Extensible Security Architectures for Java

DAN S. WALLACH
Princeton University
DIRK BALFANZ
Princeton University
DREW DEAN
Princeton University
EDWARD W. FELTEN
Princeton University

Mobile code technologies such as Java, JavaScript, and ActiveX generally limit all programs to a single restrictive security policy. However, software-based protection can allow for more extensible security models, with potentially significant performance improvements over traditional hardware-based solutions. An extensible security system should be able to protect subsystems and implement policies that are created after the initial system is shipped. We describe and analyze three implementation strategies for interposing such security policies in software-based security systems. Implementations exist for all three strategies: several vendors have adapted capabilities to Java, Netscape and Microsoft have extensions to Java's stack introspection, and we built a name space management system as an add-on to Microsoft Internet Explorer. Theoretically, all these systems are equivalently secure, but many practical issues and implementation details favor some aspects of each system.

Categories and Subject Descriptors: []:

General Terms: Java Security

Additional Key Words and Phrases: Capabilities, Internet, World Wide Web

---

1. INTRODUCTION

As the World Wide Web has been used to build increasingly complex applications, developers have been constrained by the Web's static document model. "Active" content can add simple animations to a page, but it can also transform the Web into a "platform" for writing and distributing programs. A variety of *mobile code* systems such as Java [Gosling et al. 1996], JavaScript [Flanagan 1997], ActiveX [Microsoft Corporation 1996], and Shockwave [Schmitt 1997] make this possible.

Users and developers love mobile code, but it raises serious security concerns. Software distribution over the Internet has been common for years, but the risks are greatly amplified with Web plug-ins and applets by virtue of their ubiquity and seamless integration. Users are often not even aware of mobile code's presence. Mobile code systems must have correspondingly stronger security to compensate for the increased exposure to potentially

---

To appear in the *16th Symposium on Operating Systems Principles*, October 1997, Saint-Malo, France.
Authors' e-mail addresses: {dwallach,balfanz,ddean,felten}@cs.princeton.edu
Our work is supported by donations from Sun Microsystems, Bellcore, and Microsoft. Dan Wallach did portions of this work at Netscape Communications Corp. Edward Felten is supported in part by an NSF National Young Investigator award.

hostile code.

This paper considers the problem of securely supporting mobile code on real-world systems. Unlike traditional operating systems, Web browsers must rely on software mechanisms for basic memory safety, both for portability and performance. Currently, there is no standard for constructing secure services above basic memory safety primitives. We explain three different strategies and their implementations in Java: several vendors [Goldstein 1996; Electric Communities 1996] have built capability systems, Netscape and Microsoft have extensions to Java's stack introspection, and we designed an add-on to Microsoft Internet Explorer which hides or replaces Java classes. We analyze these systems in terms of established security criteria and conclude with a discussion of appropriate environments in which to deploy each strategy.

1.1 The Advantages of Software Protection

Historically, memory protection and privilege levels have been implemented in hardware: memory protection via base / limit registers, segments, or pages; and privilege levels via user / kernel mode bits or rings [Schroeder and Saltzer 1972]. Recent mobile code systems, however, rely on software rather than hardware for protection. The switch to software mechanisms is being driven by two needs: portability and performance.

1.1.0.1 *Portability.* The first argument for software protection is portability. A user-level software product like a browser must coexist with a variety of operating systems. For a Web browser to use hardware protection, the operating system would have to provide access to the page tables and system calls, but such mechanisms are not available universally across platforms. Software protection allows a browser to have platform-independent security mechanisms.

1.1.0.2 *Performance.* Second, software protection offers significantly cheaper cross-domain calls[1]. To estimate the magnitude of the performance difference, we did two simple experiments. The results below should not be considered highly accurate, since they mix measurements from different architectures. However, the effect we are measuring is so large that these small inaccuracies do not affect our conclusions.

First, we measured the performance of a null call between Common Object Model (COM) objects on a 180 MHz PentiumPro PC running Windows NT 4.0. When the two objects are in different tasks, the call takes 230 μsec; when the called object is in a dynamically linked library in the same task, the call takes only 0.2 μsec — a factor of 1000 difference. While COM is a very different system from Java, the performance disparity exhibited in COM would likely also appear if hardware protection were applied to Java. This ratio appears to be growing even larger in newer processors [Ousterhout 1990; Anderson et al. 1991].

The time difference would be acceptable if cross-domain calls were very rare. But modern software structures, especially in mobile code systems, are leading to tighter binding between domains.

To illustrate this fact, we then instrumented the Sun Java Virtual Machine (JVM) (JDK 1.0.2 interpreter running on a 167 MHz UltraSparc) to measure the number of calls across trust boundaries, that is, the number of procedure calls in which the caller is either

[1] In Unix, a system-call crosses domains between user and kernel processes. In Java, a method call between applet and system classes also crosses domains because system classes have additional privileges.

Workload	time	crossings	crossings/sec
CaffeineMark	40.9	1135975	26235
SunExamples	138.7	5483828	35637

Fig. 1. Number of trust-boundary crossings for two Java workloads. *Time* is the total user+system CPU time used, in seconds. *Crossings* is the number of method calls which cross a trust boundary. The last column is their ratio: the number of boundary crossings per CPU-second.

more trusted or less trusted than the callee. We measured two workloads. CaffeineMark[2] is a widely used Java performance benchmark, and SunExamples is the sum over 29 of the 35 programs[3] on Sun's sample Java programs page[4]. (Some of Sun's programs run forever; we cut these off after 30 seconds.) Note that our measurements almost certainly underestimate the rate of boundary crossings that would be seen on a more current JVM implementation that used a just-in-time (JIT) compiler to execute much more quickly.

Figure 1 shows the results. Both workloads show roughly 30000 boundary crossings per CPU-second. Using our measured cost of 0.2 μsec for a local COM call, we estimate that these programs spend roughly 0.5% of their time crossing trust boundaries in the Java system with software protection (although the actual cost in Java may be significantly cheaper).

Using our measured cost of 230 μsec for a cross-task COM call, we can estimate the cost of boundary crossings for both workloads in a hypothetical Java implementation that used hardware protection. The cost is enormous: for CaffeineMark, about 6 seconds per useful CPU-second of work, and for SunExamples, about 8 seconds per useful CPU-second. In other words, a naïve implementation of hardware protection would slow these programs down by a factor of 7 to 9. This is unacceptable.

Why are there so many boundary crossings? Surely the designers did not strive to increase the number of crossings; most likely they simply ignored the issue, knowing that the cost of crossing a Java protection boundary was no more than a normal function call. What our experiments show, then, is that the *natural* structure of a mobile code system leads to very frequent boundary crossings.

Of course, systems using hardware protection are structured differently; the developers do what is necessary to improve performance (*i.e.*, buffering calls together). In other words, they deviate from the natural structure in order to artificially reduce the number of boundary crossings. At best, this neutralizes the performance penalty of hardware protection at a cost in both development time and the usefulness and maintainability of the resulting system. At worst, the natural structure is sacrificed and a significant performance gap still exists.

1.2 Memory Protection vs. Secure Services

Most discussions of software protection in the OS community focus on *memory protection*: guaranteeing a program will not access memory or execute code for which it is not authorized. Software fault isolation [Wahbe et al. 1993], proof-carrying code [Necula and Lee 1996], and type-safe languages are three popular ways to ensure memory protection.

---

[2] `http://www.webfayre.com/pendragon/cm2/`
[3] We omitted internationalization examples and JDK 1.1 applets.
[4] `http://www.javasoft.com/applets/applets.html`

1.2.1 *Software Memory Protection.* Wahbe et al. [1993] introduced *software fault isolation*. They showed they could rewrite load, store, and branch instructions to validate all memory access with an aggregate 5–30% slowdown. By eliminating these checks where possible, and optimizing the placement of the remaining checks, a perhaps acceptable slowdown is achieved in a software-only implementation. The theory is still the same as hardware-based protection: potentially dangerous operations are dynamically checked for safety before their execution.

