Technical Report 546-97, Department of Computer Science, Princeton University, April 1997.

Extensible Security Architectures for Java

Dan S. Wallach* dwallach@cs.princeton.edu
Dirk Balfanz balfanz@cs.princeton.edu
Drew Dean ddean@cs.princeton.edu
Edward W. Felten felten@cs.princeton.edu

Department of Computer Science
Princeton University

Abstract

Mobile code technologies such as Java, JavaScript, and ActiveX generally limit all programs to a single security policy. However, software-based protection can allow for more flexible security models, with potentially significant performance improvements over traditional hardware-based solutions. We describe and analyze three implementation strategies for interposing flexible security policies in software-based security systems. Implementations exist for all three strategies: several vendors have adapted capabilities to Java, Netscape Communicator extended Java's stack introspection, and we built a type-hiding system as an add-on to Microsoft Internet Explorer.

1 Introduction

The growth of the World Wide Web has enabled the Internet to reach new levels of popularity and ease of use. We are now seeing the arrival of HTML-enabled mail and news-reading tools, heralding the complete acceptance of Web documents as a medium of exchange.

As Web applications have blossomed, 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 [16], JavaScript [11], ActiveX [26], and Shockwave [35] make this possible.

Users and developers love mobile code, but it raises serious security concerns. While code has been distributed over networks for years, the risks of attack have been manageable. The ubiquitous and seamless integration of mobile code with Web browsing increases the danger because 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 hostile code.

*Portions of this work were done at Netscape Communications Corp.

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. However, 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 [13, 9] have built capability systems, Netscape has extended Java's stack introspection, and we designed an add-on to Microsoft Internet Explorer which hides and renames Java types. 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 [36]. Recent mobile code systems, however, rely on software rather than hardware for protection. The switch to software mechanisms is being driven by two technological trends: portability and performance.

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. 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.

Performance Second, software protection offers significantly cheaper cross-domain calls[1]. We measured

[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

Figure 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.

the performance of a null call between Common Object Model (COM) objects on a 180 MHz Pentium 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. This ratio appears to be growing even larger in newer processors [31, 1].

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 instrumented the Sun Java virtual machine to measure the number of calls across trust boundaries, that is, the number of procedure calls in which the caller is either more trusted or less trusted than the callee. We measured two workloads. CaffeineMark[2] is the most 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.)

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 procedure call, we calculate that these programs spend roughly 0.5% of their time crossing trust boundaries in the Java system with software protection.

Using our measured cost of 230 μsec for a cross-task call, we can calculate the estimated 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 näive 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 a crossing was low. 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 [41], proof-carrying code [29], and type-safe languages are three popular ways to ensure memory protection.

1.2.1 Software Memory Protection

Lucco, *et al.*, introduced *software fault isolation* [41] in 1993. They showed that memory accesses could be checked in software with a 5–30% slowdown. A similar technique could be used to check the target address of every branch. By optimizing the placement of the dynamic 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 introduced *proof-carrying code* [29] in 1996. 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. 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 [24] and is also used in current research operating systems such as SPIN [2]. By using a safe language, memory accesses can be controlled [27]. If the language has a mechanism for enforcing abstraction boundaries (in Java, this means ensuring that `private` variables and methods

[2] `http://www.webfayre.com/pendragon/cm2/`

[3] We omitted internationalization examples and the JDK 1.1 applet.

[4] `http://www.javasoft.com/applets/applets.html`

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 [33, 5] or static type checking (as is mostly the case in Java [8]). 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 actual implementation may 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.*, studied some of the security problems involved in creating an extensible operating system [37]. 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 also 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 getting memory protection but providing secure system services. This paper considers how secure services can be built solely with software protection.

[5]GUI event manipulation may seem harmless, but observing GUI events allows an attacker to observe typed passwords or other sensitive information, and generating GUI events allows the attacker to create keystrokes to execute dangerous commands.

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. It is a type-safe language, so the Java virtual machine (JVM) prevents programs from accessing memory or calling methods without authorization [25]. Existing JVM implementations 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 blocks many useful kinds of programs from being written. For example, the complete prohibition on filesystem access makes it impossible to write a replacement for the user's local word processor; limitations on network access likewise complicate the design of teleconferencing programs.

