The JavaSeal Mobile Agent Kernel

Ciarán Bryce
Centre Universitaire d'Informatique
University of Geneva
Switzerland
ciaran.bryce@cui.unige.ch

Jan Vitek
Department of Computer Sciences
Purdue University
West Lafayette, IN
jv@cs.purdue.edu

Abstract

JavaSeal is a secure mobile agent kernel that provides a small 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 that had to be overcome, and then present a medium-sized e-commerce application that runs over JavaSeal.

1 Introduction

Mobile agent systems come in all shapes and sizes. 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, 12], 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? How should one distinguish between mobile and immobile software components? Further, which services should be provided in the agent platform and which can be coded at user-level?
- **Security:** What execution guarantees can an agent platform provide? While, there is a wide consensus on the need for security, few systems provide clear statements of their meaning of security.

This paper reports on our experience in implementing and using a Java-based agent kernel. The JavaSeal system has been designed to support the 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 security constraints. JavaSeal provides three abstractions for supporting agents:

1. Hierarchically structured program units called *seals*.
2. Secure communication primitives.
3. A state capture mechanism and custom archive format.

This kernel approach is visible in the implementation of services, *e.g.*, the network interface or the graphical user interface are dynamically loaded user-level modules. The advantage of the minimal kernel model is that it is easier to reason about the properties of mobile programs. A related project is investigating formal proof techniques for agent systems, and has defined a formal semantics of JavaSeal as a process calculus and has been able to validate security properties [33, 34].

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 network. Migration implies that both data and code of the agent will be available to continue its computation on the new platform. We assume a medium-grained agent model in which every platform may host several hundred concurrently executing agents.

In JavaSeal, agents are organized in a hierarchy rooted at the kernel. This hierarchy is exploited to build composite agents out of other existing agents. One important use of aggregation is to move agents along with a portion of their environment. In JavaSeal, this is achieved by representing an environment as a seal, and the programs running in 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 favors portability over performance. While services can be shared without having to provoke context switches, the cost of communication is significant.

Agents are written in a restricted version of Java; they are forbidden from using several primitives and library meth-

ods for security reasons. Agents are allowed to interact, but these interactions are subject to a security policy. Security in JavaSeal is 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 that had to be addressed to implement JavaSeal. Section 5 details the implementation of JavaSeal. Section 6 presents HyperNews, a medium-sized mobile agent application for selling short-lived digital documents that runs on JavaSeal. Section 7 compares JavaSeal with other agent platforms, and Section 8 concludes with future work.

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 security measures against exogenous threats – attacks that occur from outside of the platform – from security measures against endogenous threats. The latter are used to police the execution of a single platform. Exogenous threats are addressed with mechanisms like cryptography and digital signatures [23, 18] which create secure network channels and authenticate message senders. 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 allows *untrusted* agent programs to execute locally, use local resources and services, and to interact with other co-located agents. The threats are not different from any other computer system:

- **Unauthorized disclosure**: An agent or service reads data without the proper authorization.
- **Unauthorized modification**: An agent or service modifies data in an incorrect fashion or destroys the data without authorization.
- **Denial of service**: An agent or service consumes an inordinate amount of a shared resource, thus preventing other programs from progressing.
- **Trojan horses**: An agent or service mistakenly uses code that has been maliciously modified.

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 comes from the site that it claims to have come from, and that it has not been tampered with from the time it was sent.
2. Verify that the agent program is well formed, and that it possesses the necessary credentials to execute on the platform.
3. Grant the agent access to local resources and services.
4. Grant local services the necessary rights to communicate with the agent.
5. Allow the agent to execute while enforcing that each communication between the agent and the rest of the platform is authorized.

The security architecture of an agent platform must cover all of these aspects. Our kernel approach is to focus on points 3, 4 and 5 as they are essential for providing execution guarantees. Points 1 and 2 counter exogenous threats and are implemented by user-level services in JavaSeal. In this way, the application can choose its own encryption algorithms, and its own authentication policy.

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 to which the security policy does not need to check access. 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 [10]. 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 [25]. *Legitimate channels* are mechanisms included in a system precisely as a means of communication, *e.g.*, sockets, object

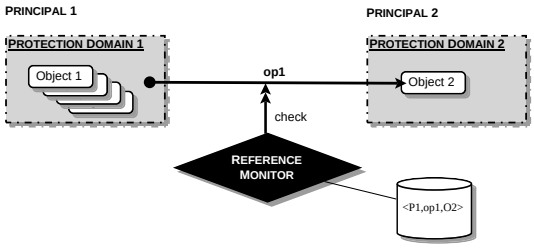

Figure 1. A system's security architecture. The reference monitor intercepts each access to an object from a remote domain, and queries the security policy.

