agents (MA'99) Palm Springs, California, October 3 - 6, 1999.

This kernel approach is visible in the implementation of services, *e.g.*, the network interface or the graphical user interface, which are user-level modules that can be loaded dynamically as seals. The advantage of this approach is that with a simple programming model, it is easier to reason about the properties of mobile programs. In fact, a related project is investigating formal proof techniques for agent systems [35, 34]. This project has defined a formal semantics of JavaSeal as a process calculus and has been able to validate some security properties, for example confinement by formal proofs [34, 31].

We begin by clarifying our use of terminology. A mobile agent *platform* is an execution environment for mobile agents. A platform is located on a single network node, several platforms connected by a communication infrastructure form a mobile agent *network*. A mobile agent is a program, in our case a multi-threaded program, that executes on a platform and may migrate to another platform in the agent system. Migration means that the data and code of the agent will be available to continue its computation on the new platform.

A JavaSeal kernel provides the core logic for an agent platform. We assume a medium-grained agent model in which every network node may host several thousand concurrently executing agents. Furthermore, agents are allowed to interact, but these interactions must be subject to a security policy.

In JavaSeal, the agents of a platform are organised in a hierarchy. The kernel is at the root of this hierarchy. Each agent can have agents nested inside of it. This hierarchy is important for aggregation, that is, to build a composite agent out of other existing agents. Aggregation is a feature of mobile agent networks because often a computation that is moved needs to have part of its environment moved with it, *e.g.*, open file descriptors, sockets for services. In JavaSeal, this is achieved by representing an environment as a seal, and the computations of that environment as children seals within this seal.

JavaSeal is written in Java as a package and runs over a single virtual machine. This design choice favours portability and means that services can be shared without having to provoke context switches. Multiple agents can concurrently execute on a JavaSeal kernel. Agents are written in a restricted version of Java; they are forbidden from using several primitives and library methods. Security in JavaSeal is in fact enforced solely using language mechanisms. This security model is derived from Java's security model though had to overcome several weaknesses of the latter.

Overview: This paper is structured as follows. Section 2 presents the security model of JavaSeal and Section 3 describes its main features. Section 4 discusses the limitations of the Java security architecture. Section 5 details the implementation of JavaSeal. Section 6 presents HyperNews, a medium-sized (30 000 line) mobile agent application for selling short-lived digital documents on the Internet that runs on JavaSeal. Section 7 compares JavaSeal with other leading agent platforms, and Section 8 concludes with prospects for future enhancements.

2 The Meaning of Security

Security is frequently mentioned in the agent literature, yet it is often difficult to know what security *guarantees* are furnished by a particular system. We differentiate between security measures against exogenous threats – attacks that occur from outside of the platform – and security measures against endogenus threats for policing the execution of a single platform. Typically exogenous threats are addressed with cryptography and digital signatures [23, 18] which protect the contents of information while on the network and authenticate users. In this paper, we focus on security within a single platform. Section 2.1 reviews the security threats that are relevant in an agent system. Section 2.2 introduces some concepts. Finally Section 2.3 enumerates the security guarantees provided in JavaSeal.

2.1 Threat Model

A mobile agent system differs from a traditional computer system in that it allows *untrusted* agent programs to execute locally, use local resources and services, and to interact with other co-located agent programs. The threats, on the other hand, are not different from any computer system:

- **Unauthorized disclosure**: Data is read without proper authorization.
- **Unauthorized modification**: Data is subvertly modified or destroyed.
- **Denial of service**: Inordinate consumption of shared resources, preventing other programs from progressing.
- **Trojan horses**: Malicious code is mistakenly executed under the authority of a trusted user.

Each of the above attacks can be mounted by an agent against the agent platform or other co-located agents, in which case we call the agent a *malicious agent*, or by the platform against the agents it hosts, in which case we call the platform a *malicious host*.

When a mobile agent arrives, the platform typically must:

1. Verify that the agent came from the site that it claims to have come from.
2. Ensure that the agent has not been tampered with from the time it was sent.
3. Verify that the agent program is well formed, and that it possesses the necessary credentials to execute on the platform.
4. Grant the agent access to local resources and services.
5. Grant local resources and services access to the agent.
6. Allow execution while enforcing the local security policy. This policy assures that the agent and services can only access one another according to the access rights granted to each.

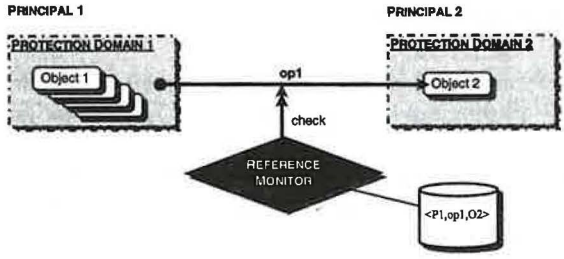

Figure 1: A system's security architecture. The reference monitor intercepts each access to an object from a remote domain, and queries the security policy.

The security architecture of an agent platform must cover all of these aspects. Our kernel approach is to focus on the points 4, 5 and 6 as they are essential for providing execution guarantees. Points 1, 2 and 3 counter exogenous threats and can be implemented by user-level services in JavaSeal.

2.2 Security Terminology

Before proceeding, we review essential security terminology. *Principals* are the entities of a system whose actions must be controlled. Principals typically represent users, though can also correspond to sites or services. Principals consume *resources* and invoke *operations* on *objects*. It is the role of the *security policy* to determine if a principal may consume a resource or invoke an operation on an object. A *protection domain* is a context in which a principal executes. It contains objects "owned" by that principal and for which the security policy does not need to be checked. Only operations that cross domains need be *mediated* by the security policy. Figure 1 illustrates these concepts. The term *reference monitor* is used for that component of a system that verifies the legality of each operation by consulting the security policy [11]. A reference monitor must satisfy two properties: total mediation — it intercepts all operations, and encapsulation — it is protected from tampering.

A real system contains a variety of channels over which protection domains can exchange information [24]. *Legitimate channels* are mechanisms included in a system precisely as a means of communication, *e.g.*, sockets, object references. Access control mechanisms regulate the use of legitimate channels. *Storage channels* are elements of the environment that can be read or written by several programs and which can therefore be used to exchange information between these programs. Examples of such channels include the shared buffers and kernel variables. The last category is that of *covert channels* which are means of communication that abuse some characteristic of the system to exchange data among programs. For instance, a value can be communicated by modulating some visible system characteristic such as the disk access rate. Covert channels are hard to block, and many security architectures are satisfied if the bandwidth of covert communication is sufficiently low. Of course, if the secret is a password, even a low-bandwidth channel is unacceptable.

