Designing JAVASEAL
or
How to Make JAVA Safe for Agents

Jan Vitek
Ciarán Bryce
Walter Binder

Abstract

JAVASEAL extends the JAVA programming environment with a model of mobile “agents” and strong hierarchical protection domains. These extensions are based on a formal model of distributed computation which we briefly overview. We then present the design and implementation of JAVASEAL and discuss the engineering compromises involved in integrating our model of mobility and protection within JAVA.

1 Introduction

The JAVASEAL system is a micro-kernel mobile agent system developed within the framework of the ASAP project. The objective of the project is to deliver a platform for securely moving active JAVA programs over the Internet. ASAP is used as the supporting infrastructure for several Swiss research projects. In particular the HYPERNEWS project uses JAVASEAL to package and deliver newspaper and magazine articles over the Internet [11]. In HYPERNEWS copyrighted digital documents are represented by so-called mobile agents, that is, programs which may move autonomously across the network. This allows digital documents to have behavior inextricably associated to them; in the simple case a document implements a straightforward access control policy: contents are decrypted only after payment has been approved by a recognized financial institution. More elaborate access control policies and behaviors can easily be implemented. The HYPERNEWS document distribution model also feature proxies for the different news providers. These proxies are installed on the subscribers sites and take care of the user interface and customer interaction. These programs are installed on a permanent basis.

The requirements for JAVASEAL are that it must provide a portable execution environment over different operating systems and hardware platforms. The agents themselves must be runtime portable: the same document may move to—and execute on—several environments in its lifetime but its semantics should be invariant, i.e. newspaper articles that can only be read on a particular operating system are not acceptable. The second requirement is that behavior (i.e. code) must move along with data; as an example consider an encrypted article, the decryption code is not necessarily present on every Internet host, it is therefore necessary to be able to associate code with the article. The third requirement is that a disconnected mode of operation be supported to allow customers to read articles they have purchased even when their machine is disconnected from the network. Finally, the last requirement is that JAVASEAL must be secure,
i.e. it must allow for the expression and effective enforcement of security policies; without this, mobility remains an academic toy. The security needs of HYPERNEWS are representative of most mobile agent systems: the documents and proxies should be protected from attacks by other agents residing on the same JAVA SEAL platform and the platform (by extension the host computer) must be protected from malicious actions of “foreign” agents.

SUN Microsystem’s JAVA programming language partially addresses these requirements. The language run-time system—the JAVA virtual machine (or JVM for short)—has been ported to most relevant operating systems and hardware platforms. JAVA specifies an intermediate program representation, the JAVA bytecode format, which is hardware independent. Programs in this format can be loaded dynamically into an executing JVM. The JAVA language and JVM incorporate extensive security mechanisms. All in all, JAVA is a reasonable choice for building a mobile agent system and it is no great surprise to see that a number of agent systems have adopted it.

Unfortunately, choosing JAVA presents some problems. During the course of the ASAP project, we discovered that none of the existing agent systems could provide satisfactory security guarantees. In fact, the JVM security model itself is inadequate for mobile agent systems and, in general, any application with security requirements that go beyond protecting user files from deletion.

JAVA SEAL takes a fairly radical approach: the JVM security model is replaced by a model which allows multiple applications (agents) to execute in isolation. All interactions between agents are explicit and may be subject to security restrictions. This allows us to offer guarantees about what an agent can do that are much stronger than in the JVM. Security comes at a price, currently none of the libraries provided in the standard JVM are available to agents directly. Instead we must rebuild all core services from the ground up, using JAVA SEAL as a micro-kernel.

This paper discusses the design of JAVA SEAL and presents its preliminary implementation. We start by presenting the SEAL calculus—the formal model of computation behind the JAVA SEAL system. Section 3 reviews JVM security mechanisms focusing on those that are exploited in the implementation. Section 4 presents the design of the system and its current implementation. Related work and future extensions are discussed in sections 5 and 6 respectively.

2 A Formal Model for Security and Mobility

The SEAL calculus [12] is a synchronous higher order distributed process calculus which was inspired by the π-calculus [10] and the Ambient calculus of Cardelli and Gordon [6]. The goal of the calculus is to provide a notation for expressing and reasoning about security and mobility properties of processes in a distributed system, but the SEAL calculus is also designed to be a programming language, hence it is essential that its basic abstractions be implementable in the context of a large scale open distributed system.

A seal, which gives its name to the calculus, is a process running multiple concurrent threads of execution, protected from other processes by a protection domain boundary. All resources
and activities that belong to a seal are within its protection domain. Communication across domains is by messages exchanged over named communication channels. When a seal wants to communicate with another seal, the recipient must explicitly open a portal to allow the communication to take place. The messages that are exchanged are passed by value to prevent cross protection domain sharing. The calculus allows nested seal structures of arbitrary depth. A seal configuration can be represented as a tree with a distinguished root seal; Fig. 1 presents a configuration, vertices represent seals and edges inclusion relationships. In the diagrammatic notation, if two seals are connected by a single edge, the highest seal is referred to as the direct parent. The parents of a seal is the transitive closure of the direct parent relation. Children and direct children are defined symmetrically.