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 network or filesystem access, is executing "locally" or "remotely." Sun's JVM has two inherent properties used to make these checks:

- Every *class* in the JVM which came from the network was loaded by a *ClassLoader*. Each class includes a reference to its ClassLoader. Classes which came from the local filesystem have a special system class loader. Thus, local classes can be distinguished from remote classes by their ClassLoader references.
- 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 would check if

the action requested was allowed (using the mechanism described above), and throw an exception if it is not allowed.

In practice, this design proved insufficient. In addition to a number of security-related bugs in the first implementations [7], many developers complained that the *policy*, applied equally to all applets, was too inflexible to implement many desirable "real" applications. The sandbox policy provides network access to exactly one machine, the origin of the applet, and provides no local filesystem access. While this might make sense across the Internet with untrusted code, programs written for internal use in a corporation may sensibly wish to use the local filesystem as well as contact multiple network servers.

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 presume 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 Many 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.

Type Hiding An interesting property of dynamic loading is the ability to create an environment where code cannot see all the types that would normally be available. By hiding a type, we can prevent code from invoking its methods.

In this section, we will focus on how each method allows and enforces interposition of protective code between potentially dangerous primitives and untrusted code. In section 4, we will then 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 interposing a reference monitor [28] through which all calls to the component must pass. 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 convenient to have an extensible reference monitor, allowing for more general interposition of code between system resources and potentially untrusted principals.

Traditionally, extensible reference monitors are built using *trusted subsystems*. For example, the utility to change a password must be trusted to safely edit the password database. For this to succeed, the system must provide for trusted subsystems, interposed between the password database and the users. Clark and Wilson [6] offer 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, a digital signature which accompanies an applet becomes a principal. Principals map one-to-one with private keys. These signatures represent *endorsements* of the code by the signer[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 Java runtime. 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

[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 by a third-party, there is no strong way for a signature to signify *authorship*.

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 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 filesystem 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.

3.2 First Approach: Capabilities

In many respects, Java provides an ideal way to implement a traditional capability system [10, 24, 20]. Electric Communities [9] and JavaSoft [13][8] have implemented such systems. This section discusses general issues for implementing capabilities in Java, rather than specifics of the Electric Communities or JavaSoft implementations.

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 [17, 30, 39]. 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, it 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 [24]. 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 filesystem access, for example, it need only not receive a filesystem 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 filesystem capability which it acquired on startup or through a broker. If a program does not receive such a capability, it should 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 filesystem — 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 filesystem. 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` *type* 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.

[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 [15].