Necula and Lee [1996] introduced *proof-carrying code*. Proof-carrying code eliminates the slowdown associated with software fault isolation by statically verifying a proof that a program respects an agreed upon security policy when the program is loaded. After the proof is verified, the program can run at full speed. Few limits are put on the form of the program; it can be hand-tuned assembly language. The major difficulty at the moment is generating the proofs: for now, they are written by hand with the help of a non-interactive theorem prover. Research is underway to automate proof generation. One promising idea is to have the compiler for a safe language generate a proof of memory safety during the compilation process.

The language-based protection model, exemplified by Java, can be traced back at least to the Burroughs B-5000 in 1961 [Levy 1984] and is also used in current research operating systems such as SPIN [Bershad et al. 1995]. Memory accesses can be controlled with a safe language [Milner and Tofte 1991]. If the language has a mechanism for enforcing abstraction boundaries (in Java, this means ensuring that `private` variables and methods are not accessible outside their class), then the system can force control to flow through security checks before executing any dangerous code. Language-based safety can use either dynamic type checking [Rees 1996; Borenstein 1994] or static type checking (as is mostly the case in Java [Gosling et al. 1996; Drossopoulou and Eisenbach 1997]). The major tradeoff is performance versus theoretical complexity. The static system should be faster, because it only checks safety once, before the program starts. Dynamic checking is simpler, because there is no need to prove a type soundness theorem: at each step during execution, a local check is sufficient to ensure the absence of type errors. Note that while static type checking may be theoretically complex, the implementation can be quite simple.

1.2.2 *Secure Services.* While memory protection is necessary and important, there is much more to security than memory protection. Memory protection is sufficient for programs that do calculations without invoking system services, but more interesting programs use a rich set of operating system services such as shared files, graphics, authentication, and networking. The operating system code implementing these services must define and correctly enforce its own security policy pertaining to the resources it is managing. For example, the GUI code must define and enforce a policy saying which code can observe and generate which GUI events. Memory protection can keep hostile code from directly reading the GUI state, but it cannot keep a hostile program from tricking the GUI code into telling it about events it shouldn't see[5].

Seltzer et al. [1996] studied some of the security problems involved in creating an extensible operating system. They argue that memory protection is only part of the solution; the bulk of their paper is concerned with questions of how to provide secure services. Consider

[5]GUI event manipulation may seem harmless, but observing GUI events could allow an attacker to see passwords or other sensitive information while they are typed. Generated GUI events would allow an attacker to create keystrokes to execute dangerous commands.

the security flaws that have been found in Unix. Very few are related to hardware memory protection: almost all of the flaws have been in trusted services such as `sendmail` and `fingerd`. Perfect memory protection would not prevent these flaws.

The challenge, then, is not only getting memory protection but providing secure system services. This paper considers how secure services can be built solely with software protection.

2. SECURITY IN JAVA

Though we could in principle use any of several mobile code technologies, we will base our analysis on the properties of Java. Java is a good choice for several reasons: it is widely used and analyzed in real systems, and full source code is available to study and modify.

Java uses programming language mechanisms to enforce memory safety. The JVM enforces the Java language's type safety, preventing programs from accessing memory or calling methods without authorization [Lindholm and Yellin 1996]. Existing JVM implementations also enforce a simple "sandbox" security model which prohibits untrusted code from using any sensitive system services.

The sandbox model is easy to understand, but it prevents many kinds of useful programs from being written. All file system access is forbidden, and network access is only allowed to the host where the applet originated. While untrusted applets are successfully prevented from stealing or destroying users' files or snooping around their networks, it is also impossible to write a replacement for the users' local word processor or other common tools which rely on more general networking and file system access.

Traditional security in Java has focused on two separate, fixed security policies. Local code, loaded from specific directories on the same machine as the JVM, is completely trusted. Remote code, loaded across a network connection from an arbitrary source, is completely untrusted.

Since local code and remote code can co-exist in the same JVM, and can in fact call each other, the system needs a way to determine if a sensitive call, such as a network or file system access, is executing "locally" or "remotely." Traditional JVMs have two inherent properties used to make these checks:

—Every class in the JVM which came from the network was loaded by a ClassLoader, and includes a reference to its ClassLoader. Classes which came from the local file system have a special system ClassLoader. Thus, local classes can be distinguished from remote classes by their ClassLoader.

—Every frame on the call stack includes a reference to the class running in that frame. Many language features, such as the default exception handler, use these stack frame annotations for debugging and diagnostics.

Combined together, these two JVM implementation properties allow the security system to search for remote code on the call stack. If a ClassLoader other than the special system ClassLoader exists on the call stack, then a policy for untrusted remote code is applied. Otherwise, a policy for trusted local code is used.

To enforce these policies, all the potentially dangerous methods in the system were designed to call a centralized SecurityManager class which checks if the action requested is allowed (using the mechanism described above), and throws an exception if remote code is found on the call stack. The SecurityManager is meant to implement a reference monitor [Lampson 1971; National Computer Security Center 1985] — always invoked, tamperproof, and easily verifiable for correctness.

In practice, this design proved insufficient. First, when an application written in Java (*e.g.*, the HotJava Web browser) wishes to run applets within itself, the low-level file system and networking code has a problem distinguishing between direct calls from an applet and system functions being safely run on behalf of an applet. Sun's JDK 1.0 and JDK 1.1 included specific hacks to support this with hard-coded "ClassLoader depths" (measuring the number of stack frames between the low-level system code and the applet code).

In addition to a number of security-related bugs in the first implementations [Dean et al. 1996], many developers complained that the *sandbox policy*, applied equally to all applets, was too inflexible to implement many desirable "real" applications. The systems presented here can all distinguish between different "sources" of programs and provide appropriate policies for each of them.

3. APPROACHES

We now present three different strategies for resolving the inflexibility of the Java sandbox model. All three strategies assume the presence of digital signatures to identify what *principal* is responsible for the program. This principal is mapped to a security policy. After that, we have identified three different ways to enforce the policy:

Capabilities. A number of traditional operating systems were based on unforgeable pointers which could be safely given to user code. Java provides a perfect environment for implementing capabilities.

Extended stack introspection. The current Java method of inspecting the stack for unprivileged code can be extended to include principals on the call stack.

Name space management. An interesting property of dynamic loading is the ability to create an environment where different applets see different classes with the same names. By restricting an applet's name space, we can limit its activities.

In this section, we will focus on how each method implements interposition of protective code between potentially dangerous primitives and untrusted code. In section 4, we will compare these systems against a number of security-relevant criteria.

3.1 Common Underpinnings

Security mechanisms can be defined by how they implement *interposition*: the ability to protect a component by routing all calls to it through a reference monitor. The reference monitor either rejects each call, or passes it through to the protected component; the reference monitor can use whatever policy it likes to make the decision. Since a wide variety of policies may be desirable, it would be nice if the reference monitor were structured to be extensible without being rewritten. As new subsystems need to be protected, they should be easy to add to the reference monitor.

Traditionally, extensible reference monitors are built using *trusted subsystems*. For example, the Unix password file is read-only, and only a setuid program can edit it. In a system with domain and type enforcement [Badger et al. 1995], the password database would have a type that is only editable by programs in the appropriate domain. In either case, the operating system has no *a priori* knowledge of the password database or password-changing utilities, but instead has security mechanisms general enough to protect the data by only allowing a small number of programs access to it. Clark and Wilson

[1987] offers a particularly cogent argument in favor of the use of trusted subsystems and abstraction boundaries in the security of commercial systems.

3.1.1 *Digital Signatures as Principals.* To implement any security policy, Java needs a notion of *principal*. In a traditional operating system, every user is a principal[6]. For Java, we wish to perform access control based on the "source" of the code, rather than who is running it. This is solved by digitally signing the applet code. These signatures represent *endorsements* of the code by the principal who signed it[7] asserting that the code is not malicious and behaves as advertised. When a signature is verified, the principal may be attached to the *class* object within the JVM. Any code may later query a class for its principal. This allows a group of classes meant to cooperate together to query each others' principal and verify that nothing was corrupted.