The seal communication model uses located interaction. Processes communicate over channels and all channels exist—are located—within a specific seal. Thus there are three modes of interaction: (1) local communication: two processes within the same seal communicate using a local channel, this is analogous to method invocation in an object oriented programming language, (2) external interaction: a process communicates with another process located in the parent using a channel also located in the parent, this requires the parent to open a portal to allow the child to communicate, an analogy in programming is a system call, and (3) internal interaction: a process communicates with another process located in a child seal using a channel located in the child, here it is the child that must open a portal, the analogy is with an upcall from the operating system. Fig. 2 illustrates the three modes. The important points to note here are that portal provide symmetric protection, that is, they protect parent from children as well as children from parent. Secondly, the communication primitives do not allow siblings to communicate directly. The implication is that all messages must travel up the hierarchy to the least common ancestor and then down again to the target.

The (strong) argument in favor of this communication model is twofold: the first is that

![Diagram of seal configuration]

Figure 1 A configuration composed of five seals; edges indicate inclusion relationships. The root seal $S_1$ models the network, child seals $S_2$ and $S_5$ model hosts running a JVM. Finally, the seals $S_3$ and $S_4$ model agents executing on $S_2$. Seal names ($S_1 \ldots S_5$) are handles known only by direct parents; naming is relative.
Figure 2 Three modes of seal communication: (a) two processes communicate over a local channel, $x$, (b) a process located in seal $S$ communicates over channel $x$ of its parent seal ($\uparrow$ denotes a direct parent), the parent has opened a portal to allow $S$ to write on channel $x$, (c) a process located in the parent seal communicates over a channel located in $S$, the latter has opened a portal.

this guarantees message delivery without a complex infrastructure,\(^2\) the second is that it allows enforcement of a hierarchical security model. A seal may specify a security policy with respect to the communication abilities of its children. The hierarchical model guarantees that the policy can not be circumvented, in particular, that it is possible to ensure subseals will not exchange messages unless explicitly permitted to do so.

The seal calculus supports strong mobility. A seal along with all of its children may be moved in one, conceptually transparent, atomic operation. The moves are transparent in the sense that seals are not notified and if the new environment provides exactly the same services, they need not be aware of moves. All moves are initiated by the direct parent, the seal itself has no control. There are three kinds of moves: (a) a seal can be moved locally (this is only used for renaming), (b) it can be moved one level up the hierarchy, (c) or one level down. Move primitives actually coincide with communications primitives, they use channels and portals in a similar way, and thus the same protection mechanisms control all seal interactions. Fig. 3 illustrates the three mobility modes.

It should be clear by now that the abstractions provided in the SEAL calculus favor local interaction over remote interaction. For two seals located in different parts of a hierarchy (e.g. on different network hosts) to be able to communicate requires either that one of them moves into the other or that messages be exchanged. In any case, the interaction can only take place if all seals separating the two communicands allow it. One of the goals of the hierarchical model

\(^2\) We considered allowing any pair of seals to communicate directly. This would require some form of globally unique names and tracking infrastructure which would be difficult to implement in a distributed system subject to failures, large latencies and, especially, mobile communicands.
Figure 3 Three modes of seal mobility. (a) seal $S_2$ is moved locally by $S_1$ and renamed $S_4$, (b) $S_2$ is moved into the parent seal, (c) $S_2$ is moved into child seal $S_3$.

is to express mobile programs and the immobile environments in which they execute within the same formalism. For this reason, the same abstraction, the seal, is used to represent both mobile programs and their immobile host runtime system. For example a configuration may have a seal standing in for a JVM, and a set of nested subseals representing applets running on it, giving a two level structure. More interesting examples include multiple levels of nesting, where nested seals implement some service and delegate the implementation of other services to their parent. By blurring the distinction between mobile and immobile entities it becomes possible to talk about, e.g., mobile sandboxes, mobile security policies and mobile virtual machines. The importance of hierarchical models and nesting has been recognized in the operating systems community [9].

Another key point of the model is that each seal has full control over all visible actions of its subseals. This is essential for security as it allows for the enforcement of comprehensive policies. A side effect is that a parent may prevent a subseal from ever leaving, which can be a security problem in itself in the presence of malicious parents. The model does at least prevent the parent from interfering directly in the internals of a seal (unless the child opens a portal, the parent is not allowed to use the child’s resources), this is important to protect a seal against breaches of integrity and privacy of seals.

The differences between the SEAL calculus and the $\pi$-calculus and ambient calculus are the following. The $\pi$-calculus is not a truly distributed model of computation; it has no direct notion of failure, no notion of mobility (in the sense we have described above) and channels exist in a flat name space. In other words, processes can always communicate (if they agree on a channel). This unrealistic in presence of firewalls and network partitions, and quite difficult to implement. Finally, the $\pi$-calculus has no built-in protection mechanisms (Abadi and Gordon have shown how to use the $\pi$-calculus for modeling security protocols and some basic cryptography, but this a different issue [1]). On the other hand, the ambient calculus is a truly mobile calculus. Ambients are hierarchically structured just as seals. Mobility primitives are similar, they too can be viewed as tree rewriting, with the major difference that the ambient calculus allows nodes to be collapsed, using the open primitive. We have considered this primitive and decided not to include it in the calculus as it would undermine security; opening a seal releases processes
capable of opening portals, moving subseals, and so on. The other major difference is the use of names for security. In the Ambient calculus an Ambient name is a capability which can be granted in full or in part to a process. Capabilities control all mobility primitives. Once issued they can not be revoked. In the SEAL calculus names are not as important, instead portals act as linear capabilities. In other words, it is possible to revoke a name by not opening (or reopening) a portal for it.

3 Protection in the JVM

Before presenting the JAVA SEAL system, we must briefly overview certain features of the JAVA language and of the JAVA virtual machine (JVM).