[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 filesystem.
// 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 filesystem.  Calls to getInputStream()
// are prepended with the desired filesystem 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);
    }
}
```

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

3.3 Second Approach: Extended Stack Introspection

This section presents an extension to the Java's current stack-introspection mechanism to support multiple *principals*, each of which may have distinct privileges.

The following approach is modeled after an implementation which will ship as part of Netscape Communicator 4.0. The implementation required about 5500 lines of Java source code (including *javadoc* in-line documentation) and about 600 lines of C source code in the Java virtual machine, including small changes to the garbage collector, method invocation, and just-in-time compiler. Existing applets run without any modification.

This security infrastructure can be decomposed into a number of different sections. Each Java class is annotated with the set of principals who signed it. The policy engine stores an access matrix [22], identifying which principals may access which targets in the system. When code wishes to use a privilege, it must explicitly *enable* that privilege; if the policy engine permits the access, an *annotation* is recorded on the call-stack. When a class which manages a potentially dangerous resource wishes to verify that somebody is permitted to use that resource, it can examine the call-stack for these annotations.

We first describe a simplified model of the system, then we describe many of the extensions and additions which were found necessary in the production system.

3.3.1 Targets

This system maintains a central registry of targets. When a resource is to be protected, a corresponding target will be registered in this table. Targets are named with a (*prin*, *string-name*) pair, where *prin* is a principal and *string-name* is a human-readable string (*e.g.*, "unrestricted-file-read" or "unrestricted-network-access"). Initially, the only targets are those defined by the *System* principal; these represent objects that the Java system itself wants to protect, such as filesystem access, network access, etc. Any Java class may define a new target, but a class may not create a target with a different principal than itself. Because all targets are stored in a central registry, the restriction on a target's principal helps disambiguate targets with the same string-name from separate programs. Furthermore, this prevents an attacker from interfering with targets from another principal. Additionally, *macro targets* may refer to a set of targets, corresponding to the privilege groups in section 3.1.2.

3.3.2 Policy Engine

The purpose of the policy engine is, given a principal requesting access for a target, to say whether the access requested is *allowed* or *forbidden*. The engine will track the security preferences of the user (and administrator) and store these preferences in an access matrix [22]. In addition to the usual *allowed* and *forbidden* labels, a default *blank* label is also present.

When code attempts to enable privileges for a target, the policy engine is consulted with that code's principals and the target in question. If the matrix entry is *allowed* or *forbidden*, the corresponding permission is immediately returned. However, if the state is *blank*, the user or administrator has not made any security decisions about whether this particular principal may access the specified target. A user dialog allows privileges to be granted just once, until the browser exits, or to persist until the underlying digital signature expires.

When a class has been signed by multiple principals, the system uses a "consensus voting rule" to combine the policies for those principals. Intuitively, consensus voting means that one negative vote can veto a decision, and at least one positive vote (and no negative votes) is required to approve.

An interface also exists for a site administrator to preload a set of policies into the browser. These pre-loaded policies cannot be overridden.

3.3.3 Protecting and Using Resources

Two fundamental primitives are necessary to use this system: `enablePrivilege(`*`target`*`)` and `checkPrivilege(`*`target`*`)`. When a dangerous resource (such as the filesystem) needs to be protected, two steps are necessary: a target must be defined for the resource (see section 3.3.1) and the dangerous resource 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 (see section 3.3.2) to see if the principal of the caller is permitted to use the resource. Once permitted, the *enabled privilege* is written in a hidden field on the caller's stack frame. When the caller returns, the privilege is automatically discarded along with the rest of the stack frame.

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

3.3.4 Example: Network Policy

The default Java policy is to allow network connections to be created only to the host from which the applet originated. Here, we show how this policy could be re-implemented in this framework by interposing a method, `safe_net_connect()`, between untrusted applets and

Principal		Privilege Operations
user	user1()	
user	user2()	
user	user3()	
System	safe_net_conn()	+networking
System	net_connect()	
System	checkPrivilege()	

Figure 3: `safe_net_connect()` enables *networking* privileges before calling the low-level `net_connect()`. Grey areas indicate enabled privileges for *networking*.

Principal		Privilege Operations
	A1()	
	A2()	
NetSrv	Netsrv1()	+networking
NetSrv	Netsrv2()	
System	read_URL()	
System	net_connect()	
System	checkPrivilege()	

Figure 4: NetSrv wishes to export connections to its own server, but not general network access. `Netsrv1()` enables *networking* privileges before calling system code. Grey areas indicate enabled privileges for *networking*.