2.3 JavaSeal Security

The goal of JavaSeal is to ensure that each agent executes in a protection domain of its own. All actions that affect other protection domains – either other agents or the kernel itself – must be controlled by the *agent reference monitor* (ARM). The JavaSeal kernel is an implementation of an ARM, and thus must be encapsulated from attacks by agents. The ARM verifies the legality of the following operations with respect to the security policy in place:

1. Creation of a protection domain.
2. Creation of a thread.
3. Communication across domain boundaries.
4. Loading of code into a protection domain.
5. Termination of a protection domain.

These operations represent all of the cross-domain operations allowed in JavaSeal. Controlling these operations is needed to satisfy the first requirement of a reference monitor (total mediation). The second requirement (encapsulation) is obtained through an assortment of programming language protection mechanisms. In JavaSeal, a protection domain is represented by the seal abstraction.

Aspects (1), (2) and (5) are hardwired into the JavaSeal kernel; each protection domain has a direct parent (see Section 3) that creates the domain and only this may terminate the child domain. Further, threads may only be created within one protection domain and cannot cross protection domain boundaries. A JavaSeal kernel has a configuration that describes which library classes may be used by services. In effect, each seal has its *directives*, which is a list of classes that the seal is allowed to load (4). Finally, inter-protection domain communications (3) can be controlled by user-defined supervisors (in our hierarchical model: parent agents). It is possible for a parent seal to allow or disallow its children access to individual services, and even to control how many times an agent may invoke an interface.

We now describe informally three security properties that an implementation of JavaSeal must have. Note that these properties must hold for all programs and all services.

- **Confinement:** Intuitively, confinement means that if the policy specifies that an agent does not have any open communication channels with other parts of the system, then that no matter what this agent does, its actions cannot affect any other part of the system. In essence, a confined agent is running behind a firewall isolated from the rest of the system. A formal definition of this concept is given in [34].
- **Mediation:** Mediation means that it is possible to interpose security code between an untrusted agent and any service available in the environment. Mediation is one step up from confinement. While confinement simply says that the ARM can close all channels, mediation means that it is possible to intercept every message going in and out of an agent.

- **Faithfulness:** This means that code executed in a protection domain under the authority of a principal actually belongs to that protection domain. This implies that JavaSeal prevents agents from somehow tricking other agents into executing foreign (Trojan Horse) code.

The guarantees are enforced entirely by means of Java's programming language based protection mechanisms. The interesting point is that the default security model of Java is not sufficient to enforce any of them, as we discuss in Section 4. JavaSeal security addresses standard and storage channels; covert channels are not specifically dealt with.

3 Seals – A Basis for Agents

Agents are autonomous programs that can move around a network while they execute. In JavaSeal, they are represented by software abstractions called *seals* which are hierarchically-structured encapsulated computations. Mobility is implemented by capturing the execution state of a seal and shipping it to another JavaSeal kernel.

We refer to a JavaSeal kernel instance as a *root seal*, one of which runs on each network node. An agent system is thus a group of root seals connected by a communication infrastructure. We now present a high-level overview of the JavaSeal system and discuss how seals can be used to structure mobile agent applications. Section 5 describes the actual implementation.

3.1 Seal Hierarchies

A seal is a self-contained program with its own data, code and execution threads, and which implements an agent protection domain. A seal may also contain a number of nested seals, called direct children. At the same time, every seal is enclosed within some other seal (or root seal) referred to as its direct parent. The set of children and parents of a seal refer to the transitive closure of direct parents and sets of direct children respectively.

The key feature of seals is the strict encapsulation that is enforced by the kernel at seal boundaries. Sharing of objects, of resources or of memory locations between seals is disallowed. Instead, every object and thread in a seal program belongs to a single seal. This clean separation makes accounting of resource usage easier and helps to enforce systematic security policies.

Seals communicate solely through messages (see 3.2). Channel communication is one-to-one and hierarchical: A seal can only communicate with its direct parent and children. Messages to distant seals, such as service requests, are encoded as a sequence of neighborly message exchanges. Thus if the service provider is at the root of the hierarchy, every seal between the client and the root must have a policy that allows this type of requests. This ensures that the services that an agent may use are either services that its direct parent implements or services available in a higher-level seal and to which the parents grants access.

Only the key services (seal creation and destruction, communication, memory management, scheduling and state capture) are under the control of the root seal. Services such as migration,

network access and even user interface are implemented at user-level by *service seals*. Service seals are special in that they are not mobile. They can only be loaded directly from disk by the root seal. Service seals have fewer security constraints imposed on them for this reason; for instance, they may use a larger number of library classes compared with standard mobile user seals. A similar dichotomy between untrusted mobile components and trusted local services is found in PLAN [20] and Mole [4].

Thus a seal controls its children in two ways. First, it is able to stop and start its children seals. Second, it intercepts messages sent from its children to other seals in its environment, and imposes security constraints on these messages. In contrast, a seal is not able to peek and poke the internals of any of its children seals, or of any other seal.

The design has been inspired by the Fluke micro-kernel [14] and work on interposition in operating systems [12, 15, 16]. We have not addressed interposition of low-level resources such as memory and the scheduler as this requires modifications to the virtual machine [3].

Two types of agents: In JavaSeal there are two categories of agents. The leaves of the seal hierarchy, which are called **complets**, are "traditional" mobile agents, while intermediate levels, called **envlets**, are mobile environments. The role of envlets is to interpose between requests of a complet (or nested envlet) and its environment. Thus for instance, they can play the role of adaptors when the services on the current platform do not match an agent's expectation (or act as the facilitators of [21]). Envlets can also play the role of a security policy. Figure 2 shows a JavaSeal platform running some complet named CompA. The requests that the complet is allowed to make to services such as the NetMgr seal are filtered by the Sandbox envlet. Depending on the current policy, the Sandbox seal may choose to disallow all network communication or only communication to a restricted set of sites. Service seals are typically structured as complets, but for more complex services nothing precludes using envlets.