There is no reason a program cannot have multiple signatures, and hence multiple principals. This means we must be able to combine potentially conflicting permissions granted to each principal, much as a traditional operating system must resolve permissions when a user belongs to multiple groups. One way to solve this problem is to choose a *dominating principal*, generally the principal about whom the user has the strongest feelings, and treat the program as though it were signed only by the dominating principal. Some systems also define an algebra for combining policies.

3.1.2 *Policies and Users.* Each system needs a *policy engine* which can store security policy decisions on stable storage, answer questions about the current policy, and query the user when an immediate decision is necessary. The details of what is stored, and what form user queries take, depend on the specific abstractions defined by each system.

To clarify the policy engine's role, consider the file system protection mechanisms in Unix. The "policy decisions" in a Unix file system are the file permission bits for each directory and file; these are stored on disk. The role of policy engine is played by code in the kernel that maintains the permission bits and uses them to decide which file access requests to allow.

In Java, a critical issue with the policy engine is how to help non-technical users make security-relevant decisions about who they trust to access particular resources. To simplify the user interface, we want to pre-define groups of common privileges and given them user-friendly names. For example, a "typical game privileges" group might refer to specific limited file system access, full-screen graphics, and network access to the game server. Such groupings allow a single dialog box to be presented to a user which does not burden the user with a long series of individual privilege decisions and their corresponding dialog boxes.

3.1.3 *Site Administration.* Another way to remove complexity from users is to move the work to their system administrators. Many organizations prefer to centrally administrate their security policy to prevent users from accidentally or maliciously violating the policy.

These organizations need hooks into the Web browser's policy mechanism to either pre-install and "lock down" all security choices or at least to pre-approve applications used

[6] *Principal* and *target*, as used in this paper, are the same as *subject* and *object*, as used in the security literature, but are more clear for discussing security in object-oriented systems.

[7] Because a signature can be stripped or replaced by a third-party, there is no strong way for a signature to guarantee *authorship*.

by the organization. If an organization purchases a new product, all users should not be burdened with dialogs asking them to grant it privileges. Likewise, if a Web site is known to be malicious, an administrator could block it from ever asking any user for a privilege.

Both Netscape and Microsoft have extensive support for centralized policy administration in their Web browsers. While malicious users may not necessarily be prevented from reinstalling their Web browser (or operating system) to override the centralized security policies, normal users can at least benefit from their site administrators' work to pre-determine which applications should and should not be trusted.

3.2 First Approach: Capabilities

In many respects, Java provides an ideal environment to build a traditional capability system [Fabry 1974; Levy 1984]. Electric Communities [1996] and Goldstein [1996][8] have implemented such systems. This section discusses general issues for capabilities in Java, rather than specifics of the Electric Communities or JavaSoft systems.

Dating back to the 1960's, hardware and software-based capability systems have often been seen as a good way to structure a secure operating system [Hardy 1985; Neumann et al. 1980; Tanenbaum et al. 1986]. Fundamentally, a *capability* is an unforgeable pointer to a controlled system resource. To use a capability, a program must have been first explicitly given that capability, either as part of its initialization or as the result of calling another capability. Once a capability has been given to a program, the program may then use the capability as much as it wishes and (in some systems) may even pass the capability to other programs. This leads to a basic property of capabilities: any program which *has* a capability *must* have been permitted to use it[9].

3.2.1 *Capabilities in Java.* In early machines, capabilities were stored in tagged memory. A user program could load, store, and execute capabilities, but only the kernel could create a capability [Levy 1984]. In Java, a capability is simply a reference to an object. Java's type safety prevents object references from being forged. It likewise blocks access to methods or member variables which are not labeled `public`.

The current Java class libraries already use a capability-style interface to represent open files and network connections. However, static method calls are used to acquire these capabilities. In a more strongly capability-based system, all system resources (including the ability to open a file in the first place) would be represented by capabilities. In such a system, an applet's top-level class would be passed an array of capabilities when initialized. In a flexible security model, the system would evaluate its security policy before starting the applet, then pass it the capabilities for whatever resources it was allowed. If an applet is to be denied all file system access, for example, it need only not receive a file system capability. Alternately, a centralized "broker" could give capabilities upon request, as a function of the caller's identity.

3.2.2 *Interposition.* Java capabilities would implement interposition by providing the exclusive interface to system resources. Rather than using the `File` class, or the public constructor of `FileInputStream`, a program would be required to use a file system

[8] The Java Electronic Commerce Framework (JECF) uses a capability-style interface, extending the signed applet support in JDK 1.1. More information about JavaSoft's security architecture plans can be found in Gong [1997a].

[9] For example, the combination to open a safe represents a capability. The safe has no way to verify if the combination has been stolen; any person entering the correct combination can open the door. The security of the safe depends upon the combination not being leaked by an authorized holder.

```
// In this example, any code wishing to open a file must first obtain
// an object which implements FileSystem.

interface FileSystem {
    public FileInputStream getInputStream(String path);
}

// This is the primitive class for accessing the file system. Note that
// the constructor is not public -- this restricts creation of these
// objects to other code in the same package.

public class FS implements FileSystem {
    FS() {}

    public FileInputStream getInputStream(String path) {
        return internalOpen(path);
    }

    private native FileInputStream internalOpen(path);
}

// This class allows anyone holding a FileSystem to export a subset of
// that file system.  Calls to getInputStream() are prepended with the
// desired file system root, then passed on to the internal FileSystem
// capability.

public class SubFS implements FileSystem {
    private FileSystem fs;
    private String rootPath;

    public SubFS(String rootPath, FileSystem fs) {
        this.rootPath = rootPath;
        this.fs = fs;
    }

    public FileInputStream getInputStream(String path) {
        // to work safely, this would need to properly handle `..' in path
        return fs.getInputStream(rootPath + "/" +  path);
    }
}
```

Fig. 2. Interposition of a restricted file system root in a capability-based Java system. Both `FS` and `SubFS` implement `FileSystem`, the interface exported to all code which reads files.

capability which it acquired on startup or through a capability broker. If a program did not receive such a capability, it would have no other way to open a file.

Since a capability is just a reference to a Java object, the object can implement its own security policy by checking arguments before passing them to its private, internal methods. In fact, one capability could contain a reference to another capability inside itself. As long as both objects implement the same Java interface, the capabilities could be indistinguishable from one another.

For example, imagine we wish to provide access to a subset of the file system — only files below a given subdirectory. One possible implementation is presented in figure 2. A

`SubFS` represents a capability to access a subtree of the file system. Hidden inside each `SubFS` is a `FileSystem` capability; the `SubFS` prepends a fixed string to all pathnames before accessing the hidden `FileSystem`. Any code which already possesses a handle to a `FileSystem` can create a `SubFS`, which can then be passed to an untrusted subsystem. Note that a `SubFS` can also wrap another `SubFS`, since `SubFS` implements the same interface as `FS`.

Also note that with the current Java class libraries, a program wishing to open a file can directly construct its own `FileInputStream`. To prevent this, either `FileInputStream` must have non-`public` constructors, or the `FileInputStream` *class* must be hidden from programs. This restriction would also apply to other file-related Java classes, such as `RandomAccessFile`. While the Java language can support capabilities in a straightforward manner, the Java runtime libraries (and all code depending on them) would require significant changes.

3.3 Second Approach: Extended Stack Introspection

This section presents an extension to Java's current stack-introspection mechanism[10]. This approach is taken by both Netscape[11] and Microsoft[12] in version 4.0 of their respective browsers. JavaSoft is working on a similar design [Gong 1997b]. Although the designs of Netscape and Microsoft differ in many ways, there is a core of similarity. Accordingly, we will first describe the common elements and then discuss the differences.

As in the other approaches, extended stack introspection uses digital signatures to match pieces of incoming byte code to principals, and a policy engine is consulted to determine which code has permission for which targets.