potentially dangerous system code. We first define a target called (*System*, *networking*)[10]. The network-related code in the Java runtime library is signed by the *System* principal, and the policy engine will allow access to *networking* by *System*.

The `safe_net_connect()` procedure calls `enablePrivilege(`*networking*`)` if it can determine the requested connection is to the host of the applet's origin; otherwise `safe_net_connect()` does not enable its privileges at all (see figure 3). When `safe_net_connect()` calls the low-level `net_connect()` procedure, `net_connect()` will proceed with the connection if and only if `checkPrivilege(`*networking*`)` succeeds.

Note that the applet cannot violate security by directly calling `net_connect()`. The connection is allowed only if the privileges for the *networking* target are enabled, and the only way to do this is to call through `safe_net_connect()`.

3.3.5 Example: a Trusted Applet

Suppose a user wants to designate some principal as authorized to connect to any network address. The user will instruct the policy engine to allow the *networking* target for the applet's principal. Code signed by that principal will then enable its privileges to include *networking* and call directly into `net_connect()`.

3.3.6 Example: a Trusted Subsystem

Rather than trust an unknown vendor, users are more likely to trust name-brands. Such a trusted vendor can create and digitally sign a set of Java classes that implements a trusted subsystem. These classes, distributed with any applet, can extend the security model in a controlled way. For example, a subsystem may allow the applet to connect to a specific host on the network other than the host where the applet originated. An example is shown in figure 5, in which the trusted subsystem allows a connection to the *NetSrv* network-based service to which the user may have subscribed. To make this work, it is imperative that the extended network privileges are only available to the trusted classes, and not to all classes in the system.

In this example, the policy is changed to allow access to the *networking* target by the *NetSrv* principal. The trusted `Netsrv1()` method calls `enablePrivilege(`*networking*`)`, as shown in figure 4 and then eventually calls through `Netsrv2()` into `net_connect()` to connect to the appropriate machine at NetSrv headquarters.

This gives applets three ultimate ways to make a network connection: call through `Netsrv1()` and connect to the NetSrv machine, call into `safe_net_connect()` and connect to the host from which the applet was loaded, or request *networking* privileges for itself. There are no other ways to enable privileges for the *networking* target,

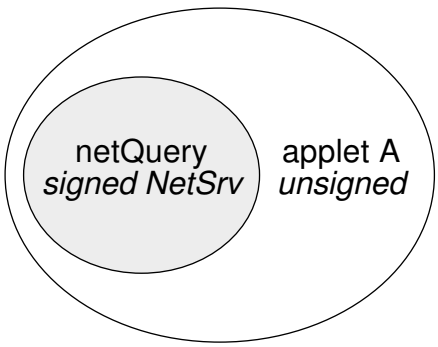

Figure 5: NetSrv may provide a subsystem to connect to their own server. They sign their own classes, which may then be distributed as part of any applet.

[10]Referred to as "*networking*" elsewhere for simplicity

so there are no other ways to make a network connection. Note that, even should the applet call `Netsrv2()` directly, the low-level call will still fail.

3.3.7 Details

The Netscape implementation has many additional features beyond those described above. We will attempt to describe a few of these enhancements and features here.

"Smart Targets" Targets are just Java objects. A subclass of `Target` may be stored in the target registry; the subclass may define its own policy. On any policy decision, the target is passed the principal, and it returns a permission (*blank* by default). This is combined with permissions from the access matrix (using the consensus voting rule of section 3.3.2). An example of this is a smart filesystem target which takes the file name as an argument and consults a separate access-control-list.

Stack Introspection Once a privilege has been enabled, all methods within the dynamic scope of the enabling stack frame operate with this privilege and could be subject to attack. In particular, if an attacker could arrange for its own method to be called with an enabled privilege above it on the stack, it might be able to use this privilege to circumvent the security system. We call this a *luring attack*. This can occur, for example, when a system thread calls back into a user class — a common occurrence with Java's Abstract Window Toolkit (AWT). The solution is to "hide" enabled privileges when a method call crosses classes with different principals. When `checkPrivilege(`*target*`)` searches backward up the stack for an enabled privilege to *target*, it will consult the policy engine for the principals of every stack frame and terminate early if it reaches a stack frame whose principals are forbidden access to *target*. The end result is that a luring attacker will only be able to use the enabled privilege if and only if a call to `enablePrivilege(`*target*`)` would have succeeded, anyway.

3.4 Third Approach: Type Hiding

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 type hiding, as an extension to Microsoft Internet Explorer. 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, much of which manages the user interface.

Original name	Alice	Bob
java.net.Socket	java.net.MySocket	java.net.Socket
java.io.FileSystem	—	java.io.SubFS
...	...	...

Figure 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.

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

3.4.1 Design

In type hiding, we enforce a given security policy by controlling how names in a program are resolved into runtime entities. We can either remove an entity entirely from the name-space (thus causing mentioning its name to result in a runtime error), or we can cause its name to refer to a different entity which is compatible with the original. This technique is used in Safe-Tcl [5] to hide commands in an untrusted interpreter. Plan 9 [32] can similarly attach different programs and services to the filesystem viewed by an untrusted process.