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 shared buffers and kernel variables. The last category is that of *covert channels* which are means for programs to communicate by exploiting a visible system characteristic in an unconventional manner. For instance, a program can signal the value of a PIN to its environment by creating a file with exactly that number of bytes. 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 and data into a protection domain.
5. Termination of a protection domain.

These operations represent all domain operations allowed in JavaSeal. Protection domains are represented by the seal abstraction. Mediation and encapsulation of the ARM are achieved by a combination of language mechanisms. JavaSeal security addresses standard and storage channels; covert channels are not specifically dealt with.

The following are the three security properties that an implementation of JavaSeal must have. Note that these properties must hold for all programs and all services.

- **Confinement:** This 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.
- **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 code.

The guarantees are enforced entirely by language based protection mechanisms. The interesting point is that the default security model of Java is not sufficient to enforce any of our requirements as we discuss in Section 4.

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 platform.

We now present a high-level overview of the system and discuss how seals are used to structure mobile agent applications. Section 5 describes the actual implementation.

3.1 Seal Hierarchies

A seal is a self-contained program, and protection domain, with its own data, code and execution threads. 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 referred to as its direct parent. The set of children and parents of a seal refer to the transitive closure of direct children and of direct parents respectively. The

base of the hierarchy is the *root seal* which houses kernel code.

The key feature here is the strict encapsulation that is enforced at seal boundaries. Sharing of objects, classes, or threads is disallowed. Instead, every object and thread belongs to a single seal and the resources it consumes can be charged to that seal.

Seals communicate solely through named channels. A seal can send a message only to its direct parent or to one of its children. Messages to distant seals, such as most service requests, are encoded as a sequence of neighborly message exchanges and every seal along the way must accept to transfer the request.

Only the core services (seal creation and destruction, communication, memory management, scheduling and state capture) are under the control of the kernel. 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].

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. This is essential for protecting visiting agent seals from their local environment.

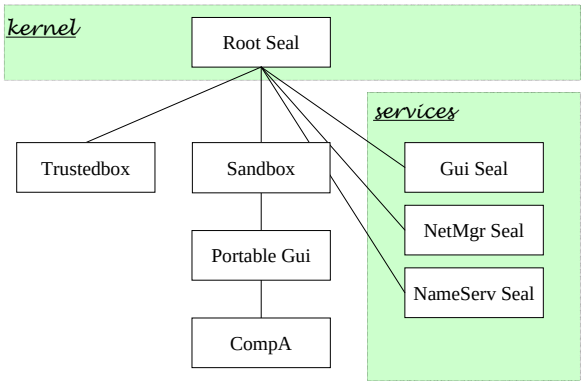

Figure 2. A seal hierarchy. The root seal runs three service complets and two envlets. Sandbox hosts untrusted agents while Trustedbox hosts friendly ones.

Two types of agents: JavaSeal has two categories of agents. The leaves of the seal hierarchy, which are called **complets**, are "traditional" mobile agents. Intermediate levels of the hierarchy, called **envlets**, are mobile environments. The role of envlets is to interpose between requests of a complet, or nested envlet, and its environment. They can play the role of adapters when the services on the current platform do not match an agent's expectation or act as the facilitators of [21]. Envlets can also implement a security policy. Figure 2 shows a platform running a complet named CompA. The requests that this complet is allowed to make to services such as the NetMgr seal are filtered by the Sandbox envlet. This envlet is provided by the local environment to control the behavior of CompA. Depending on the current policy, the Sandbox seal may choose to disallow all network communication or only allow communications with a restricted set of sites. Service seals are usually represented as complets.

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

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 and can stop a subseal. Thus even on a trusted machine and a trusted kernel there may be a malicious host problem. However, malicious envlets are restricted by the core kernel protection mechanisms, and can not, for instance, modify the memory of a child.

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 multiple different channels for different purposes. Use of channels is regulated by a separate access control mechanism called *portals*.

Channels are synchronous. The sender thread blocks until a receiver accepts the message. In order to have multiple outstanding requests a seal must create multiple threads. Values exchanged over channels are transmitted by copy to avoid the accidental or malicious introduction of storage channels between seals.

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 A sending a message to its parent on a channel named `netreq`:

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

The channel `netreq` is created if it does not already exist. 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. For the communication to fire the parent must have first opened a portal allowing A to use `netreq`:

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

This allows one use of the `netreq` channel. 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.