3.3.1 *Basic Design.* Three fundamental primitives are necessary to use extended stack introspection:

—`enablePrivilege(`*target*`)`

—`disablePrivilege(`*target*`)`

—`checkPrivilege(`*target*`)`

When a dangerous resource (such as the file system) needs to be protected, two steps are necessary: a target must be defined for the resource, and the system must call `checkPrivilege()` on the target before accessing the protected resource.

When code wishes to *use* the protected resource, it must first call `enablePrivilege()` on the target associated with that resource. This will consult the policy engine to see whether the principal of the caller is permitted to use the resource. If permitted, it will create an *enabled privilege*. After accessing the resource, the code will call `disablePrivilege()` to discard the enabled privilege.

`checkPrivilege(`*target*`)` searches for an enabled privilege to the given target. If one is not found, an exception is thrown.

3.3.2 *Improved Design.* The design as described so far is simple, but it requires a few refinements to make it practical. First, it is clear that enabled privileges must apply only to

[10]This approach is sometimes incorrectly referred to as "capability-based security" in some marketing literature.

[11]`http://developer.netscape.com/library/documentation/signedobj/`

[12]`http://www.microsoft.com/ie/ie40/browser/security/sandbox.htm`

the thread that created them. Otherwise, an unfortunately timed thread switch could leak privileges to untrusted code.

Second, the basic design is flawed in that an enabled privilege could live forever if the programmer neglected to disable it on all paths out of the method that created it. This error is easy to make, especially in the presence of exceptions, so the design is refined to attach enabled privileges to a hidden and protected field of the stack frame of the method that created them. This would cause the privileges to be discarded automatically when the method that created them exits.

The design has one more serious weakness: it is subject to *luring attacks* in which trusted code that has some privileges enabled is tricked into calling into untrusted code. This would have the effect of delegating the enabled privileges to the untrusted code, which could exploit them to do damage. Calls from trusted to untrusted code are common, for example in callbacks associated with Java's system library functions such as the Abstract Windowing Toolkit.

The solution to luring attacks is to use a more restricted stack-frame searching algorithm in `checkPrivilege()`. This algorithm, which both Netscape and Microsoft use, is shown in figure 3. The algorithm searches the frames on the caller's stack in sequence, from newest to oldest. The search terminates, allowing access, upon finding a stack frame that has an appropriate enabled privilege. The search terminates, forbidding access (and throwing an exception), upon finding a stack frame which is forbidden by the policy engine from accessing the target.

This neatly eliminates luring attacks: untrusted code cannot exploit the privileges of its caller, because the stack search will terminate upon finding the untrusted stack frame. A callee can sometimes use the privileges of its caller, but only those privileges that the callee could have requested for itself.

We note that Netscape and Microsoft take different actions when the search reaches the end of the stack uneventfully: Netscape denies permission and Microsoft allows it. The Netscape approach follows the principle of least privilege, since it requires that privileges be explicitly enabled before they can be used. The Microsoft approach, on the other hand, may be easier for developers, since no calls to `enablePrivilege()` are required in the common case where independent untrusted or semi-trusted applets are using more-trusted system libraries. It also allows local, trusted Java applications to use the same JVM without modification: they run as a trusted principal so all of their accesses are allowed by default[13].

3.3.3 *Example: Creating a Trusted Subsystem.* A trusted subsystem can be implemented as a class that enables some privileges before calling a system method to access the protected resource. Untrusted code would be unable to directly access the protected resource, since it would be unable to create the necessary enabled privilege. Figure 4 demonstrates how code for a trusted subsystem may be written.

Several things are notable about this example. The classes in figure 4 do not need to be signed by the system principal (the all-powerful principal whose privileges are hard-wired into the JVM). They could be signed by any trusted principal. This allows third parties to gradually extend the security of the JVM without opening the entire system to attack.

[13]Netscape's stack introspection is currently only used in their Web browser, so compatibility with existing Java applications is not an issue.

```
checkPrivilege(Target target) {
    // loop, newest to oldest stack frame
    foreach stackFrame {
        if(stackFrame has enabled privilege for target) return Allow;

        if(policy engine forbids access to target by class executing
           in stackFrame) return Forbid;
     }

    // if we get here, we fell off the end of the stack
    if(Netscape)    return Forbid;
    if(Microsoft)   return Allow;
}
```

Fig. 3. The stack walking algorithm used by Netscape and Microsoft.

With the trusted subsystem, an applet has two ways to open a file: call through `TrustedService`, which will restrict it to a subset of the file system, or request `UniversalFileRead` privileges for itself. There are no other ways to enable privileges for the `UniversalFileRead` target, so there are no other ways to open a file. Note that, even should the applet try to create an instance of `FS` and then call `getInputStream()` directly, the low-level file system call (inside `java.io.FileInputStream`) will still fail.

3.3.4 *Details.* The Netscape and Microsoft implementations have many additional features beyond those described above. We will attempt to describe a few of these enhancements and features here.

3.3.4.1 *"Smart Targets".* Sometimes a security decision depends not only on which resource (the file system, network, etc.) is being accessed, but on which specific part of the resource is involved, for example, on exactly which file is being accessed. Since there are too many files to create targets for each one, both Microsoft and Netscape have a form of "smart targets" which have internal parameters and can be queried dynamically for access decisions. Details are beyond the scope of this paper.

3.3.4.2 *Who can define targets?.* The Netscape and Microsoft systems both offer a set of predefined targets that represent resources that the JVM implementation wants to protect; they also offer ways to define new targets. The Microsoft system allows only fully trusted code to define new targets, while the Netscape system allows anyone to define new targets. In the Netscape system, targets are named with a (*principal*, *string*) pair, and the system requires that the *principal* field match the principal who signed the code that created the target. Built-in targets belong to the predefined principal *System*.

Allowing anyone to define new targets, as Netscape does, allows third-party library developers to define their own protected resources and use the system's stack-introspection mechanisms to protect them. The drawback is that users may be asked questions regarding targets that the Netscape designers did not know about. This makes it harder to give the user guidance. Guidance from the principal that defined the target is probably safe to use, since the defining principal is the one whose security depends on proper use of the target. Still, it may be questionable to rely on third-party developers for part of the security user interface.

```
// this class shows how to implement a trusted subsystem with
// stack introspection

public class FS implements FileSystem {
    private boolean usePrivs = false;

    public FS() {
        try {
            PrivilegeManager.checkPrivilege("UniversalFileRead");
            usePrivs = true;
        } catch (ForbiddenTargetException e) {
            usePrivs = false;
        }
    }

    public FileInputStream getInputStream(String path) {
        // only enable privileges if they were there when we were constructed

        if(usePrivs)
            PrivilegeManager.enablePrivilege("UniversalFileRead");
        return new java.io.FileInputStream(path);
    }
}

// this class shows how a privilege is enabled before a potentially
// dangerous operation

public class TrustedService {
    public static FileSystem getScratchSpace() {
        PrivilegeManager.enablePrivilege("UniversalFileRead");

        // SubFS class from the previous example
        return new SubFS("/tmp/TrustedService", new FS());
    }
}
```

Fig. 4. This example shows how to build a `FileSystem` capability (see figure 2) as an example of a protected subsystem using extended stack introspection. Note that figure 2's `SubFS` can work unmodified with this example.

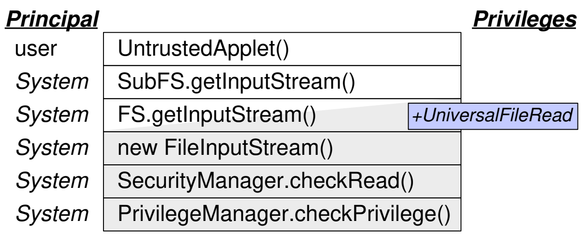

Fig. 5. This figure shows the call stack (growing downward) for a file system access by an applet using the code in figure 4. The grey areas represent stack frames running with the `UniversalFileRead` privilege enabled. Note that the top system frames on the stack can run unprivileged, even though those classes have the system principal.