Envlets are mobile just as any other seal. They can, for example, be used to make mobility

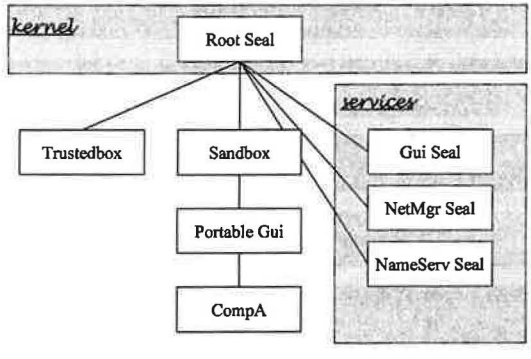

Figure 2: A seal hierarchy. The root seal runs three service complets and two envlets. Sandbox hosts untrusted agents while Trustedbox hosts friendly ones.

somewhat transparent. In Figure 2 the user interface is maintained by a local service seal. This means that when CompA moves its binding with its user interface are torn down. The Portable Gui envlet wrapped around CompA interposes on UI requests to keep track of the state of the user interface and rebuilds it after each move. In this scenario the envlet moves with its complet. The HyperNews application presents good examples of both types of agents (see Section 6).

From a security standpoint, malicious envlets are similar to malicious hosts in that they can control all communications going in and out of a subseal as well as stop a subseal. Thus even on a trusted machine and a trusted JavaSeal kernel there may be a malicious host problem. The difference is that JavaSeal does not allow an envlet to peek and poke in its children's memory, nor learn any secrets that they are not ready to divulge on their channel interface.

3.2 Communication

Synchronous message passing via named channels is the only inter-agent communication mechanism of the JavaSeal kernel. Channels are used for communication between neighbor seals: a parent may send a message to one of its children or a child may send a message to its parent. Channels are named; thus it is possible to have different channels for different purposes. Furthermore, the usage of channels is regulated by a separate access control mechanism called portals.

Channels are synchronous. The sender blocks until the receiver accepts a message. In order to have multiple outstanding requests a seal must create multiple threads. Values exchanged over channels are transmitted by copy (in, so-called, capsule objects described in Section 5) to avoid introducing sharing and thus avoids covert communication channels. Further security considerations are detailed in Section 5 when the implementation of JavaSeal is discussed.

The primitives for channel based communication are `send`, `receive` and `open`, to, respectively, send a message on a channel, receive a message on a channel and open a portal. Creation and destruction of channels is implicit. As an example consider seal `CompA` sending a message to its parent on a channel named `netreq`:

```
send( netreq , parent , message ) ;
```

The sender blocks until the parent accepts the message, which is written:

```
receive( netreq , self, val ) ;
```

Object `val` is bound to a copy of `message`. But, for the communication to fire the parent must have first opened a portal allowing `CompA` to use `netreq`:

```
Portal.open( netreq , CompA , 1) ;
```

This allows one use of `netreq`. Portals can be opened for any number of uses (including unbounded). Separating portals from communication allows seal designers to localize the security

code in an access control module independent from the main logic, and thus eases the task of verifying security properties.

The choice of synchronous communication is somewhat controversial as most other systems offer asynchronous communication mechanisms [8, 2, 25]. The advantages of synchronous communication are that (1) messages from the same thread are causally ordered, (2) acknowledgments of message reception are not needed, and (3) the number of outstanding request is bounded by the number of threads in a seal. This last property makes it easier to prevent denial of service attacks that flood a server with requests. It is not possible to flood an agent with requests and since JavaSeal limits the number of threads in any given seal (in practice the number of threads in a seal is bound by a kernel imposed limit), and there is an upper bound on the number of outstanding requests.

3.3 State Capture

The state capture mechanism of JavaSeal creates a machine independent portable representation of a seal. The procedure recursively traverses the seal hierarchy rooted at the target seal, stops the threads in each seal and pickles the data and code of each one into a *seal archive format* (SAF) object. This format is suitable for storage on disk, or network transfer. The latter is used to implement mobility. An archive can be used as a basis to create a seal. The creation procedure first verifies the validity of the archive with extended bytecode verification (see Section 5) and then unpickles the topmost seal in the archive. It is then up to that seal to decide if its children should be awakened.

With the exception of kernel code, all the code used by a seal is included in its SAF. This means that our archives are potentially quite large, definitely larger than those of agent systems that do loading on demand. Our motivations for this choice are the following: (1) We cannot rely on the connectivity of an agent's source. If an agent's source site is a portable PC, then the site might not be connected to the Internet when an agent begins to execute at its destination and discovers that a class that it needs is not present; (2) Versioning support in Java is weak, we cannot guarantee faithfulness if, for example, two classes are released with the same interface and version number (a common problem). The disadvantage is that the size of SAF files is larger and thus their transfer costs more. JavaSeal uses a custom code compressor called Jazz [6] (part of the JavaSeal project) which is able to reduce Java bytecode files to 24% of their original size. Further reduction can be achieved by not transmitting code if it is certain that the receiving site already has that class. This can be integrated in JavaSeal as a user-level service in NetMgr.

The interface for archiving and loading seals consists of two operations: `wrap` which takes the name of a subseal and returns an archive and `unwrap` which takes an archive object and a subseal name and creates a new subseal.

```
safObj = wrap( subsealName ) ;

unwrap( safObj , subsealName ) ;
```

These operations are consistent with the hierarchical control of the seal model. The kernel ensures that only the direct parent of a seal can wrap it. Similarly, new seals are always rooted in the currently executing seal. A single thread is started by default in each unwrapped seal.

4 Limitations of Java Security Model

The JavaSeal platform has strong security requirements since its goal is to enforce a strong separation between seals. Since JavaSeal is designed to execute Java agents, we exploit Java language verification mechanisms to enforce security. We have considered the Java security architecture [37, 17] for JavaSeal, but after a detailed investigation, we concluded that it is not strong enough to guarantee the security properties of confinement, mediation and faithfulness that we mentioned. Furthermore, we have identified some serious denial of service attacks that can jeopardize the entire JavaSeal platform.

Java treats classes as protection domains and uses `SecurityManager` objects to ensure that a class from one domain can only call methods that it has been authorized to invoke. `Access modifiers` are a second form of protection. They are used to protect sensitive fields of the JVM. For instance, a user cannot have a system class replaced by subtyping it since these classes are declared with (the access modifier) `final`. Finally, bytecode verification guarantees that programs are well formed and will not break language safety. In addition to this, applet programs from different origins are separated from one another by a form of namespace protection domains. That is, all classes of each applet are copied and considered to have a distinct type from other copies. This ensures that applets do not acquire references to objects belonging to other applets, and so any attempt to reference an object of another applet is signaled as a type violation. However, this also means that applets are not allowed to communicate.