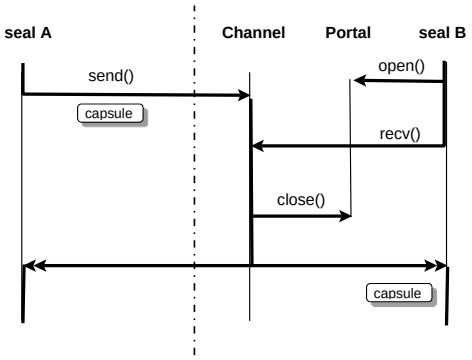

Figure 3. Channel based communication.

Communication between a sender seal A and receiver seal B is illustrated in Figure 3. We assume that A and B agree on a channel name, initial agreement is achieved using a set of pre-programmed channel names. A strand executing within A invokes the `send` primitive on the channel with a capsule – a message container – 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.

The choice of synchronous communication is somewhat controversial. Many systems offer asynchronous communication mechanisms [8, 2, 26]. 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. It is not possible to flood an agent with requests since the kernel limits the number of threads in any given seal.

Programming with channels and portals is sometimes cumbersome. Other communication utilities can be designed to run over the channel mechanism. One example is a shared object space that offers the possibility for associative, asynchronous and anonymous communication between objects [7]. This proposition includes access control mechanisms to protect the contents of entries in the shared space.

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* object. This format is used for storing the seal on disk, and for transferring it on the network. Archives are used to restore seals; 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 archive. 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 archive objects is larger and thus their transfer costs more. We have developed a custom code compressor called Jazz [6] 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.

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 and executes by default the `run` method of the seal.

4 Limitations of Java Security Model

JavaSeal has strong security requirements since its goal is to enforce separation between seals. We considered the Java security architecture [36, 17], but after a detailed investigation concluded that it is not sufficient to guarantee the properties of confinement, mediation and faithfulness that we mentioned earlier. We also identified some serious denial of service attacks that can jeopardize the entire 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`. 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 name spaces. That is, all classes of each applet are considered to have a distinct type from the same class in other applets. 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 the Java 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 [36]. 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 *what applet* is using this code. It is clear that the primary goal of Java's security architecture is to protect the virtual machine from the applets and programs running over it.

There are ways to circumvent Java security. We identified a few in earlier work [35], here we focus on those related to the 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 due to shared library classes. In Java `static` variables are variables of a class that are visible to all instances of that class. 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. Every object in Java has an associated lock. When two domains can access 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.

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 it blocks and loops forever, leading to a denial of service attack.

Denial of service attacks are effective since Java does not have a resource management interface that could 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. Finalizers 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 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 does not 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 system runs on JDK1.2, though can run on 1.1 with only slight modification. The body of the system is structured as several 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 which has a list of classes that can belong to these packages.

5.1 JavaSeal Kernel classes

There are only a 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 exchange capsules. Class `Portal` is used to control access to channels. Class `Strand` is a restricted version of `Thread`, the name has been changed partly to avoid confusion. Finally, the `SealLoader` class implements seal loading and verification. Some interfaces are shown in Figure 4.

5.1.1 Seals

A seal is made up of classes, objects, threads and a `SealLoader` object. When a seal is created, the name of its main class is specified and a `SealLoader` is created to load all classes needed by the seal. If all classes pass the extended verification discussed in Section 5.2, an instance of the main seal class is created. The seal thus consists of a base object and all objects reachable from that root. A seal's objects and classes are not shared with any other seal. JavaSeal's use of class loaders is a key element in achieving seal domain separation. A type cast error is raised whenever an object of one domain attempts to directly reference an object of another domain.