3.3.4.3 *Signing Details.* Microsoft and Netscape also differ over their handling of digital signatures. In the Microsoft system, each bundle of code carries at most one signature[14], and the signature contains a list of targets that the signer thinks the code should be given access to. Before the code is loaded, the browser asks the policy engine whether the signer is authorized to access the requested targets; if so, the code is loaded and given the requested permissions.

In the Netscape system, code may have several signers, and signatures mention no targets. Instead, once the code is loaded, the code itself must tell the policy engine which targets it will want access to. The decision whether to grant access to those targets is made by the policy engine, based on the identities of the signers. Applets are strongly advised to request their privileges at startup, though this rule is not enforced. Failure to request privileges on startup might lead, for example, to the user working with a document-editing applet and then discovering that he cannot save his work without violating his security policy.

Since the Netscape system allows a class to be signed by several principals, there must be a way to combine the policy engine's opinions about each of the signers into a single policy decision. This is done by using a "consensus voting" rule. Intuitively, consensus voting means that one negative vote can force access to be forbidden, while at least one positive vote (and no negative votes) is required in order to allow access. Since the local user and site administrator may not know all of the signers, the policy engine can be told to vote "abstain" for an unknown signer.

Allowing multiple signers, as Netscape does, seems better than requiring a single singer. Since different users and sites may trust different signers, multiple signatures are required in order to produce a single signed object that everyone will accept. With a single-signature scheme, the server has to have multiple versions of the code (one for each signer) and must somehow figure out which version to send to which client. Also, one signer might sign a subset of the classes, creating a trusted subsystem within a potentially untrusted applet.

Putting the target-access requests in the signature, as Microsoft does, provides superior flexibility in another dimension. It allows a signature to be a partial endorsement of a piece of code. For example, a site administrator might sign an outside applet, saying (in effect), "The local security policy allows this applet to access outside network addresses, but nothing more."

Many other details of the Microsoft and Netscape designs are omitted here for brevity.

3.4 Third Approach: Name Space Management

This section presents a modification to Java's dynamic linking mechanism which can be used to hide or replace the classes seen by an applet as it runs.

We have implemented a full system based on name space management, as an extension to Microsoft Internet Explorer 3.0. Our implementation did not modify the JVM itself, but changed several classes in the Java library. The full system is implemented in 4500 lines of Java, most of which manages the user interface.

We first give an overview of what name space management is and how it can be used as a security mechanism. Then we describe our implementation in detail.

[14] Actually, a sequence of signatures is allowed, but the present implementation recognizes only the first one.

Original name	Alice	Bob
java.net.Socket	security.Socket	java.net.Socket
java.io.File	—	security.File
...	...	...

Fig. 6. Different principals see different name spaces. In this example, code signed by Alice cannot see the `File` class, and for Bob it has been replaced with compatible subclasses.

3.4.1 *Design.* With name space management, we enforce a given security policy by controlling how names in a program are resolved into runtime classes. We can either remove a class entirely from the name space (thus causing attempts to use the class to fail), or we can cause its name to refer to a different class which is compatible with the original. This technique is used in Safe-Tcl [Borenstein 1994] to hide commands in an untrusted interpreter. Plan 9 [Pike et al. 1990] can similarly attach different programs and services to the file system viewed by an untrusted process.

In an object-oriented language, classes represent resources we wish to control. For example, a `File` class may represent the file system and a `Socket` class may represent networking operations. If the `File` class, and any other class which may refer to the file system, is not visible when the remote code is linked to local classes, then the file system will not be available to be attacked. Instead, an attempt to reference the file system would be equivalent to a reference to a class which did not exist at all; an error or exception would be triggered.

To implement interesting security policies, we can create environments which replace sensitive classes with compatible ones that check their arguments and conditionally call the original classes. These new classes can see the original sensitive classes, but the mobile code cannot. For example, a `File` class could be replaced with one that prepended the name of a subdirectory (see figure 6), much like the capability-based example in figure 2. In both cases, only a sub-tree of the file system is visible to the untrusted mobile code. In order for the program to continue working correctly, each substituted class must be compatible with the original class it replaces (*i.e.*, the replacement must be a subclass of the original).

To make name space management work as a flexible and general access control scheme, we introduce the notion of a *configuration*, which is a mapping from class names to implementations (*i.e.*, Java class files). These configurations correspond to the privilege groups discussed in section 3.1.2. When mobile code is loaded, the policy engine determines which configuration is used for the code's name space, as a function of its principal. If the principals have not yet been seen by the system, the user is consulted for the appropriate configuration.

An interesting property of this system is that all security decisions are made statically, before the mobile code begins execution. Once a class has been hidden or replaced, there is no way to get it back.

3.4.2 *Implementation in Java.* Name space management in Java is accomplished through modifying the Java *ClassLoader*. A ClassLoader is used to provide the *name* → *implementation* mapping. Every class keeps a reference to a ClassLoader which is consulted for dynamic binding to other classes in the Java runtime. Whenever a new class is referenced, the ClassLoader provides the implementation of the new class (see figure 7).

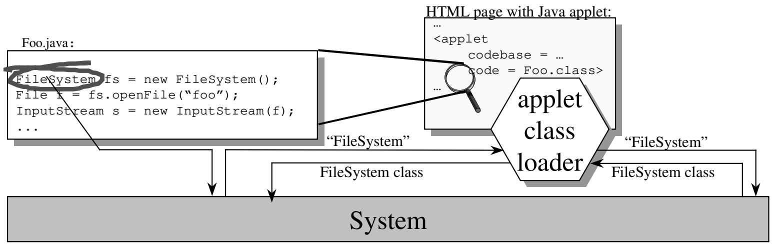

Fig. 7. A *ClassLoader* resolves every class in an applet. If these classes reference more classes, the same ClassLoader is used.

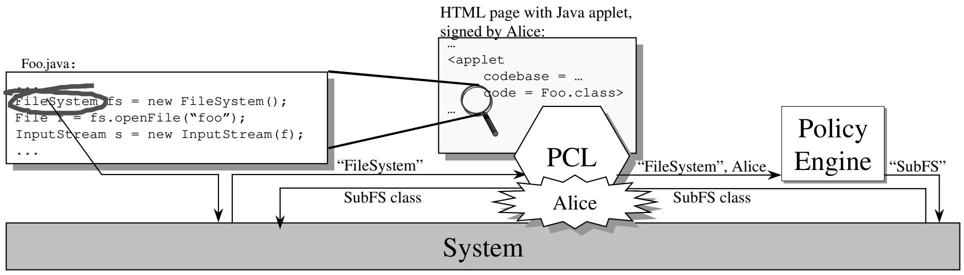

Fig. 8. Changing the name space: A PrincipalClassLoader (PCL) replaces the original class loader. The class implementation returned depends on the principal associated with the PrincipalClassLoader and the configuration in effect for that principal.

Usually in a Web browser, an *AppletClassLoader* is created for each applet. The AppletClassLoader will normally first try to resolve a class name against the system classes (such as `java.io.File`) which ship with the browser. If that fails, it will look for other classes from the same network source as the applet. If two applets from separate network locations reference classes with the same name that are not system classes, each will see a different implementation because each applet's AppletClassLoader looks to separate locations for class implementations.

Our implementation works similarly, except we replace each applet's AppletClassLoader with a PrincipalClassLoader which enforces the configuration appropriate for the principals attached to that class (see figure 8). When resolving a class reference, a PrincipalClassLoader can

(1) throw a `ClassNotFoundException` if the calling principal is not supposed to see the class at all (exactly the same behavior that the applet would see if the class had been removed from the class library – this is essentially a link-time error),

(2) return the class in question if the calling principal has full access to it, or

(3) return a subclass, as specified in the configuration for the applet's principal.

In the third case we hide the expected class by binding its name to a subclass of the original class. Figure 9 shows a class, `security.File`, which can replace `java.io.File`.

When an applet calls `new java.io.File("foo")`, it will actually get an instance of `security.File`, which is compatible in every way with the original class, except it restricts file system access to a subdirectory. It might appear that this could be circumvented by using Java's `super` keyword. However, Java bytecode has no notion of `super`, which is resolved entirely at compile time. Note that, to achieve complete protection, other classes such as `FileInputStream`, `RandomAccessFile`, etc. need to be replaced as well, as they also allow access to the file system.