It is important to note that the JAVA programming language does not deal with security but rather with safety. Security is the realm of the JAVA virtual machine and the JDK libraries. This distinction is important to the discussion, as we do not argue against the language, but rather against the design of the virtual machine and libraries.

3.1 A Three-level Maginot Line

The JVM security model relies on the core safety property of the JAVA language. This property guarantees that object references can not be forged or manipulated within JAVA programs and that all accesses to memory are strongly typed. Hence, a method can only be invoked if the target object actually implements the message and the method is accessible to the caller. The safety property essentially forbids all forms of pointer arithmetic so widespread in C-based languages. It is enforced by a mix of static and dynamic checks; programs are mostly statically type checked, but arrays index errors as well as stack overflows, among others, must be checked at run-time.

The JVM was designed to allow users to download executable content—JAVA applets—from the network and run these applets on their machine. Consequently, the JVM comes with a number of protection mechanisms that are used to implement customizable security policies. These mechanisms are designed to allow a site to reduce the set of system operations that an applet can invoke and ensure, to a degree, that an applet does not maliciously interfere with the JVM. Security relies on three lines of defense: bytecode verification, class loading and stack inspection.

The first line of defense is the class verifier whose role is to check structural and semantic validity of class files as they are loaded in the JVM. The most interesting feature of the class verifier is the bytecode verification algorithm which performs a data flow analysis over the code of each class to formally prove that the applet’s code does not breach the safety property of the high level language semantics. In effect, this process attempts to detect hand-coded bytecode sequences designed to break the JVM safety.

The second line of defense is the class loading mechanism. A class file contain numerous unresolved references to other classes that are used by this class. Class loading refers to the process of resolving those references. In the JVM, class files are loaded by a ClassLoader
object. The role of the loader is to link new classes against previously loaded classes or, if the requested class is not found, to load it either from disk or from the network and then link the class. This mechanism is very powerful as it gives full control over the execution environment of applets. For instance, the loader may disallow the sharing of one particular class between applets by loading the class twice, once for each applet, in effect creating separate type environments for each applet. This control can be used to prevent the sharing of variables and to prevent applets from installing dodgy versions of popular classes. A number of core classes, implementing the basic JAVA abstractions (classes, threads, strings, etc.), are always loaded by a special class loader from local disk. These library classes are shared amongst all applets.

The third, and last, line of defense is represented by the stack inspection mechanism. Stack inspection is used by objects of the type SecurityManager to verify the identity of the requestor of a system service. Principals are associated to classes on the grounds of the origin of the code (originally the difference was only local/remote, now digitally signed code allows finer distinctions). Whenever a system operation is about to be executed, a call to the security manager must be issued, it checks whether the call is allowed. This decision is taken by inspecting the call stack of the thread which issued the call, the call is allowed only if all methods on the stack are trusted.

3.2 Inadequacy and Insecurity

The JVM model is adequate for protecting a single user from the dangers of executable content downloaded from shady Internet hosts but fails to provide a secure basis for building complex applications composed of untrusted or fallible components such as our agent systems.

Past the well known, extensively documented, and hopefully already fixed, security bugs of current JVM implementations, there are a number of basic deficiencies in the JVM security model that make it almost impossible to provide any real security guarantees. Vitek, Serrano and Thanos have discussed security issues in the context of object oriented languages in [13]. Some of the same problems have been discussed in the context of JAVA by Hawblitzel et al. [7], and Balfanz and Gong [3].

All the difficulties mentioned in those papers stem from the fact that, at heart, the JAVA virtual machine is a single user operating system. Its design does not take the requirements of multi-processing into account. Protection mechanisms are almost an afterthought. Certainly they were not meant to protect multiple interacting applications. Thus, the JVM security policy has no provision for applet interactions, nor does it make any attempt at accounting resources.

A secure mobile agent system implies multi-processing support. Agents are applications, running with different privileges on the behalf of different principals. Thus protection mechanisms have to protect (1) agents from other agents, (2) the execution environment (VM) from agents, and (3) agents from the execution environment. We argued in [13] that capability based

---

3The protection model for applets is really a case of "them against us": the host system is protected against "untrusted" applets in toto, without distinction between applets. Moreover, the protection machinery is not appropriate to prevent applets from attacking each other. The only protection between applets is total isolation implying that applets can not interact at all. Not only is this too restrictive, but the policy is not enforced by the implementation.
Designing J AVA S EAL

protection at the object-reference level is too fine grained to be practical. Roughly, this is because pervasive aliasing makes it almost impossible to define a clear interface between applications. Object references can very easily leak from one application to the next.\(^4\)

Agent systems require a notion of strong protection domains which preserve the following characteristics:

- **Domain Isolation**: Unless otherwise specified, each object should belong to a single protection domain, the domain in which it was created. The implementation should not allow for language level values to flow from one domain to the next.\(^5\)

- **Total Mediation**: The interfaces between protection domains must be explicit and the implementation must guarantee that this interface is respected. In particular, the value and type of the data exchanged between domains must be specified and actual messages checked against this specification.

- **Accountability**: Every resource in use, and every operation must have a clear, unequivocal author, that is all objects and threads should be attributed to a protection domain. Not only is this absolutely essential for resource usage accounting, e.g. memory and cycles, for preventing denial of service attacks, but it is also mandatory for protection domain termination.