In an object-oriented language, classes represent resources we wish to control. For example, a `FileSystem` class may represent the filesystem and a `Window` class may represent graphics operations. If the `FileSystem` class, and any other class which may refer to the filesystem, is not visible when the remote code is linked to local classes, then the filesystem will not be available to be attacked. Instead, an attempt to reference the filesystem 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 `FileSystem` 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 filesystem 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 type hiding work as a flexible and general access control scheme, we introduce the notion of a *con-*

figuration, 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 type has been hidden, there is no way to get it back.

3.4.2 Implementation in Java

We implemented type hiding for signed Java applets within Microsoft Internet Explorer. By hiding dangerous classes from untrusted code, type hiding makes the traditional Java `SecurityManager` unnecessary.

Type hiding in Java is accomplished through modifying the Java `ClassLoader`. A `ClassLoader` is used to provide the *class-name* $\rightarrow$ *class-implementation* mapping. Every class keeps a reference to a `ClassLoader` which is consulted for dynamic binding to other classes in the Java runtime. When methods of a class *C* reference another class *D*, then the `ClassLoader` of class *C* is asked to provide the implementation of class *D* (see figure 7).

Usually in a Web browser, an `AppletClassLoader` is created for each applet. The `AppletClassLoader` will normally first resolve class-names 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 class'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),
2. return the class in question if the calling principal has full access to that type, or
3. return a subclass, as specified in the configuration for the applet's principal.

In the third case we hide the expected class, say *C*, by returning *S*, which is a subclass of *C*. The applet code now has no way to name the class *C*. It cannot create an instance of type *C*, nor can it cast another object to type *C*; in either case this would require naming *C*.[11] Even though the applet is not allowed to directly see *C*, *S* is provided by the system, so it has a different ClassLoader which does allow it to see *C*.

Type hiding also allows third-party trusted subsystems to be distributed as part of untrusted applets (see section 3.3.6). 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.

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 [34].

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.

To implement a security system in Java, we have several other concerns.

[11] It might appear that this could be circumvented by using Java's `super` keyword. However, `super` is resolved at compile time, and the Java bytecode has no such notion.

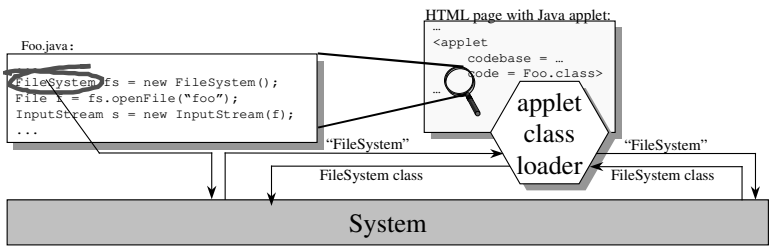

Figure 7: The normal workings of a `ClassLoader`. It resolves every class in an applet and is in turn consulted by the system to resolve classes that are being referenced by classes that it resolved.

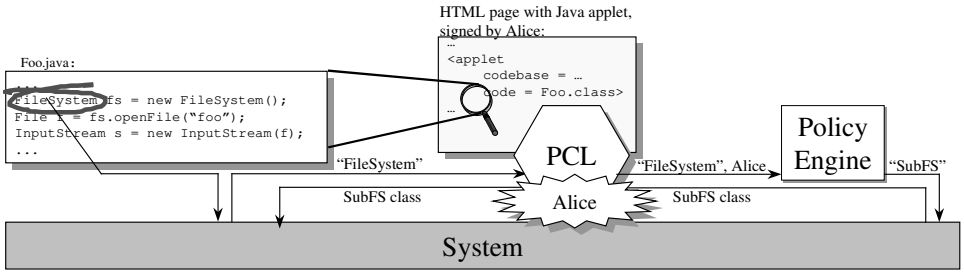

Figure 8: Changing the name-space: A `PrincipalClassLoader` (PCL) replaces the original class loader. The class implementation returned depends on the principal associated with the PCL and the configuration in effect for that principal.

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 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, type hiding is possibly the simplest. The implementation requires redesigning the ClassLoader as well as tracking the different hiding configurations. The mechanisms that remove and replace types, and hence provide for interposition, are minimal. However, they affect critical parts of the Java system and a bug in this code could easily compromise the whole system. Likewise, the configurations must include every class to be protected. Any omissions can result in system compromise.

Extended stack introspection requires some complex changes to the virtual machine. Like the ClassLoader changes above, changes to the JVM could destabilize the whole system, so a great deal of effort was spent testing the basic implementation. Each class to be protected must explicitly consult the security system to see if 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 type hiding. And, similar to type hiding, 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 contain the propagation of capabilities would have a more complex implementation, possibly requiring stack introspection or type hiding to work properly. A fully capability-based Java would additionally require redesigning the Java class libraries to present a capability-style interface — a radical departure from the current API.

4.2 Fail-safe Defaults

Type hiding 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 type hiding, the protected resource would not appear in the program's name space so the resource would not be 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.

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 [23]. 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 [20] propose 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 [14], make capabilities aware of "who" called them; they can know who they "belong to" and become useless to anyone else. The IBM System/38 [3] associates optional access control lists with its capabilities, accomplishing the same purpose. Other systems use hardware mechanisms to block the sharing of capabilities [21]. For Java, either 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 unshareable, you must either hide its type, or block all communication channels that could be used to leak it (effectively blocking all inter-program memory-sharing).