Name space management also allows third-party trusted subsystems to be distributed as part of untrusted applets (see section 3.3.3). The classes of the trusted subsystem will have different principals from the rest of the applet. Thus, those classes may be allowed to see the "real" classes which are hidden from other applet classes.

The viability of name space management is unknown for future Java systems. New Java features such as the reflection API (a feature of JDK 1.1 which allows dynamic inquiry and invocation of a class's methods) could defeat name space management entirely. Also, classes which are shared among separate subsystems may be impossible to rename without breaking things (*e.g.,* both file system and networking classes use `InputStream` and `OutputStream`). Both reflection and the class libraries could be redesigned to work properly with name space management.

```
package security;
public class File extends java.io.File {

    // Note that System.getUser() would not be available in a system
    // purely based on name space management.  In the current
    // implementation, getUser() examines the call stack for a
    // PrincipalClassLoader. A cleaner implementation would want to
    // generate classes on the fly with different hard-coded prefixes.

    private String fixPath(String path) {
        // to work safely, this would need to properly handle `..' in path
        return "/tmp/" + System.getUser().getName() + "/" + path;
    }

    public File(String path) {
        super(fixPath(path));
    }

    public File(String path, String name) {
        super(fixPath(path), name);
    }
}
```

Fig. 9. Interposition in a system with name space management. We assume that at runtime we can find out who the currently running principal is.

4. ANALYSIS

Now that we have presented three systems, we need a set of criteria to evaluate them. This list is derived from Saltzer and Schroeder [1975].

Economy of mechanism. Designs which are smaller and simpler are easier to inspect and trust.

Fail-safe defaults. By default, access should be denied unless it is explicitly granted.

Complete mediation. Every access to every object should be checked.

Least privilege. Every program should operate with the minimum set of privileges necessary to do its job. This prevents accidental mistakes becoming security problems.

Least common mechanism. Anything which is shared among different programs can be a path for communication and a potential security hole, so as little data as possible should be shared.

Accountability. The system should be able to accurately record "who" is responsible for using a particular privilege.

Psychological acceptability. The system should not place an undue burden on its users.

Several other practical issues arise when designing a security system for Java.

Performance. We must consider how our designs constrain system performance. Security checks which must be performed at run-time will have performance costs.

Compatibility. We must consider the number and depth of changes necessary to integrate the security system with the existing Java virtual machine and standard libraries. Some changes may be impractical.

Remote calls. If the security system can be extended cleanly to remote method invocation, that would be a benefit for building secure, distributed systems.

We will now consider each criterion in sequence.

4.1 Economy of Mechanism

Of all the systems presented, name space management is possibly the simplest. The implementation requires redesigning the ClassLoader as well as tracking the different name space configurations. The mechanisms that remove and replace classes, and hence provide for interposition, are minimal. However, they affect critical parts of the JVM and a bug in this code could open the system to attack.

Extended stack introspection requires some complex changes to the virtual machine. As before, changes to the JVM could destabilize the whole system. Each class to be protected must explicitly consult the security system to see if it has been invoked by an authorized party. This check adds exactly one line of code, so its complexity is analogous to the configuration table in name space management. And, as with name space management, any security-relevant class which has not been modified to consult the security system can be an avenue for system compromise.

Unmodified capabilities are also quite simple, but they have well-known problems with confinement (see section 4.3). A capability system modified to control the propagation of capabilities would have a more complex implementation, possibly requiring stack introspection or name space management to work properly. A fully capability-based Java would additionally require redesigning the Java class libraries to present a capability-style interface — a significant departure from the current APIs, although capability-style classes ("factories") are used internally in some Java APIs.

An interesting issue with all three systems is how well they enable code auditing — how easy it is for an implementor to study the code and gain confidence in its correctness. Name space management includes a nice list of classes which must be inspected for correctness. With both capabilities and stack introspection, simple text searching tools such as `grep` can locate where privileges are acquired, and the propagation of these privileges must be carefully studied by hand. Capabilities can potentially propagate anywhere in the system. Stack annotations, however, are much more limited in their ability to propagate.

4.2 Fail-safe Defaults

Name space management and stack introspection have similar fail-safe behavior. If a potentially dangerous system resource has been properly modified to work with the system, it will default to deny access. With name space management, the protected resource cannot be named by a program, so it is not reachable. With stack introspection, requests to enable a privilege will fail by default. Likewise, when no enabled privilege is found on the stack, access to the resource will be denied by default. (Microsoft sacrifices this property for compatibility reasons.)

In a fully capability-based system, a program cannot do anything unless an appropriate capability is available. In this respect, a capability system has very good fail-safe behavior.

4.3 Complete Mediation

Barring oversights or implementation errors (discussed in sections 4.1 and 4.2), all three systems provide mechanisms to interpose security checks between identified targets and anyone who tries to use them.

However, an important issue is *confinement* of privileges [Lampson 1973]. It should not generally be possible for one program to delegate a privilege to another program (that right should also be mediated by the system). This is the fundamental flaw in an unmodified capability system; two programs which can communicate object references can share their capabilities without system mediation. This means that any code which is granted a capability must be trusted to care for it properly. In a mobile code system, the number of such trusted programs is unbounded. Thus, it may be impossible to ever trust a simple capability system. Likewise, mechanisms must be in place to *revoke* a capability after it is granted. Many extensions to capabilities have been proposed to address these concerns. Kain and Landwehr [1987] proposes a taxonomy for extensions and survey many systems which implement them.

Fundamentally, extended capability systems must either place restrictions on how capabilities can be used, or must place restrictions on how capabilities can be shared. Some systems, such as ICAP [Gong 1989], make capabilities aware of "who" called them; they can know who they belong to and become useless to anyone else. The IBM System/38 [Berstis et al. 1980] associates optional access control lists with its capabilities, accomplishing the same purpose. Other systems use hardware mechanisms to block the sharing of capabilities [Karger and Herbert 1984]. For Java, any such technique would be problematic. To make a capability aware of who is calling it, a certain level of introspection into the call stack must be available. To make a capability object unshareable, you must either remove its class from the name space of potential attackers, or block all communication channels that could be used for an authorized program to leak it (either blocking all inter-program memory-sharing or creating a complex system of capability-sharing groups).

Name space management can potentially have good confinement properties. For example, if a program attempts to give an open `FileInputStream` to another program which is forbidden access to the file system, the receiving program will not be able to see the `FileInputStream` class. Unfortunately, it could still likely see `InputStream` (the superclass of `FileInputStream`), which has all the necessary methods to use the object. If `InputStream` were also hidden, then networking code would break, as it also uses `InputStream`. This problem could possibly be addressed by judicious redesign of the Java class libraries.

Stack introspection has excellent confinement. Because the stack annotations are not directly accessible by a program, they can neither be passed to nor stolen by another program. The only way to propagate stack annotations is through method calls, and every subsequent method must also be granted sufficient privilege to use the stack annotation. The system's access control matrix can thus be thought of as mediating delegation rights for privileges. Because the access matrix is consulted both at creation and at use of privileges, privileges are limited to code which is authorized to use them.

4.4 Least Privilege

The principle of least privilege applies in remarkably different ways to each system we consider.

With name space management, privileges are established when the program is linked. If those privileges are too strong, there is no way to revoke them later — once a class name is resolved into an implementation, there is no way to unlink it.

In contrast, capabilities have very desirable properties. If a program wishes to discard a capability, it only needs to discard its reference to the capability. Likewise, if a method only needs a subset of the program's capabilities, the appropriate subset of capabilities may be passed as arguments to the method, and it will have no way of seeing any others. If a capability needs to be passed through the system and then back to the program through a call-back (a common paradigm in GUI applications), the capability can be wrapped in another Java object with non-`public` access methods. Java's package scoping mechanism provides the necessary semantics to prevent the intermediate code from using the capability.