In the SEAL model, seals correspond directly to protection domains, hierarchical domains in fact. It is thus possible to associate both principals and privileges to seals. They facilitate resource accounting as there is no sharing, each object belongs unambiguously to a particular seal.\(^6\) Interfaces between protection domains are explicit and guarantee total mediation as cross protection domain communication is differentiated syntactically and require open portals, in other words permissions must be granted before cross domain interaction is allowed. Accountability follows from the fact that every operation is performed within a particular seal and thus is performed with the authority of the seal.

Current JVM implementations do not have these three characteristics. We will now present the implementation of J AVA S EAL and discuss how it addresses these concerns.

**Design note (1)**

Wallach et al. \[14\] estimate the number of protection domain crossing to about 30'000 per second over a typical JVM workload.

Convert storage channels exist on several levels. The core JDK classes, classes shared by all applications contain over 800 shared variables. Many of these are easily accessed

---

\(^4\)An embarrassing example is the recent defect in an implementation of SUN’s digital signatures which permitted any applet to become trusted, simply because an alias to the system’s internal list of signatures was being returned, instead of a deep copy of that list.

\(^5\)By this we mean actual JAVA objects, of course the task of ensuring that information does not flow between domains is much more challenging.

\(^6\)Immutable objects can be shared for efficiency purposes. Semantically this sharing can not be detected. This would complicate the task of resource accounting, and garbage collection.
(public!) and some are of object type. In cases when these objects belong to classes that are not final or that have non-final members, applications can substitute them with subclasses and can thus freely exchange JAVA objects, breaching any form of confinement.

More subtle covert channels abound. For instance, the activeCount method of class Thread is a low bandwidth covert channel. Two application can communicate by creating threads and killing them. Statistical analysis will extract the information from the noise coming of other application’s thread creation behavior.

Threads also pose problems as an application may terminate a thread running in another protection domain, leaving the other domain in an inconsistent state.

The type system may be used for code injection attacks. When agents communicate by method invocation, it is possible for the caller to substitute for the declared type of an argument any subtype. In particular, it may substitute an object that contains code which will break the receiver.

Domain termination is difficult to implement in the current JVM because of the interaction between object finalization and the implementation of garbage collection. The finalization code may be executed at any time by a deamon thread belonging to the garbage collector. This means that even when an application has terminated and all of its threads have been killed, it may still continue to execute. Furthermore a simple denial of service attack involves finalizers that never return.

4 Designing JAVA SEAL

JAVA SEAL is micro-kernel for agent systems. The goal of our design was to provide a minimal set of features needed for general purpose agent programming. Thus JAVA SEAL should not be viewed as a user-level programming environment, it is too low level for programming large systems comfortably, but rather as a basic brick on which domain specific agent systems can be implemented. A domain specific agent system includes services and syntactic extension (sugar) for common agent coding idioms. For example, we are implementing a domain specific agent system for the HYPERNEWS project.

In this paper we focus on JAVA SEAL. We start with an overview of the architecture and continue with implementation considerations.

4.1 JAVA SEAL Architecture

The architecture of a JAVA SEAL platform is illustrated in Fig. 4. Each JAVA SEAL platform runs a single RootSeal, this object is in charge of starting up other seals and initializing the ground security policies. The root also instantiates a SealLoader responsible for loading and verifying user seals (for simplicity we refer to user seals as agents). The root is also responsible for starting up and managing services. The exact services provided will depend on the application, currently we provide the NetMgr service seal to manage network interaction and the SpinGui service seal which provides a Html user interface for agents.
Figure 4. A JAVASeal platform consists of agent and service seals running within a RootSeal. The SealLoader is responsible for loading seal classes and enforcing static integrity constraints. The JDK classes are almost entirely hidden by the JAVASeal packages, they are accessible only to service seals.

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. Agents are not allowed to refer to classes outside of these packages, furthermore, user code can only be added to the package seal.usr. This restriction is enforced at class loading time by the loader. Service seals are allowed to refer to standard JAVA classes; these seals are trusted components which can not move and must be installed by the platform administrator.

Several distinct roots may be started on the same JVM, this has the effect of creating logically distinct JAVASeal platforms, which can, for instance, implement different domain specific systems. But having several distinct machines has disadvantages. For one, they must communicate through the network interface, and secondly it is harder to implement a coherent security policy. The preferred way of structuring applications is to represent different environments as subseals of the same root (e.g. one subseal could support the digital document model of HYPERNEWS, the other could be an interface to a database allowing itinerant agents to perform local queries). This allows communication between the environments (either to extend the database with the contents of the digital documents and thus allow queries on both, or to allow digital documents to contain queries to the database) and common security policies to be expressed (ex., disallow articles to leave after performing queries larger than some limit).

There is a small number of exceptions: classes like Object must be known. In total there are 20 classes from java.lang, java.io and java.util that must be visible. The majority of those are exceptions or interfaces.
Design note (2)

The first decision made for JAVA SEAL was whether to implement the SEAL model within the JVM or to program it as a JAVA package. A JVM extension would allow a more faithful implementation of the model, in particular we wanted to experiment with strong mobility (i.e. mobility of active threads). Practical considerations held us back. Structuring JAVA SEAL as a package allows the system to be transparently portable to all JVM implementations, which is a very important concern for our partners. Furthermore, it allowed us to prototype much faster.

4.2 JAVA SEAL Kernel classes

The interface to the JAVA SEAL system is composed of a small set of classes that model the SEAL calculus concepts in JAVA (shown in Fig. 5). We review these classes briefly, a more thorough description is given in the following subsections.