The main problem for enforcing security with this model comes from the choice of class-based protection domains. A conservative estimate places the number of cross-domain operations per second at 30 000 [37]. This means that it is impossible on efficiency grounds to check all operations, thus there can be no real reference monitor. Class domains do not facilitate resource accounting: though one can control *what code* is using memory and CPU resources, one cannot control *who* is using this code. The primary goal of this security architecture is to protect the virtual machine from the programs running on top of it.

There are ways to circumvent Java security. We identified a few in earlier work [36], here we focus on those related to three JavaSeal security properties.

Confinement: The difficulty in obtaining confinement is that the JVM is one very large shared data structure. There are numerous covert and storage channels for domains to communicate thanks to shared library classes. Java has `static` variables that can be used to implement global variables. Many of these variables are also `public` meaning that they are visible to all clients. Every object in Java has an associated lock. When two domains can see the same lock they have a covert way to exchange information. Similarly the fields like the `threadCount` of class `Thread` can be used as a low bandwidth storage channel. In Java,

a storage channel is opened if there is a way through some sequence of calls to cause a static variable to be modified and if it possible to read back that value. Threads also pose problems as they can be stopped abruptly by their creator. For instance, if an agent creates a thread and calls a method in the interface of another agent, then stopping that thread while it is executing in the second agent could leave the victim in an inconsistent state.

Mediation: Even if confinement holds, as soon as any inter-agent communication is allowed, unchecked channels can arise through dynamic aliasing. A good example is the security breach found in the JDK 1.1.1 implementation of digital signatures which allowed untrusted code to acquire extended access rights [30]. This was caused by mistakenly returning a reference to the system's key ring which allowed any applet to increase its own access rights by adding signers to the key ring. As we observe in [5] there is no systematic way to ensure that such channels do not exist.

Faithfulness: Java version control does not guarantee faithfulness because version numbers are not guaranteed to be unique. Further, subtyping can be used to mount code injection attacks. In this attack, instead of sending an object of an expected type, the attacker sends an instance of a subtype; this is allowed by the type system, and when the victim uses the object it is the code of the attacker that is executed. For instance, one could define a subclass of some Java collection type with an iterator that does not return, so that when a thread tries to traverse the iterator its gets stuck and loops forever.

A further issue to consider is denial of service attacks. Java does not have a resource management interface that could allow us to account for the usage of resources such as CPU and memory by a program. Simply creating an unbounded number of new threads can cause a denial of service attack. Another problem is linked with finalization, if an object has a finalizer method that contains an infinite loop, then when the garbage collector will be stuck and most JVM implementations crash in less than a minute. Finalizer also make domain termination difficult to implement. The finalization code may be executed at any time and revive a killed application.

Under these circumstances, in a system the size and complexity of the Java virtual machine security breaches inevitably occur and *proving* that an application built over the JVM is secure is bound to be difficult. On a more fundamental level, the problem with JDK is that the shared kernel interface (comprised of the JDK core classes) is too big to reason with, and there are no checks of the communication effected between the kernel and the protection domains. The JVM model is adequate for protecting a single user from the dangers of executable content downloaded from remote Internet hosts but fails to provide a secure basis for building complex applications composed of untrusted or fallible components such as agent applications.

5 JavaSeal Implementation

JavaSeal consist of 20 000 lines of pure Java code. The systems runs on JDK1.2, though can run on 1.1 with only slight modification. The body of the JavaSeal system is structured as several Java packages (named `seal.sys`, `seal.lib`, `seal.srv` and `seal.usr`) which almost completely replace the standard JDK packages. User code can only be added to the package `seal.usr`. This restriction is enforced by the loader.

In this section, we first look at the implementation of JavaSeal concepts, and then look at the security implementation issues.

5.1 JavaSeal Kernel classes

There are only few core classes in the kernel that are visible at user-level. The class `Seal` is the base class of all user defined agents. Channels are instances of the `Channel` class and they exchange capsules. Class `Portal` is used to control access to channels. Class `Strand` is a restricted equivalent of `Thread`, the name has been changed to avoid confusion. Finally, the `SealLoader` class implements seal loading and verification. Some selected interfaces are shown in Figure 3.

5.1.1 Seals

A seal consists of classes, objects and threads. The classes of a seal are loaded by a dedicated `SealLoader` and are not shared with any other seal. All objects that are reachable from a seal are owned by the seal. Every seal has a `run` method which is called to start execution.

When a seal is created it is assigned its own class loader. The seal's class is linked into the JVM using this loader. Each class subsequently referenced by the seal is thus loaded by the same `SealLoader`, and each seal has its own copy of all of its classes. There is a small number of exceptions: classes like `Object` must be shared. The set of classes loaded by a loader, along with all of the instances of these classes form a protection domain. A type cast error is generated if an object of one domain attempts to directly reference an object of another domain.

A `SealLoader` has two possibilities for finding classes. System classes are found in pre-defined locations on disk. User-defined classes are stored in a seal's *archive*. The archive is used to enforce faithfulness: a seal always uses the classes with which it was defined. It does not rely on any other seal, or its environment, to furnish it with a (perhaps infected) version of its classes. Furthermore, seal archive files are immutable. That is, a seal may not add new classes to its SAF during execution. The advantage of immutability is that the SAF may be digitally signed and any attempt to tamper with the code can thus be detected.

A seal creates a child seal through a kernel operation — a class archive is created and a new loader is allocated for the child. The parent can subsequently wrap the child seal. Wrapping a seal entails stopping its threads, serializing its data into a byte array, and then packing this byte array and the class archive into a SAF. This SAF can be used to re-instantiate the seal, or

```
public abstract class Seal implements Runnable, Serializable {
     public static Seal currentSeal()
     public static void dispose(Name subseal)
     public static void rename(Name subseal, Name subseal)
     public static SAF wrap(Name subseal)
     public void run();
...
}

public final class Channel {
     private Channel(Name me);
     public static Capsule receive(Name channel, Name seal);
     public static void send(Name channel, Name seal, Capsule caps);
}

public final class Portal {
     private Portal();
     public static int status(Name channel, Name seal);
     public static int open(Name channel, Name seal);
     public static int close(Name channel, Name seal);
}

public class Capsule implements Serializable{
     public Capsule(Object obj);
     public Object open();
}

public class Strand {
     private Strand();
     public static Strand create(Runnable target);
     public static Strand currentStrand();
     public void start();
     public void stop();
     ...
}
```

Figure 3: The JavaSeal kernel classes

alternatively it can be sent over a channel and then re-instantiated within another seal. This is how agent migration is done in JavaSeal.