A `SealLoader` has two ways to resolve classes. System classes are found in predefined locations on disk. User-defined classes are stored in a seal's *archive*. As mentioned, the archive is used to enforce faithfulness: a seal always uses the classes with which it was defined and so does not rely on any other seal to furnish it with a (perhaps infected) version of its classes. Furthermore, seal archive files are immutable – a seal may not add new classes to its archive during execution. An advantage of this is that the archive may be digitally signed and so any attempt to tamper with the code can 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

```
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 4. The JavaSeal kernel classes

the class archive into a seal archive. This can be used to re-instantiate the seal, or alternatively it can be sent over a channel and then re-instantiated within another seal.

There is a small number of exceptions to the constraint that classes not be shared between seals. In effect, basic classes like `java.lang.Object` must be shared. One reason for this is practical: Sun's JVM does not allow multiple instances of class `Object`. The other reason is that shared classes are needed to effect communication between seals.

Shared classes are loaded by the predefined system loader. In Java, a class can only reference other classes loaded by the same loader or by the system loader. For instance, consider 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 interface 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 feature is used to maintain internal links in a seal to the seal's children and parent. In effect, a class `InternalSealPointer` is a shared interface which points to the related seals, even though seal classes possess their own class loaders.

Seal 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 [9].

5.1.2 Strands

The threads that execute inside of seals are called *Strands*. The crucial difference between threads and strands is that strands are bound to the seal in which they were created and that they cannot be used to gain information about other seals. 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 a 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'');
Capsule cp = new Capsule(str);
ch.send(x, cp, Seal.getParent());

String x = new String(''req'');
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. 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. If the receiver is not yet ready, the message remains stored on the *sender's* side. Thus, the message queue is moved with the sending seal if the seal moves before the receiver is ready. It also means that a seal cannot cause another seal to have a memory overflow by sending it large messages.

5.1.4 Capsules

Capsules transport data over 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 objects in the sending seal. A capsule is created by specifying a root object and copying all the objects in the transitive closure from that root into the capsule. This is currently done using Java serialization.

A capsule may only be opened once. Opening a capsule requires that all classes be available in the current seal before releasing the capsule's contents. If the `SealLoader` is not able to find all classes required by the capsule, then the `open` operation fails. The classes found might have different versions; we rely on Java type compatibility rules to verify the validity of a capsule.

5.1.5 Performance measures

We conducted some measurements on a JavaSeal kernel running over SunOS 5.6 on a 333 MHz UltraSparc-IIi processor with 124MB of main memory. The time taken to create an agent (with an empty `run`() method) is 45 milliseconds. The platform can support over 1100 concurrent agents (where each agent has its own executing thread); further creates are very slow, resulting from too much swapping. Inter-agent communication is slow. In one configuration measurement, a child sends a request to its parent who sends back a reply. This mimics an RMI call and return between seals (involving two seal context switches) and yields an average cost of 414 μ-seconds. A communication through the hierarchy from a child A to its parent and then to another child B takes an average of 380 μ-seconds. Agents and messages are exchanged in capsules. A capsule containing the *null* object is 5 bytes. A capsule for a 100-byte array is 127 bytes. A capsule for an agent with an empty `run()` method occupies 1287 bytes, and an agent with a thread blocked on a receive has a 1644 byte capsule. A sample agent containing an article representing the HTML page `www.unige.ch` has a capsule of size 84434.

5.2 JavaSeal Security

We mentioned the set of ARM operations in Section 2.3: protection domain creation and termination, loading code and data in a domain, and control of communication between domains. The concepts whose implementation we outlined in 5.1 are designed precisely with this functionality as a goal. JavaSeal's goal is also give precise security guarantees: confinement, mediation and faithfulness. This subsection summarizes how these properties are met.

Confinement There are four basic mechanisms needed to enforce this in JavaSeal:

1. Class loader enforced domains: by assigning a class loader to each seal, Java's typing rules prohibit user-level classes and their objects from being directly shared.

2. The capsule mechanism is designed to ensure that no inter-seal alias can result from communication.

3. Threads do not cross seal domain boundaries. The effects of killing a thread are thus confined to the owning seal.

4. A number of restrictions are placed on the classes loaded into a seal by `SealLoader`: **1** Only classes from `seal.usr` or system classes from `CLASSPATH` may be loaded. In particular, no user-defined subtype of seal kernel classes may be used. **2** Stringent restrictions on finalizers are imposed. 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.

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.

As mentioned, **mediation** is obtained by nesting a target seal in another seal, the second seal being responsible for interposing on sensitive channels. **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 requirement for the agent reference monitor is that it be protected from tampering by user seals (section 2.3). The agent reference monitor of JavaSeal is implemented within the JavaSeal kernel, which is shared by all seal domains. The kernel is made up of kernel classes – the `seal.sys` package – as well as base JDK classes. Classes that are shared between domains, like communication classes are also within `seal.sys`. There are too many classes within the JDK to allow these to be shared by all domains. Kernel security in JavaSeal is concerned with protecting the kernel from tampering by user seals, and minimizing the chance of storage channels being created by the kernel for seals to exploit.