The Seal class is the abstract class which defines the common behavior of all seals, currently it has two subclasses: RootSeal which the single root of the seal hierarchy and Agent which is represent mobile computations. Handle objects are used to represent reference to seals (either direct parent or direct children). The Chan class represents a communication channel, it has only methods for synchronous sending and receiving of messages. The subclass TypedChan has been defined to allow a style of programming similar to RMI, with

Figure 5 An overview of the classes in the JAVA SEAL packages. Shaded boxes indicate classes that may be extended in user code.
channels having interface types composed of a set of operations. The class \texttt{Capsule} and its subclasses represent the message values transferred along channels. \texttt{ObjCap} contains a single object, \texttt{MsgCap} has been designed for RMI style communications, and \texttt{SealCap} contains a seal. The latter encapsulates all the information sent when seals migrate. The class \texttt{Strand} is the \texttt{JAVASeal} equivalent of a thread of execution, in fact it is a modified version of the \texttt{Thread} class. The name was changed simply to avoid confusion between two different thread abstractions. Finally, the \texttt{SealLoader} class implement seal loading and verification.

4.3 Seals

A seal is an encapsulated (sealed) system of objects and threads of control. The \texttt{JAVASeal} system creates a boundary around each seal, inclusive of its subseals, which defines both a protection domain and a unit of mobility. The activities going on within a seal’s domain are carried out by its strands. Strands are allowed to operate freely on a seal’s objects and class instances, but can not cross seal boundaries. Thus both parent and children are viewed as black boxes. The only operations allowed to cross boundaries are channel sends and receives and they require portals to be open explicitly. An example of a running seal is illustrated in fig. 6.

![Figure 6 A seal contains an object of type Seal, a SealLoader which has a loading policy, three objects (o1, o2, o3) of class C1, a subseal S1, several portals and strands. Portals x, y, z are open for the direct parent, while portals u, v are open for the child S1.](image-url)
Seals are also type environments. Each seal is assigned its class loader at creation. All classes needed by the seal are loaded and instantiated by this class loader. This means that the seal contains its own copy of all the class objects, and therefore of all the class variables of the classes it references. This has several implications for safety and security. First, class static variables are never shared across seals, neither are the locks used in synchronization. Second, the types of the same classes in different seals are different. Both safety and security are improved by removing sharing; an erroneous program may not affect the variables upon which other applications depend, a malicious program may not use shared locks to block other programs, and finally, programs may not use sharing as covert channels to bypass security. The second point means that objects have only meaning in their domains, that, is an object of some class A within a seal SA can not be used within another seal even if this seal also has an instance of class A because their types are different.

JVM and seal.sys classes are loaded by the system loader and imported by the seal’s class loader. This means that a limited form of sharing does exist between seals which has to be controlled for security reasons. (see design note)

The SealLoader class enforces a load policy, that is a set of validity checks that are applied to all class files loaded in the seal. These rules forbid classes to refer to classes of the JDK (except Object and a handful of others). Furthermore, the implementation of the classes is checked to guarantee that no class defines finalizers. The reason for forbidding finalizers is that they may be executed even after the seal terminates and there is no way to prevent them from running. This means that finalizers could easily be used for denial of service attacks.

Seals cannot name each other directly (i.e., using object pointers) for two reasons. First, each seal has its own class loader, and JVM typing rules prevent a class of one loader from referring to a class of another loader. Second, one cannot control the propagation of references in the system; thus, naming seals directly via object references means that seals could name each other and therefore communicate with each other irrespective of the hierarchy constraint that a parent seal must intercept messages. For this reason, seals name each other using an opaque data type called a handle implemented by the Handle class. When a seal is created, it receives a handle for its parent, and the creating seal also obtains a handle for the seals that it creates.

Design note (3)

An earlier design of JAVASeal involved disjoint type environments for seals. In many ways this would have been the cleanest and easiest design. Different seals would have different Object types, and thus even if a reference from a seal leaked into another, the object would be totally opaque and trying to invoke any method on it would cause a type error. Unfortunately, the single root Object type is hardwired throughout the implementation of the JVM, trying to have two different Object types breaks the code and would require extensive modifications. For this reason, the Object class and type are shared by all seals, furthermore all types that are in the closure of Object, i.e. all type directly or indirectly referenced by the class, must be shared.
4.4 Channels and Portals

As mentioned, channels are the basis for communication in JAVASeal, permitting one-way rendez-vous. The base channel class `Chan` contains `send` and `recv` methods that transfer an object of type `Capsule`. A channel has a name and a location. The name is used to identify the channel, the location indicates in which seal it resides.

Both `send` and `recv` operations are blocking; the strands issuing them will be blocked until the communication is allowed to fire.

A communication may fire if there is a pair of matching communication offers. Given two seals $S_1$ and $S_2$, such that either $S_1 = S_2$, $S_1$ is a direct parent of $S_2$ or $S_2$ is a direct parent of $S_1$, two strands $s_1$ and $s_2$ located respectively in $S_1$ and $S_2$ and a channel name $x$, a pair of communication offers is matched iff

1. $s_1$ blocks on a `send` on channel $x$ located in $S_2$ and $s_2$ blocks on a `recv` on channel $x$ located in $S_2$.

2. $s_1$ blocks on a `recv` on channel $x$ located in $S_1$ and $s_2$ blocks on a `send` on channel $x$ located in $S_1$.

3. $s_1$ blocks on a `recv` on channel $x$ located in $S_2$ and $s_2$ blocks on a `send` on channel $x$ located in $S_2$.