5.1.2 Strands

The threads that execute inside of seals are called Strands. A strand is bound to the seal in which it is created. It cannot leave the seal or cannot be used to gain information about strands in other seals. This is a crucial different between strands and Java threads. Each seal object

has a `run()` method; when a seal is created or unwrapped, a strand is automatically created to execute this method.

In the implementation, there is a mapping between threads and strands. An initial strand is explicitly created when a seal is started, and to handle parallelism, *daemon* strands should be started to service external calls. In practice, the daemon has a limit on the number of strands, and manages strands by reusing passive strands when possible.

5.1.3 Channels

The channel class has methods `send` and `receive` to transfer an object of type `Capsule` from a sender seal to a receiver. Both operations are blocking; the strands issuing them will be blocked until the communication is allowed to fire.

```
String x = new String("req");
Chan ch = new Chan(x);
Capsule cp = new Capsule(str);
ch.send(x, cp, Seal.getParent());

String x = new String("req");
Chan ch = new Chan(x);
Portal.open(x, Name("Agent1"), 1);
ch.receive(x, Name(''Agent1''), cp);
String s = (String) cp.open();
```

The first code fragment tries to send a string object `str` along channel `x`, the second code fragment waits on channel `x` and unpacks the value received into a string. A portal acts as a control on a communication channel, and must be explicitly opened by the owning seal for any communication to take place. In JavaSeal, this is represented by the `Portal` class. Its `open` method opens a portal for a channel and seal pair, enabling the named seal to communicate with the owning seal over the channel. The `close` method has the reverse effect.

5.1.4 Capsules

Capsules transfer data across channels. A capsule contains a copy of a group of objects. The copy is done in kernel mode and ensures that the capsule does not share references with the sender. Capsules are currently implemented with Java serialization. Opening a capsule releases its contents into the local environment. A capsule is opened successfully only if the `SealLoader` of the receiver is able to resolve all the classes required by the objects in the capsule.

A capsule is created by specifying an object, the *root*, which will be copied into the capsule. The copy is a *deep copy*, that is all the objects in the transitive closure of the root object are copied as well. A complete capsule thus contains a disjoint copy of a portion of the object graph, *i.e.* there is no sharing or aliasing between objects in the application and the objects in a capsule.

A capsule may only be opened once. Opening a capsule releases its content. This process requires finding matching classes definitions. The important point to note is that the `ClassLoader` must be able to find all classes required by the capsule contents in its environment. The classes found might have different versions, currently we rely on Java type compatibility rules to verify the validity of a capsule. The `open` operation fails if some of the classes required by the capsule are not found in the local environment.

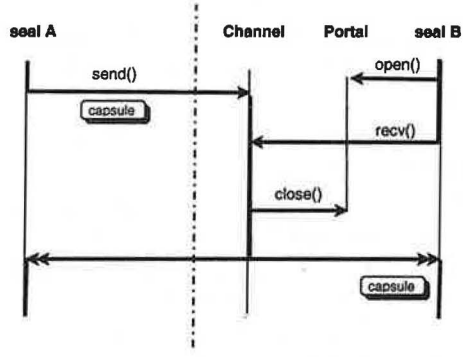

Figure 4: Channel based communication offered by the MPI.

Communication between a sender seal A and receiver seal B is illustrated in Figure 4. We assume that A and B agree on a channel name, initial agreement is achieved using a set of "standard" channel names. A strand executing within A invokes the `send` primitive on the channel with a capsule as argument. A's strand blocks until the communication completes. Seal B must at some point (possibly after the call to `send`) open a portal for A on the agreed channel. Then B must invoke a `receive` operation on the channel. This primitive blocks until a matching offer appears. In this case, the communication proceeds: the portal is first closed, the capsule is copied into B, and then both strands are notified that the communication was successful.

5.2 JavaSeal Security

Seals are protection domains in JavaSeal and language mechanisms enforce their security. Confinement is achieved by a combination of two main means. First, each seal is assigned its own loader, meaning that there is no sharing (nor storage channels) between seals. Second, capsules are designed to prevent dynamic aliases from occurring by using a deep copy mechanism to copy parameters.

The isolation imposed by seal loaders may appear a bit drastic as we effectively separate each seal from most of the JDK. One may argue that it may be possible to prove the JDK classes free of storage channels and then it would be safe to share them. The problem is that we cannot be sure in which environment a JavaSeal platform will be used. Depending on which classes

are loaded on the JVM, or which versions of the classes, storage channels may exist. It takes only one class to break the entire security.

Seal loaders also perform extended bytecode verification to impose a fairly stringent restriction on finalizers: they are forbidden from containing loops or calls to methods, as the latter could be an invocation of a non-terminating method. A more sophisticated analysis could be used to allow more behavior in finalizers but we have not yet encountered practical cases where this is needed in applications written for JavaSeal.

Channels enforces another aspect of domain isolation through `Strands`. Strands only exist in a single seal. Thus we prevent attacks that rely on killing a thread while it executes within some other protection domain.

Mediation is obtained by nesting a target seal in another seal. the second seal being responsible for interposing on sensitive channels.

Domain termination is achieved by simply stopping all of the strands of a seal and setting all domain-specific kernel pointers to null. Memory will eventually be reclaimed by the garbage collector. The resources used by a seal are relatively easily accounted for. They include classes loaded by the seal's loader and objects reachable from the seal class. Thread objects are accountable through the strands. In the current version of JavaSeal, precise control over memory and CPU is not provided. It would be fairly straightforward to approximate resource usage by instrumentation of the bytecode, but a cleaner approach would be to extend the JVM interface with hooks for that purpose [10].

As mentioned, faithfulness is enforced by the seal loader. When a seal is created all of its classes are extracted from its archive. In addition, when messages are exchanged, the seal loader checks that no opened capsule tries to inject new classes.

5.3 Kernel Security

An important property of a reference monitor is that it must be encapsulated. We cannot enforce strong isolation for the kernel classes since some key JDK classes have to be shared. This is a design choice of the Sun JVM implementation. The JavaSeal kernel classes are also shared. This sharing is a security worry since it can be the source of storage channels. The basic idea is to have a **well-defined** and **small** interface, and to use a combination of access modifiers and type abstraction [26] to ensure that this interface is correctly used. The kernel interface is restricted to 8 JavaSeal kernel types and 25 standard Java types most of them exceptions (this includes classes `Object`, `String` and `StringBuffer`, as well as interfaces `java.io.Serializable` and `java.lang.Runnable`). All arguments and return values of these types are also part of the kernel. These classes have no static variables and instances have no accessible fields that are not instances of kernel types.