Stack introspection provides an interesting middle-ground. A program which enables a privilege and calls a method is *implicitly* passing a capability to that method. If a privilege is not enabled, the corresponding target cannot be used. This allows straightforward auditing of system code to verify that privileges are enabled only for a limited time and used safely. If a call path does not cross a method that enables its privileges, then that call path cannot possibly use the privilege by accident. Also, there is a `disablePrivilege()` method which can explicitly remove a privilege before calling into untrusted code. Most importantly, when the method which enabled a privilege exits, the privilege disappears with it. This limits the lifetime of a privilege.

4.5 Least Common Mechanism

The principle of least common mechanism concerns the dangers of sharing state among different programs. If one program can corrupt the shared state, it can then corrupt other programs which depend on it. This problem applies equally to all three Java-based systems. An example of this problem was Hopwood's `interface` attack [Dean et al. 1996], which combined a bug in Java's interface mechanism with a shared public variable to ultimately break the type system, and thus circumvent system security.

This principle is also meant to discuss the notion of *covert storage channels* [Lampson 1973], an issue in the design of multi-level secure systems [National Computer Security

Center 1985]. Java presently makes no effort to limit or control covert channels, but this could be an interesting area for future work.

4.6 Accountability

In the event that the user has granted trust to a program which then abuses that trust, logging mechanisms will be necessary to prove that damages occurred and then seek recourse.

In each system, the interposed protection code can always record *what* happened, but it requires more effort to identify the principal responsible.

In the stack introspection system, every call to enable a privilege can be logged; an administrator can learn which principal enabled the privileges to damage the system.

In a capability system, a capability can remember the principal to which it was granted and log this information when invoked. If the capability can be leaked to another program (see section 4.3), the principal logged will not be the same as the principal responsible for using the capability. A modified capability system would be necessary for strong accountability.

With name space management, information about principals is not generally available at run-time. This information could possibly be associated with Java threads or stored in static variables behind interposed classes. Likewise, capabilities could store a principal in a private variable.

This is all hypothetical, unfortunately, since current browsers do not provide the tamper-resistant logging necessary for trustworthy auditing. Once available, any of these architectures should be able to use it.

4.7 Psychological Acceptability

The user interface is the most important aspect of the security system. If a Web browser shows its security dialog box too often, users will learn to ignore its contents and hit "OK" to continue what they were doing.

All three systems here can present the same fundamental interface to a user. When signed mobile code arrives in the Web browser, the user can be queried whether they trust the signatory. The original version of Microsoft's Authenticode [Microsoft Corporation 1996] followed a "shrink-wrap" model — once the software was installed, it had unrestricted access to the user's machine. The systems here can provide the user more fine-grained control over which privileges are granted. One promising strategy is for the Web browser to allow common "profiles" of privileges with intuitive-sounding names (*e.g.*, "typical game privileges") rather than a low-level list of primitives (limited file access, limited network access, full-screen video, etc.).

The user must become involved in security decisions at some point. The important question is when. There are two choices: either when the applet is loading or when the applet attempts a privileged operation. Both times have advantages and disadvantages. The advantages of asking early are that the user does not spend time working in an applet only to find out that they cannot save their work, and the applet cannot generate a series of fake security dialogs to get the user in the habit of clicking "OK".[15] The disadvantage is that the user must decide what privileges to grant an applet before the applet starts running, so a user cannot try out an applet first. The advantage of asking the user later is that the

[15] While the sandbox model puts up a warning strip on windows opened by untrusted code, windows opened by JavaScript have no such warning.

applet has done as much as it can without privilege, so the user may quit the applet first, and never see a security dialog. The disadvantages are the vulnerability to the spoofing outlined above, and the fact that the user may be stuck with no way to save their last hour of work. It is not clear that either strategy is always correct.

All of the architectures discussed in this paper potentially allow for security dialogs to occur either early or late. The dialog can always be presented early, and the answer remembered for later. A capability system could use the late strategy by displaying the dialog only when a privileged capability is requested. The stack introspection model naturally accommodates the late strategy, by presenting the dialog when checking whether a privilege is enabled, *i.e.*, as a privileged operation is requested. For name space management, implementing the late strategy would be based on the lazy nature of dynamic linking in Java: a class does not have to be loaded until it is actually needed. Netscape's implementation of stack introspection uses the late strategy, while Microsoft's stack introspection and our name space management use the early strategy.

In all cases, the user's security preferences can be saved persistently, to prevent repeated dialog boxes. Additionally, these persistent privileges could be preset by the user's site administrator, further reducing the need for dialog boxes.

4.8 Performance

As discussed in section 1, performance is one of the attractions of language-based protection. Because hardware protection is so much slower than any of the systems presented in this paper, we will instead discuss the performance differences among the three software systems. In all cases, we are assuming that the JVM uses a JIT compiler to generate and execute efficient machine code.

The stack introspection system has the highest runtime costs. At runtime, system classes must check whether the current enabled privileges allow them to proceed. At worst, this will have cost proportional to the current stack depth. These checks occur less often then one might think. Currently, stack introspection is used only to guard when a file or network connection is opened (an already expensive operation). The input and output streams act as capabilities for the open file or network connection and need no further security checks on read and write operations. While a specific input or output stream could leak, the general ability to open a file or network connection would still be contained.

Name space management does not incur any overhead at runtime, nor do unmodified capability systems. However, all systems must pay similar runtime costs when they implement interposition layers (*i.e.*, to validate or limit arguments to low-level system routines).

4.9 Compatibility

One lesson we learned from the implementations of both name space management and extended stack introspection is that language based protection can be implemented on top of a type-safe language without diverging much from the original specification of that language. For both name space management and stack introspection, old applets, those written against the original Java API and unaware of the new security mechanisms, will continue to run unmodified in browsers equipped with the new authorization scheme. As long as they only use features allowed by the traditional sandbox security policy, they will notice no difference.

A notable exception are capability systems. As mentioned in section 4.1, a capability system would require a new library API and thus completely break compatibility with traditional Java APIs.

4.10 Remote Calls

Recent systems are beginning to offer convenient distributed programming in Java, using a distributed object model similar to that of SRC network objects [Birrell et al. 1993] or CORBA [Siegel 1996]. It would be highly desirable for language based security schemes to easily extend to a distributed environment. A single security policy and implementation mechanism could deal naturally with both local and remote code.

Capabilities extend quite naturally across a network. If remote object references include an unguessable string of bits (*i.e.*, 128 random bits), then a remote capability reference has all the security properties of a local capability [Tanenbaum et al. 1986; Gong 1989] (assuming all communication is suitably encrypted to prevent eavesdropping). Unfortunately, the confinement issues discussed in section 4.3 become even more problematic. If the capability is leaked to a third party, then the third party has just as much power to use the capability as its intended holder. By themselves, networked capabilities offer no way for an object to identify its remote caller. However, if the remote method invocation used a cryptographically authenticated channel (as provided by SSL [Freier et al. 1996] or Taos [Wobber et al. 1994]), the channel's remote identity might be useful. Gong [1989] and van Doorn et al. [1996] describe implementations of this.

Name space management is comparable to the directory services offered by most RPC systems. The directory server, which can be reached through a well-known mechanism, is used to map names to objects. An interesting possibility would be for a secure directory server to provide different objects to different remote identities.

Stack introspection is interesting because it helps answer the question "on whose behalf is this request running?". This can become very complex to answer when RPCs pass through a number of different systems, calling one another. Secure RPC standards [Hu 1995; Object Management Group 1996] refer to this as *delegation*. When Java code is receiving a remote method call and invoking another one, the stack annotations could possibly help mediate the delegation information.

Actually building a secure RPC system using any of the mechanisms in Java is future work. Current Java RPC systems, with the possible exception of Electric Communities [1996], have no provisions for flexible security policies.

5. CONCLUSION

Software-based protection systems are coming into common use, driven by their inherent advantages in both performance and portability. Software fault isolation, proof-carrying code, or language-based mechanisms can be used to guarantee memory-safety. Secure system services cannot be built without these mechanisms, but may require additional system support to work properly.