4. $s_1$ blocks on a `send` on channel $x$ located in $S_1$ and $s_2$ blocks on a `recv` on channel $x$ located in $S_1$.

An example of channel-based communication in JAVASeal is shown in Fig. 7. Here we assume that the two strands have agreed to exchange a `String` object. The seal method `getSelf` returns the seal handle (type `SHan`) of the seal in which a strand is executing. The seal method `getChild` takes a string and returns the handle of the subseal with matching name. A channel is created with a `String`, the channel name, and a handle, the channel location. The communication offers are located in the same seal.

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

```java
String x = new String("req");
Chan ch = new Chan(x, getSelf());
Caps cp = (Caps) ch.recv();
String s = (String) cp.open();
```

Figure 7 The code on the left tries to send a string object `str` along channel $x$, the code on the right waits on channel $x$ and unpacks the value received into a string.

Note that the rules for matching communication offers require that at least one of the offers be located in the current seal, that is the strand must use a channel whose location is the value returned by `getSelf`. This seal is called the target of the communication. Note that a target may be trying to `send` or `recv`. The other seal is then the source. The communication
involves crossing a seal boundary. In this case it is necessary that the target specifically allows
the exchange to take place. This is done by opening a portal for the channel and source.

We now give an example which involves cross boundary communication. A seal Agent1
is trying to communicate with its parent, using a channel req located in the parent.