The key to enforcing kernel security is to have a **well-defined** and **small** interface between the kernel and the program seals. This is done in a number of ways.

First, the kernel executes inside of the root seal. Whenever a seal requires a JDK-like service, e.g., a reading from disk, it must send the request to the kernel over a channel. The only point where a seal is allowed to exchange arbitrary objects with the kernel is as arguments in the capsule of a message that the seal exchanges with the root seal. However, capsules are always opened in the receiving environment and so a seal can at no time gain a reference to a kernel object.

Second, the number of classes actually shared between seals and the kernel is kept to a minimum. We cannot enforce strong isolation for the kernel classes since some key JDK classes have to be shared. The JavaSeal kernel classes are also shared. 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.

The final part of 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. A different set of access modifiers can be given to different seals. In this way a GUI service seal has access to GUI classes that must remain invisible to other seals.

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 (or inexpensive documents) on the Internet. 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 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 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.

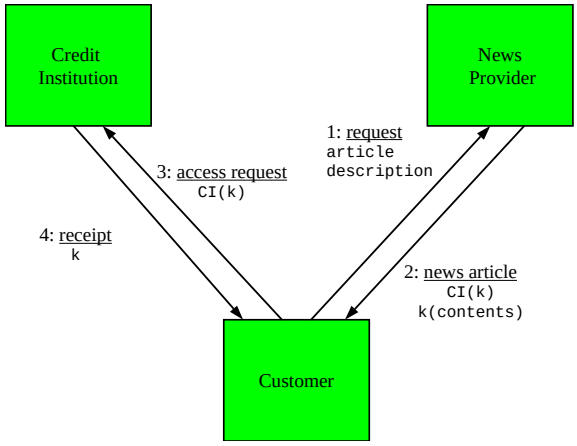

Figure 5. HyperNews allows customers to request news articles from news agencies. Payment is handled through trusted credit institutions. All exchanges between any two sites are encrypted by a session key negotiated between those sites.

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 a certain symmetry in the architecture. For instance, a client can act as a press agency. He can collect articles, 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. This is because the initial article remains encrypted with a key chosen by its owner.

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 become visible 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 an acceptable risk to run.

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 using their feeds or articles, 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.

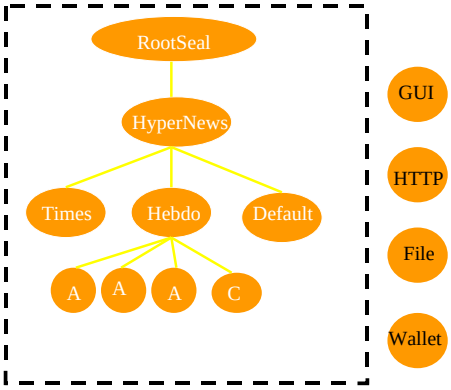

Figure 6. The HyperNews application. News Feeds for the Times, Hebdo (a Swiss magazine) and a default news feed are shown. These are envlets with couriers and articles as complets. The dotted line represents the trust boundary, that isolates mobile seals and the network area from trusted immobile services.

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.

6.3.1 Architecture

The overall structure of a running HyperNews platform consists of a number of News Feed agents and a large number of article agents (see Figure 6). The News Feed agents are envlets that manage all the data and services common to one news provider. For example, a News Feed 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 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 News Feed 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 daemon 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 hierarchy point of view, encapsulates all platforms of the mobile agent network as children [33]. 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.

6.3.2 HyperNews 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 News Feeds 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, News Feeds are not allowed to communicate. Security on the articles is enforced by the News Feeds which decide whether the articles that they host may communicate (usually there is no need to).

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 [37], TACOMA [31] and M0 [32]. 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.

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 [18] (from Tcl/Tk), Voyager [16], Ajanta [24] and Aglets [26]. 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 to the agent domains. A protection domain in the J-Kernel is also a name space implemented using a class loader [19]. 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.

Sun's JVM (JDK1.2) includes many changes to the security model — including protection domains based on dis-

tinct 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. More work is needed on Java security for resource control, and for isolating domains from one another. As we mentioned, type casting and sharing can easily violate the constraint that one domain not reference an object of another domain (name space).

The hierarchical communication model has been inspired by the Fluke micro-kernel [13], L3 [27], and work on interposition in operating systems [11, 14, 15]. We have not addressed interposition of low-level resources such as memory and the scheduler as this requires modifications to the virtual machine [3].

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 functionality. 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, JavaSeal still has some efficiency problems, with respect to data transfer times and 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 This research was conducted while the second author was in the OSG at the University of Geneva. The authors thank Walter Binder, Manuel Oriol, and Karim Taha for their work on JavaSeal, and Jean-Henry 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] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. *ACM SIGPLAN Notices*, 33(10):21–35, Oct. 1998.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] G. Glass. ObjectSpace voyager — the agent ORB for Java. *Lecture Notes in Computer Science*, 1368:38–48, 1998.

[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.

[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*, 1419:188–201, 1998.

[24] N. Karnik and A. Tripathi. Agent server architecture for the ajanta mobile-agent system. In *1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1998.

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

[26] D. B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web Journal*, 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] Tromsø University and Cornell University. TACOMA Project, http://www.cs.uit.no/DOS/Tacoma/.

[32] 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.

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

[34] 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,.

[35] 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.

[36] 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.

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