We have described three designs which support interposition of security checks between untrusted code and important system resources. Each design has been implemented in Java and both extended stack introspection and name space management have been integrated in commercial Web browsers.

All three designs have their strengths and weaknesses. For example, capability systems are implemented very naturally in Java. However, they are only suitable for applications where programs are not expecting to use the standard Java APIs, because capabilities require a stylistic departure in API design.

Name space management offers good compatibility with existing Java applets but Java's libraries and newer Java mechanisms such as the reflection API may limit its use.

Extended stack introspection also offers good compatibility with existing Java applets and has reasonable security properties, but its complexity is troubling and it relies on several artifacts of Sun's Java Virtual Machine implementation.

We believe the best solution is to combine elements of these techniques. Name space management allows transparent interposition of security layers between system and applet code with no run-time performance penalty. Stack introspection can allow legacy system code to run with less than full privileges without being rewritten in a capability style. Yet, capabilities provide a well understood extension to remote procedure calls. Understanding how to create such a hybrid system is a main area for future research.

ACKNOWLEDGEMENTS

Thanks to Jim Roskind, Raman Tenneti, and Tom Dell of Netscape for helping us understand their design, and to Netscape for supporting Dan Wallach as a visiting member of the design and implementation group. Thanks to Jeff Bisset, Mike Jerger, and Mike Toutonghi for helping us understand the Microsoft design. Any remaining errors in the description of these designs are ours.

We also thank Li Gong for helping us solidify our ideas. Thanks to Andrew Appel, Mark Miller, Marianne Mueller, Ben Renaud, and Frank Yellin for many interesting conversations about our work. Martín Abadi, Dan Boneh, Luca Cardelli, Simson Garfinkel, Rob Shillner, and Steve Wallach gave valuable feedback on drafts of this paper. Dave Gifford and the anonymous referees made many helpful suggestions.

Our work is supported by donations from Sun Microsystems, Bellcore, and Microsoft. Edward Felten is supported in part by an NSF National Young Investigator award.

REFERENCES

ANDERSON, T. E., LEVY, H. M., BERSHAD, B. N., AND LAZOWSKA, E. D. 1991. The interaction of architecture and operating system design. In *Proceedings of the Fourth ACM Symposium on Architectural Support for Programming Languages and Operating Systems*.

BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. 1995. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*. 66–77.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. 1995. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*. 251–266.

BERSTIS, V., TRUXAL, C. D., AND RANWEILER, J. G. 1980. System/38 addressing and authorization. In *IBM System/38 Technical Developments*, 2nd ed. IBM, 51–54.

BIRRELL, A., NELSON, G., OWICKI, S., AND WOBBER, E. 1993. Network objects. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. 217–230.

BORENSTEIN, N. S. 1994. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *IFIP International Working Conference on Upper Layer Protocols, Architectures and Applications*.

CLARK, D. AND WILSON, D. 1987. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*. Oakland, CA.

DEAN, D., FELTEN, E. W., AND WALLACH, D. S. 1996. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. Oakland, CA, 190–200.

DROSSOPOULOU, S. AND EISENBACH, S. 1997. Java is type safe — probably. In *Proceedings of the Eleventh European Conference on Object-Oriented Programming*.

Electric Communities 1996. *The Electric Communities Trust Manager and Its Use to Secure Java*. Electric Communities. `http://www.communities.com/company/papers/trust/`.

FABRY, R. S. 1974. Capability-based addressing. *Communications of the ACM 17,* 7 (July), 403–411.

FLANAGAN, D. 1997. *JavaScript: The Definitive Guide*, 2nd ed. O'Reilly & Associates, Inc.

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. 1996. *The SSL Protocol: Version 3.0*. Internet draft, `ftp://ietf.cnri.reston.va.us/internet-drafts/draft-freier-ssl-version3-01.txt`.

GOLDSTEIN, T. 1996. *The Gateway Security Model in the Java Electronic Commerce Framework*. JavaSoft. `http://www.javasoft.com/products/commerce/jecf_gateway.ps`.

GONG, L. 1989. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*. Oakland, CA, 56–63.

GONG, L. 1997a. New security architectural directions for Java. In *Proceedings of IEEE COMPCON '97*.

GONG, L. 1997b. personal communication.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley.

HARDY, N. 1985. KeyKOS architecture. *ACM Operating Systems Review 19,* 4 (Oct.), 8–25.

HU, W. 1995. *DCE Security Programming*. O'Reilly & Associates, Inc.

JAEGER, T., RUBIN, A. D., AND PRAKASH, A. 1996. Building systems that flexibly control downloaded executable content. In *Sixth USENIX Security Symposium Proceedings*. San Jose, CA, 131–148.

JONES, A. K. AND LISKOV, B. H. 1976. A language extension for controlling access to shared data. *IEEE Transactions on Software Engineering SE-2,* 4 (Dec.), 277–285.

KAIN, R. Y. AND LANDWEHR, C. E. 1987. On access checking in capability-based systems. *IEEE Transactions on Software Engineering SE-13,* 2 (Feb.), 202–207.

KARGER, P. A. AND HERBERT, A. J. 1984. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*. Oakland, CA, 2–12.

LAMPSON, B. W. 1971. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*. Princeton University, 437–443. Reprinted in *Operating Systems Review*, 8(1):18–24, Jan. 1974.

LAMPSON, B. W. 1973. A note on the confinement problem. *Communications of the ACM 16,* 10 (Oct.), 613–615.

LEVY, H. M. 1984. *Capability-Based Computer Systems*. Digital Press.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.

Microsoft Corporation 1996. *Proposal for Authenticating Code Via the Internet*. Microsoft Corporation. `http://www.microsoft.com/security/tech/authcode/authcode-f.htm`.

MILNER, R. AND TOFTE, M. 1991. *Commentary on Standard ML.* MIT Press, Cambridge, MA.

NATIONAL COMPUTER SECURITY CENTER. 1985. *Department of Defense Trusted Computer System Evaluation Criteria (The Orange Book)*.

NECULA, G. C. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*. Seattle, WA, 229–243.

NEUMANN, P. G., BOYER, R. S., FEIERTAG, R. J., LEVITT, K. N., AND ROBINSON, L. 1980. A provably secure operating system: The system, its applications, and proofs. Tech. Rep. CSL-116, 2nd Ed., SRI International. May.

OBJECT MANAGEMENT GROUP. 1996. *Common Secure Interoperability*. OMG Document Number: orbos/96-06-20.

OUSTERHOUT, J. K. 1990. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of Summer 1990 USENIX Conference*. 247–256.

PIKE, R., PRESOTTO, D., THOMPSON, K., AND TRICKEY, H. 1990. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*. London, 1–9.

REES, J. A. 1996. A security kernel based on the lambda-calculus. Tech. Rep. A.I. Memo No. 1564, Massachusetts Institute of Technology, Artificial Intelligence Laboratory. Mar.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proceedings of the IEEE 63,* 9 (Sept.), 1278–1308.

SCHMITT, B. 1997. *Shockwave Studio: Designing Multimedia for the Web*. O'Reilly & Associates, Inc.

SCHROEDER, M. D. AND SALTZER, J. H. 1972. A hardware architecture for implementing protection rings. *Communications of the ACM 15,* 3 (Mar.), 157–170.

SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*. 213–227.

SIEGEL, J., Ed. 1996. *CORBA Fundamentals and Programming*. John Wiley and Sons.

TANENBAUM, A. S., MULLENDER, S. J., AND VAN RENESSE, R. 1986. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*. Cambridge, MA, 558–563.

VAN DOORN, L., ABADI, M., BURROWS, M., AND WOBBER, E. 1996. Secure network objects. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. Oakland, CA.

WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. 1993. Efficient software-based fault isolation. In *Proceedings of the Fourteenth Symposium on Operating System Principles*.

WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. 1994. Authentication in the Taos operating system. *ACM Transactions on Computer Systems 12,* 1 (Feb.), 3–32.