Another reason for insisting on a small interface is to prohibit security leaks from dynamic typing. In Java, an object can only reference an object of a class loaded by the same loader, or by the system loader[1]. However, dynamic typing can lead to a bypassing of this rule. Consider

[1] This explains how the core JDK classes can be shared. These classes are loaded with the system loader. Since all domains need to have access to these classes, the typing rule is "bent" to accommodate this.

a class C which is loaded by two loaders; we denote the resulting class instances C_1 and C_2. Suppose that the class C implements the Java interface I and that this interface is loaded by the system loader. If some class has a variable of type I then objects of this class may reference instances of classes C_1 and C_2. This is because I is visible in all domains, and dynamic typing permits an object of a subclass (*e.g.*, C_1 or C_2) to replace an instance of I. This clearly constradicts the name space approach of Java.[2]

The only point where a seal is allowed to give arbitrary objects to the kernel is as an argument to a capsule of a message that the seal exchanges with the root seal. However, the capsule is opened in the receiving environment and can at no time reference a kernel object.

The final part of the kernel security is obtained by *selective access modifiers* which are enforced by a form of extended bytecode verification. We extend the standard Java access modifiers with a more fine-grained version enforced at load time by the `SealLoader`. We introduce directives that specify selective access modifiers. An example directive sequence is the following:

```
see java.lang.Object;
final seal.sys.Capsule;
private java.lang.Object.getClass();
```

The first directive specifies that class `Object` is visible. In other words, the running seal object may link against this (shared) class. The second specifies that a class is to be treated as final; thus no subclasses are allowed in the seal. The last directive specifies that an attribute of a class cannot be used within a seal. These modifiers are read in by the `SealLoader` and all classes loaded are checked to conform to these restrictions. The directives are enough to ensure that the only types exchanged using shared classes are those permitted in the kernel interface.

6 HyperNews: Selling News on the Web

HyperNews is a system for the electronic distribution of news articles in which a client can only read the contents of an article that he has paid for [29]. This section describes the implementation of HyperNews over JavaSeal.

6.1 The HyperNews Business Model

The goal of HyperNews is to support the electronic sale of news articles. The actors are the consumers, the press agencies producing articles, and the credit institutions (CIs) that manage payments. A typical HyperNews transaction is illustrated in Figure 5. A consumer requests a set of sports related articles from a news agency. These articles are downloaded to the consumer's site. To read an article the consumer must have paid for it. For this reason an article's contents are encrypted with a symmetric key k. The article contains k encrypted with the public key of

[2] This feature is nonetheless exploited in the JavaSeal implementation to implement hierarchial operations such as wrapping and seal creation. It is nevertheless hidden from user seals.

the CI. If the consumer is reading the article for the first time, the CI is contacted and the article price is debited from the consumer's account. The CI extracts the value of k for the consumer and sends it to the consumer with a receipt of payment. It is only with the key k that the article can be read. Subsequent uses of the article simply require presenting the receipt to the CI, after validation of which k is extracted and returned.

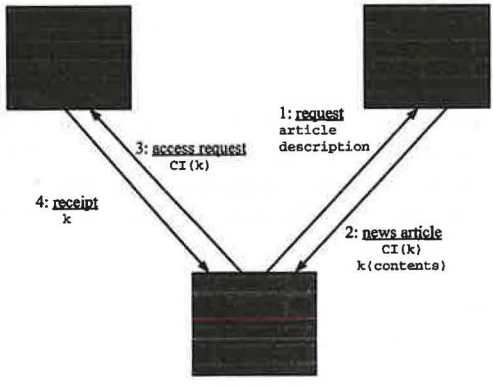

Figure 5: HyperNews allows customers to request news articles from news agencies. Payment is handled through trusted credit institutions. All exchanges between any two sites are encrypted by a session key negotiated between those sites.

HyperNews is designed as a large scale distributed application. Not only are there many consumers, but there are, of course, also multiple competing news agencies and credit institutions. A consumer is at liberty to buy articles from any news agency. There is also certain symmetry in the architecture. For instance, a client can act as a press agency. He can collect articles of some class, annotate them with his own comments, and later resell them; HyperNews nevertheless guarantees that whenever the original article is viewed, its rightful owner still gets paid.

6.2 The HyperNews Security Model

HyperNews is built with the goal of doing electronic commerce over the Internet. With respect to security, this implies having well-specified and maintainable trust relations and the use of encryption.

Regarding trust, a consumer trusts his credit institution to store his credit account. Similarly, news agencies trust the CIs with the keys k to their articles which they see when handling payments. At the same time, CIs are also trusted to archive public keys.

The detail of the payment is as follows. Each article's contents is encrypted with a symmetric key k that is chosen by the news agency. The key k is then encrypted with the public key of the CI, yielding *CI(k)*. The encrypted contents and *CI(k)* are packed into the article agent which

is downloaded to the consumer. At the consumer's site the local HyperNews platform manages payment requests. Whenever a user asks to read an article, HyperNews sends a request to the CI and this request contains *CI(k)*. If the customer has sufficient funds to pay for the article, the CI debits the consumer's account and then forwards a receipt of payment and k back to the consumer. The HyperNews system can now decrypt the article contents. Immediately following the decryption, the key k is discarded by the runtime in order to reduce the risk that an attack on the consumer platform can lead to k being revealed. The next time the user wishes to read the same article, the CI must again be contacted. This time, the user sends a copy of his receipt, which the CI validates, and replies with k.

Security of the article keys relies on the integrity of the HyperNews platform on the client's site. Clearly, a hacker may tamper with the system and steal keys. But this requires some skills, and a key only unlocks a single article. Further keys are obtained only after the document has been paid for. In this way, the worst that an attacker can do is to distribute the article contents free of charge. For commercialization of short-lived, low-value, documents such as news articles, this is not likely to be a major problem.

HyperNews uses agents to customize the treatment and user interface of different news sources. Thus, each provider is allowed to install a *news feed* agent at the customer. The news feed is responsible for verifying receipt of payment before access to the article, and for decrypting the contents and then throwing the key k away. Articles also may contain code for interacting with the user. The remaining security measures in HyperNews are directed to guaranteeing that different news agencies are not able to disrupt each other, at preventing malicious agents damaging the consumer's system, and at preventing denial of service attacks.

6.3 Implementing HyperNews

The main attraction of agents for implementing HyperNews is that they allow different news providers to customize the application installed on the customer's station with value-added services on a per-document basis. The advantage over a client-server solution is that no connectivity with the news provider is needed.