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

String x = new String("req");
Chan ch = new Chan(x, getSelf());
Portal.open(x, getChild("Agent1"));
Capsule cp = ch.recv();
String s = (String) cp.open();
```

Figure 8 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.

As mentioned, a portal acts as a control on a communication channel, and must be explicitly
opened by the owning (creating) seal for any communication to take place. In JAVA SEAL, 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.

The rendez-vous one way communication model of channels is the basis of the method
call and return style implemented by subclasses of TypedChan. A channel communication
involves a strand blocking on a send until a strand in the remote seal completes execution
of its recv. For the class Chan, the recv just involves reading the value transmitted. For
typed channels, the recv involves dispatching the invoking strand to execute a method and
transferring a result to the sending strand. This offers an RMI style of programming to seals,
hiding the low-level communication primitives.

Typed channels are used in the following way. To communicate with a seal possessing
the methods m1 and m2, one needs a typed channel class that includes these methods in its
interface. The send of a typed channel class encodes the method name to be called and the
sequence of parameters; the recv type checks the arguments received against the signature of
the requested method, makes the call and returns a result over the channel to the calling strand.

4.5 Capsules

Capsules are used to transfer data across seal boundaries and over the network. The intuition is
that, in order to transfer data across protection domains without fear of aliasing it is necessary
to extend JAVA with a value parameter passing mode. Capsules play this role.
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.

The JAVA SEAL implementation of this abstraction relies on JAVA serialization. Thus the same mechanisms as for RMI are used to control the size of the closure. Note that we do not copy the class object of the objects being serialized.

A capsule may only be opened once. Opening a capsule releases its content in the local environment. 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 version, 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.

The JAVA SEAL system provides subclasses of **Capsule** specialized for RMI style cross-seal method invocation. The **MesCap** contains a method identifier and an argument list.

The **SealCap** is another specialized capsule class. It is designed for transferring seals, either across seal boundaries or across the network. The capsule contains a serialized object graph as well as class archive. The class archive contains a map from all class names (used in the seal) to data structure containing either class fingerprints or full class files. A class fingerprint is a class name and 128 bit checksum of the `.class` file. The idea is that seals will be moved either along with full code closure, or, to reduce bandwidth, with enough information to recreate a compatible environment at receiving site. Versioning issues are a difficult problem in environments with mobile code as many different version of the same classes may have been unleashed on the net. The JAVA type compatibility rules are not really sufficient to ensure that a group of classes can work together (in general, the fact that interfaces match does not suffice to ensure that code taken from different releases of a system can be merged into a single application.) The checksum approach is perhaps too strict in certain situations, but it does guarantee that seals execute in a valid environment.

A **SealCap** is created by serializing seals recursively. To serialize a seal one must first kill its threads and then serialize its state. This is done recursively for children of the seal being serialised: the serialised form of each child is stored in the seal before it is serialised. After movement, a seal is de-serialised and its **run** method is re-executed. The **run** will typically check whether it is alive for the first time or whether it is being re-started. If it is being re-started, it can create new seals or selectively restart the serialised subseals that it contains if this is needed. The point is that not all subseals need to be re-started after a move; a local printer subseal for instance could be replaced after a seal move by a seal furnished by the receiving site.

---

8 This is on top of the checksum of the class interface used by the serialization package for assessing type compatibility.
Design note (4)

The hierarchical serialization of seals mentioned above is performed in-kernel \((\text{serialize in Seal})\). An earlier version of JAVA SEAL implemented hierarchical serialization as a user level protocol: a seal possessed a prepareToSerialize method programmed to selectively serialize selected subseals and clean up the seal state before the move. The advantage of the user-level protocol approach to hierarchical serialization is that it allows seals to clean their state up before the move, and subseals that do not need to be moved can be jettisoned before the transfer rather than being needlessly carried with the transferred seal. On the other hand, the approach can starve the system since there is no guarantee that the prepareToSerialize method will return. Time-outs are a ad-hoc solution. In any case, the cleaning up of a child’s state can just as well be done a posteriori: since a parent is responsible for providing an execution environment for its child seals, it could contain methods commit and prepareForTransaction that a child could use to regularly save data if needed. This approach is closer to the Seal model since the parent seal is being exploited as a control and execution environment. Though the kernel solution implies that the size of data transferred might be greater (since all subseals are transferred), size of capsule is already a problem for JAVA SEAL which has to be addressed in the future.

4.6 Strands

A strand is simply a thread running within a seal. We call it a strand for two reasons. First, it is bound to its seal and can never leave it. Second, a strand’s methods have been modified to involve only information about the strands running within the current seal. Thus for instance, when requesting information about the number of strands, only those that belong to the seal will be listed. Seals are not allowed to load the Thread class since it contains unsafe methods. In the implementation, there is a mapping between threads and strands, but this mapping need not be static or even one-to-one.

A strand is explicitly created when a seal is started and executes the \(\text{run}\) method of that seal. To handle parallelism, strands should be started automatically. For convenience daemon strands are started to services external calls, typically firing off new strands for every incoming channel communication (in practice, the deamon would have a limit on the number of strands, and would manage strands, reusing passive strands when possible).

4.7 Mobility

Recall that mobility in the SEAL calculus is strong mobility enabling seals to be moved transparently. Movement of seals in JAVA SEAL is not so transparent. Since we did not modify the JVM, there is no way to capture process—or thread—state. For this reason, threads (strands) in a JAVA SEAL agent are killed, and only the data reachable from the seal is moved.

The basic mobility model of JAVA SEAL is based on the seal’s \(\text{run}\) method which is executed when a seal is created, and also when a seal is re-started after a move. To move an active seal, one must first close all portals, stop all strands currently within the seal. The seal, at this point, is passive. It can then be stored in a capsule. After movement, the capsule is opened,
and the seal’s run method is invoked. The run will typically check whether the seal is a newborn or whether it is being re-started and load new subseals or selectively restart the serialised subseals, some subseals might not be restarted after every move.

To transfer a seal, a parent sends it as a parameter in a message. The same parameter passing mode applies when transferring seals, they are moved by value inside SealCap objects.

**Design note (5)**

*Strong mobility requires saving the execution state of running threads. The difficulty from an implementor’s viewpoint is that JVM stacks are untyped, so the byte-code implementation must be rewritten to tag operands on stacks with their types, as was done in [2] at a serious performance cost. A more fundamental problem is the use of C and machine dependent code to implement byte-code instructions and certain primitive operations. A JAVA thread has two stacks: the JAVA stack and an internal stack used when executing within C code. To maintain hardware independence, mobility would have to be restricted to byte-code boundaries. In this way, only the JAVA stack needs to be saved. In Sumatra this was achieved by adding a go instruction, threads were allowed to move only on go instructions, at this point the internal stack was guaranteed to be empty. The Sumatra approach cannot be adopted in JAVASeal since we want to move groups of threads (all threads within a seal) irrespective of their execution state. This leads to problems, especially with respect to locking. Monitors are implemented in machine dependent code. Thus the real execution state of a thread is not reflected in its JAVA stack — no information on locks held by this (or any) thread is found there. To correctly capture locking it must be moved to the machine independent part of the JVM.*

### 4.8 The Security Model

The basis of the JAVASeal security model is an implementation of the principles mentioned in section 3.2: protection domains, total mediation and accountability.