The HyperNews application is built as a collection of cooperating seals. A *HyperNews platform* is a JavaSeal kernel loaded with the HyperNews seals. The entire application has been designed using agent technology. Everything from session key negotiation to news articles is done with agents.

6.4 Architecture

The overall structure of a running HyperNews platform consists of a number of NewsFeed agents and a large number of article agents (see Figure 6). The NewsFeed agents are envlets that manage all the data and services common to one news provider. For example, a NewsFeed may keep track of the news classification of its provider, it may contain code for filtering incoming articles according to user-defined criteria, as well as custom code for decryption or decompression. Articles are complets which execute within their provider's envlet. They also

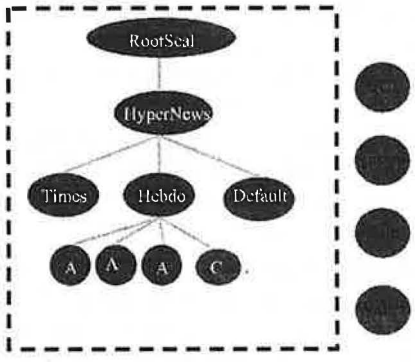

Figure 6: The HyperNews application. NewsFeeds for the Times, Hebdo and a default news feed are shown. These are envlets with couriers and articles as complets. The dotted line represents the trust boundary, that isolates mobile seals and the network area from trusted immobile services.

may contain code. Articles can have special behavior with respect to payment, display or consumer interaction.

Every HyperNews platform has a reception area implemented by a complet. The reception area is a service seal with network access. Its role is to receive incoming seals, authenticate them, and decide if they should be allowed to execute. The incoming seals can be either new Article agents or Courier agents (A and C in Figure 6 respectively). An Article agent contains articles, a Courier agent carries receipts or article keys. The former are expected only if the user signed up for news from that particular press agency, the latter should belong to one of the existing feeds. A NewsFeed is started on its own in the Sandbox (another envlet). Couriers are forwarded to their feed and will be allowed to execute within that envlet.

The services include a HTTP deamon for the Netscape browser used to visualize article contents and a Swing-based *GUI* implements the HyperNews control panel. A *storage* agent is used for storing serialized article agents; as soon as the environment detects that the platform is becoming too heavily loaded, articles are selected for swapping to disk by the file storage agent. An *electronic commerce* agent manages a purse GUI containing the consumer's credit, and decides to ask for more credit when needed. Finally, a *utility* agent implements cryptographic functions and manages the environment variables.

Starting JavaSeal creates a RootSeal. This creates the main application seal of the application, whose name must have been passed as parameter in the command line. This seal proceeds to create new seals.

After the RootSeal has instantiated itself, it starts the NetSeal that is responsible for communicating between sites. The kernel actually treats the NetSeal as being the parent of RootSeal. There are two reasons for this. First, the NetSeal represents the network, which from the hi-

erarchy point of view, encapsulates all platforms of the mobile agent network as children [34]. Second, the elements received from the network must be isolated from the system services and other agents; for this reason this component is inside of the JavaSeal protection barrier. NetSeal is the only service that executes within JavaSeal; all other services exist outside of JavaSeal though execute within the same JVM.

After creating the NetSeal, RootSeal creates a Bridge object that is used to forward messages between seals and the services. Services like GUI, FileStorage, etc. are represented as static classes; these classes are instantiated in the `main()` of RootSeal, and use the basic system loader. Services are represented in this way so that sharing with seals is kept to a minimum, since the seals occupy different loader spaces – and protection domains.

6.5 Security

One important point related to security is that a HyperNews platform is a long-lived application. The state capture mechanism is used to make the platform persistent. The implication is that malicious agents should not be allowed to crash the system, and also that shutting down the JVM is not an appropriate response to a denial of service attack. JavaSeal tries to control resources so as to reduce the potential for denial of service attacks, but there is plenty more work to do in that field.

One problem that is solved by JavaSeal is the protection of NewsFeeds from one another. This is achieved by the isolation imposed by the seal model: all feeds are represented as child seals of the main application seal. All potential interactions are subject to the reference monitor and allowed only if there is a specific permission for two seals to communicate. By default, NewsFeeds are not allowed to communicate. Security on the articles is enforced by the NewsFeeds which decide whether the articles that they host may communicate (usually there is no need to). A further security property comes from the fact that the environment (root seal) is not able to peek and poke the messages exchanged between a NewsFeed and its Courier agents, represented as children seals. This is because communication is only possible between parent and children, and Courier agents are started as children of their provider's news feed envlet. In this way, the article key k is localized on the platform, and there is less risk of it escaping into the environment.

7 Related Work

Mobile Agents are a combination of active objects [2] and mobile objects [22]. Active objects are objects that possess their own thread of control and which execute independently of their creator. Mobile objects in Emerald could also be moved transparently between sites of a distributed system. The arrival of the Internet renewed interest in mobile objects, though mobility could no longer be done transparently: an agent had to be aware of where it was executing because resources and administration could differ between sites.

Among the first mobile agent systems were Telescript [38], TACOMA [32] and M0 [33]. The former two possess a coarse-grained notion of agent, the latter uses lightweight agents.

Telescript transported much information with its agents; the model became too complicated and eventually the project was stopped. M0, and other agent systems based on scripting languages such as Tcl/Tk and FACILE are lighter weight agents; ironically, their simplicity makes coding of envlets harder.

The arrival of Java brought a wave of Java-based agent systems. The reason for this is that use of Java is widespread, it has enough utility classes and possesses notions of security and mobility. Example systems include Mole [4], D'Agents (from Tcl/Tk), Voyager, Ajanta and Aglets. However, these systems do not provide a level of security based on strict separation between agents. since the kernel does not occupy a different domain the agent domains. The J-Kernel implements capabilities [19]. A protection domain in the J-Kernel is also a name space implemented using a class loader. Communication between domains is achieved by invoking a method on a capability object which acts as a mini-RMI stub. Parameters are deep-copied between domains and only capabilities and core classes are shared. In the J-Kernel service classes are shared between agents, thus lending themselves to covert channels. Further, J-Kernel does not possess the notion of hierarchy; this makes it difficult to implement envlets, as required by HyperNews and other applications.

The Sun's JVM (JDK1.2) includes many changes to the security model — including protection domains based on distinct class loader spaces. But as we argued here , distinct loader spaces do not constitute real protection domains unless a real attempt is made to isolate the variables shared between loaders — those variables whose classes are loaded by the system loader. Further, we argue that no real change to the JVM is needed to achieve this level of security, rather a fundamental redesign of the JDK.

Protection domains are also an operating system issue and many of the ideas here are influenced by such work. For instance, the hierarchial model is influenced by Fluke [14] and L3 [27], as well as by work on interposition [12, 15, 16].

8 Conclusion

This paper has described the JavaSeal platform. This is a secure kernel for mobile environments (envlets) and mobile objects (complets). JavaSeal is a kernel in that it offers minimal service funtionality. Since services differ between sites, one should be able to build different services on a kernel. JavaSeal is secure in that it isolates agents (or seals) from one another by exploiting the typing mechanism, and it extends the class loading verifier to ensure that seals do not use forbidden or untrusted classes.

The main lesson that we have learned from the JavaSeal implementation is that it is possible to implement a secure kernel based on Java. We qualify security in this case as strong separation between agents, and between agents and services. Of course, some covert channels may remain in the kernel though we believe these to be of insignificant bandwidth compared to the storage channels that can exist in the JDK service classes. Like others [1], we have also learned that full migration was not easy in the Sun JDK due to low-level implementation issues. Finally, we also noted that Java still has some efficiency problems, with respect to wrapping sizes and data transfer times. This motivated our work on compression. A more significant performance

problem is caused by the fact that messages are rerouted through common parents. Our current work includes investigation of a shared object concept: this is an object that can be directly shared between two domains without the security policy in place being violated.

Acknowledgments The authors thank Walter Binder, Manuel Oriol, and Karim Taha for their work on JavaSeal, and Jean-Henri Morin for using our system to implement HyperNews.

References

[1] Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Ressource-Aware Programs. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1997.

[2] G. Agha. *Actors – A model of concurrent computation in distributed systems*. The MIT Press, 1986.

[3] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

[4] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser. Mole - Concepts of a mobile agent system. *World Wide Web*, 1(3):123–137, 1998.

[5] B. Bokowski and J. Vitek. Confined Types. In *Proceedings 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, USA, November 1999.

[6] Q. Bradley, R. Horspool, and J. Vitek. JAZZ, compression of Java bytecode. In *CASCON'98*, 1998.

[7] C. Bryce, M. Oriol, and J. Vitek. A coordination model for agents based on secure spaces. In P. Ciancarini and A. Wolf, editors, *Proceedings of the 3rd Conference on Coordination Languages and Models*, volume 1594 of *LNCS*, pages 4–20. sv, 1999.

[8] 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.

[9] Carriero and Gelernter. Applications experience with Linda. *ACM Sympos. on Parallel Programming*, July 1985.

[10] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. *ACM SIGPLAN Notices*, 33(10):21–35, Oct. 1998.

[11] DOD. Tcsec: Trusted computer system evaluation criteria. Technical Report 5200.28-STD, U.S. Department of Defense, Dec. 1985.

[12] T. Fine and S. E. Minear. Assuring Distributed Trusted Mach. In IEEE, editor, *Proceedings of the 32nd IEEE Conference on Decision and Control, San Antonio, TX, USA, December 15–17, 1993*, pages 206–217, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[13] F. for Intelligent Physical Agents. FIPA 97 specification part 1: Agent management, Oct. 1998. Version 2.0.

[14] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 137–151, Berkeley, CA, USA, Oct. 1996. USENIX.

[15] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 39–52, Berkeley, USA, June 15–19 1998. USENIX Association.

[16] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. Interposition as an operating system extension mechanism. Technical Report CSD-96-920, University of California, Berkeley, Apr. 9, 1997.

[17] L. Gong. Java security architecture (JDK 1.2). Technical report, JavaSoft, July 1997. Revision 0.5.

[18] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, Jan. 1998.

[19] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. Technical Report 97-1660, Cornell University, Department of Computer Science, 1997.

[20] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998. Available at `www.cis.upenn.edu/~switchware/papers/plan.ps`.

[21] N. Jamali, P. Thati, and G. A. Agha. An Actor-based architecture for customizing and controlling agent ensembles. *IEEE Intelligent Systems*, 1998.

[22] E. Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, University of Washington, Computer Science Department, Dec. 1988.

[23] G. Karjoth, D. B. Lange, and M. Oshima. A security model for Aglets. *Lecture Notes in Computer Science*, 1419:188–??, 1998.

[24] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16, 1973.

[25] D. B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web Journal*, 1998.

[26] X. Leroy and F. Rouaix. Security properties of typed applets. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, San Diego, California, 19–21 Jan. 1998.

[27] J. Liedtke. Improving IPC by kernel design. In B. Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–188, New York, NY, USA, Dec. 1993. ACM Press.

[28] D. Milojicic, M. Breugst, I. Busse, and J. Campbell. MASIF: The OMG mobile agent system interoperability facility. *Lecture Notes in Computer Science*, 1477, 1998.

[29] J.-H. Morin and D. Konstantas. HyperNews: A MEDIA application for the commercialization of an electronic newspaper. In *Proceesings of SAC '98 - The 1998 ACM Symposium on Applied Computing*, Marriott Marquis, Atlanta, Georgia, U.S.A, Feb. 27 - Mar. 1 1998.

[30] Secure Internet Programming Group. http://www.cs.princeton.edu/sip/news/april29.html. 1997.

[31] P. Sewell and J. Vitek. Secure composition of insecure components. In *IEEE Computer Security Foundations Workshop (CSFW12)*, Mordano, Italy, June 1999.

[32] Tromsø University and Cornell University. TACOMA Project, http://www.cs.uit.no/DOS/Tacoma/.

[33] C. Tschudin. The messenger environment M0 – A condensed description. In *Mobile Object Systems: Towards the Programmable Internet*, pages 149–156. Springer-Verlag, Apr. 1997. Lecture Notes in Computer Science No. 1222.

[34] J. Vitek. *The Seal model of Mobile Computations*. PhD thesis, University of Geneva, 1999.

[35] J. Vitek and G. Castagna. Towards a Calculus of Secure Mobile Computations. In *Workshop on Internet Programming Languages*, Chicago, Ill., May 1998. reprinted in *Electronic Business Objects*, Ed. Tsichritzis, University of Geneva, 1998,.

[36] J. Vitek, M. Serrano, and D. Thanos. Security and Communication in Mobile Object Systems. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1997.

[37] D. Wallach, D. Balfanz, D. Dean, and E. Felton. Extensible Security Architectures for Java. In *Proceedings of the 16th Symposium on Operating System Principles*, 1997.

[38] J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.