The protection domain in JAVASeal is the seal, containing the main seal object and all objects created by that seal. By assigning each seal its own class loader, the JVM guarantees that aliasing between agent objects does not occur. This is the major part of ensuring isolation. The only other form of object sharing that can occur between seals is through the shared variables of the seal.sys and JVM classes. It is to control the use of these variables as covert channels that we restrict the set of classes that a seal can load.

Total mediation is enforced by the use of portals and channels, as well as through the hierarchy. All effective communication between two seals must be over a channel. Further, a typed channel enforces a type-check on the parameters transferred in a method call, guaranteeing that an object serialized and transferred is deserialised with the same type. The channel also has a portal linked to it, allowing a seal to refuse calls from the sending seal by simply closing the portal. In addition, a seal can program a security policy that is linked to the portal — this is a Java program that is evaluated on each message sent over the channel. The role of this policy is to enable the seal programmer to specify a set of access constraints that must be met before the calling seal may have its method call serviced [5].
The second important feature of mediation is the hierarchal model. The property that a parent seal intercepts a message exchanged between children is crucial for enforcing a mandatory or application wide security policy on a group of agents. For instance, given the control of covert channels that exist, it becomes possible for the root seal to allocate high-level and low-level agents to enforce multi-level security.

Accountability — being able to identify what seal is responsible for which actions — follows to some extent from the JAVA SEAL design. Actions relate to resources used: memory, cycles and services. Memory usage is hard to monitor since it involves controlling, not only, the use of the new command but also of automatic memory management. However, the creation of new seals can be controlled. Cycles are easier to monitor in the sense that each seal has its own strands that do not traverse the seal boundary — each strand is mapped to a Java thread and so a strand can be neutralized by killing the associated thread. Finally service usage is the most easily accounted for since access to the service is intercepted by the root seal or the application’s mains seal.

The key to security in the Seal model is that a parent seal can act as a secure environment for its children. Since a parent intercepts the messages exchanged between children, it can enforce security constraints on these exchanges. For instance, it can forbid a communication or sanitise the parameters in the message. A further aspect to JAVA SEAL security is its reliance on Java language-level verification. This is used to reason about the absence of covert channels and can also be used to reason about security since the policy that a seal uses to control messages exchanged between children seals is expressed in Java code. Any properties proven about this policy depend therefore on Java language semantics. The JAVA SEAL system is very open in this sense: though a rigid hierarchy exists, any security policy can be inserted. More details about implementing security policies in Java can be found in the position paper [5].

4.9 Service Infrastructure

JAVA SEAL applications require services such as a network interface or a graphical user interface, but this leads to a contradiction. On the one hand, these services must be built on top of the corresponding JDK classes; on the other hand, these classes contain global variables and covert channels, exactly the reason why we went to the trouble of implementing JAVA SEAL in the first place.

The pragmatic solution to this problem is to provide a minimal interface to these services, to check carefully their implementation, and to add new services sparingly. The interfaces are minimalised in two ways. First, the loading restrictions mean that a user seal shares a minimum set of classes with service seals. Second, service seals are also structured as seals - loading restrictions can be imposed upon them and they communicate with each other and with user seals over channels. This means that covert channels between user and service seals are minimised, and that messages exchanged can be monitored by a security policy in RootSeal.

The remaining security problem for services is controlling the exchange of information between service seals via the shared variables of their classes, since service seals have access to a larger set of classes. This calls for selective allocation of JDK classes between service seals,
and for use of replicated rather than shared JDK classes in service seals when this is possible.

One of the goals for JAVASeal is to structure services as seals. The principal advantage of
this uniform system structuring is that we can exploit the security architecture further, that is,
calls to and from services can be mediated via portals, and one can control the set of classes
visible to a service seal.

The security risks created by services can be limited by using different loader and package
spaces for services and for user seals. Service seals (from the package seal.service) may
use a wider set of classes loaded by the system loader; service seals thus effectively run in
the system loader space. User seals that wish to interact with service seals must use the JAVASeal
channel based communication mechanism.

From the point of view of security, a user seal only shares a minimum set of JVM
classes with a service seal — those that its class loader permit it to load. Further, RootSeal
intercepts calls to services and so can enforce security controls on service use.

Service seals such as NetMgr (network interface) and SpinGui (Html user interface) are
immobile. They are structured as a seal.service seal simply to benefit from the JAVASeal
mediation and class loading facilities. In any case, it makes little sense to move such seals –
their role is to provide local servicing for user seals that arrive at that site.

5 Related Work

Wallach et al. discuss three software protection mechanisms for JAVA: capabilities, stack in-
trospection and controlled name spaces [14]. The failings of these approaches have been men-
tioned in section 3.2 and in [13].

Mole is an academic mobile agent system built as a JAVA package [4] which was used in
an earlier stage of the ASAP project. A Mole system is made up of agents and places. An
agent is moved (weakly) with its code and state and state between places. Agent-agent uses
message passing for flexibility, other form of communication include RPC (!) and anonymous
events. Agent IDs uniquely name agents; badges that can be pinned on and off give application
level semantics to names (these are logical predicates). The services provided by Mole can be
emulated over JAVASeal. The main difference between Mole and JAVASeal is the attention to
security policy expression and enforcement. It is important to stress that we are aiming to build
secure agents and not just protected agents. The difference is important. A system such as Mole
seeks protection by using capabilities (badges) the idea is that an agent be able to verify and
control who calls it. What we want to do is to impose a restriction on two seals communicating,
even if they want to. This is what makes covert channel analysis important, and is not generally
considered in other papers.

The Aglets system is a similar attempt to add mobility to Java [8]. An agent here is a thread
and applet. An agent can move itself at will and responds to event messages from the environ-
ment. Agents execute within a “context” — a form of security domain: domains can enforce
policies controlling how agents use local resources. The security model of Aglets is based on
principals: these can be aglets, aglet owners, contexts, context owners. An authorisation lan-
language has been developed that allows expressive security policies controlling principal use of
resources. As is the case with Mole, we believe that real security domains implies restricting
the classes — and static variables — that seals can share. No sharing restrictions are enforced in
the Aglet system. Further, the Aglet agent model may be emulated in JAVA SEAL by program a
seal class that emulates the event model of Aglet and to include some simple mobility services.

Sumatra is a modified JVM allowing for thread mobility [2]. The goal of Sumatra was
to make programs network-aware (to monitor resource availability of environment) and agile
(to react to changes in environment and to control the way resources are used). For agility,
Sumatra employs a go instruction, which tells the executing thread to move itself. Thread
mobility depends on typed stacks and class-loaders that lazily demand the code for migrated
objects. Sumatra seems to be the only strongly mobile JAVA-based system.

The next version of 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.

6 Conclusions

This paper has reviewed the status of the JAVA SEAL system which was designed to support
secure mobile programming over the Internet. JAVA SEAL is based on a formalism, the SEAL
calculus, but due to implementation constraints only a subset supporting weak mobility was
implemented.

The model presented in this paper represents a complete redesign of the JVM model. The
implementation of the abstractions presented here is under way at the time of this writing.

Work is continuing on performance optimization of the system, currently the implementa-
tion of inter-seal communication represents a serious bottle neck for interaction intensive
applications. The techniques that are being investigated are analogous to fast remote procedure
call implementation techniques as used in operating systems research. On another track we
are working on techniques for reducing the cost of code migration by novel data compression
techniques.

In parallel, work on implementing services to support the HYPERNEWS project and a forth-
coming SPP project, DNX which concerns Internet domain name registration, is on going.

References

the Fourth ACM Conference on Computer and Communications Security, Zürich, April 1997. ACM Press,