Type hiding 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 filesystem, the receiving program will not be able to see the `FileInputStream` type. 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. Still, an attacker might be able to trick trusted code into calling it with an enabled privilege on the stack. As discussed in section 3.3.7, visibility of these annotations is limited based on the principals associated with each stack frame. The system's access control matrix can 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 type hiding, privileges are established when the program is loaded. If those privileges are too strong, there is no way to revoke them later – once a class is resolved into a name-space, there is no way to unload it without unloading the entire name-space.

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 can-

not possibly use the privilege by accident. Also, there is a `disablePrivilege()` method which can explicitly remove a privilege before calling into untrusted code.

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 [7], 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.

Addressing this issue is a prerequisite to deploying *any* flexible security mechanism above Java. We believe this is tractable but difficult. Current Java systems have benefited from the extensive scrutiny received over the past few years, but more will be necessary.

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 legal 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 type hiding, 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 types. Likewise, capabilities could store a principal in a private variable.

This is all hypothetical, of course, since current browsers do not provide the tamper-proof logging necessary for trustworthy auditing. However, once available, our architecture 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 digitally signed mobile code arrives in the Web browser, the user can be queried whether they trust the signatory. Microsoft's Authenticode [26] follows a "shrink-wrap" model — once the software is installed, it has 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 only real difference between the three systems presented in this paper are *when* this dialog is presented to the user. The type hiding strategy necessarily must present the dialog box before the applet may begin running. The hiding configuration must be initialized before the applet can be loaded.

The stack introspection system and the capability system can defer the dialog until it is absolutely necessary. With the stack introspection system, the applet may begin running immediately with ordinary applet privileges. This will be sufficient for the needs of many applets and no user dialogs will be required. When an applet wants a restricted privilege, it may then ask for it by name, causing the user dialog only when necessary.

A capability system could be designed the same way. An unprivileged applet could be granted a capability to request other capabilities. When used, the same dialog box would appear, and new capabilities would then be returned to the applet.

In all cases, the user's security preferences can be saved persistently, to prevent repeated dialog boxes. Additionally, these persistent privileges could be pre-set 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 our three software systems. In all cases, we are assuming that the Java virtual machine uses a just-in-time (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. If stack introspection were used only to guard when a file or network connection is opened (an already expensive operation), then a capability for the open file or network could be returned which would need no security checks on read or write operations. While that specific file capability could leak, the general ability to open a file would still be contained.

Type hiding does not incur any overhead at runtime, nor do unmodified capability systems. However, all systems must pay 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 type hiding 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 type hiding 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-objects model similar to that of SRC network objects [4] or CORBA [38]. 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 [39, 14] (assuming all communication is suitable 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 to identify the remote caller. However, if the remote method invocation used a cryptographically authenticated channel (as provided by SSL [12] or Taos [42]), the remote identity may be useful. Gong [14] and van Doorn, *et al.* [40] describe implementations of this.

In the case of stack introspection, the local stub for the RPC could be labeled with the remote principal (again, provided via SSL or Taos). The local stub can then enable privileges on behalf of its remote caller.

In the case of type hiding, the local stub for the remote procedure call could be treated with the same hiding configuration as the remote principal might have.

In the latter two cases, no deep changes are necessary to adapt the systems to a distributed environment.

Actually building RPC systems using any of the mechanisms in Java is future work. Current Java RPC systems, with the possible exception of Electric Communities [9], have no provisions for flexible security.

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.

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 type-hiding 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 such a radical stylistic departure in API design.

Type hiding offers good compatibility with existing Java applets. However, for all its simplicity and static enforcement of policy, the sharing of common classes in the Java class library makes it difficult for one resource to be hidden without affecting other resources.

Extended stack introspection also offers good compatibility with existing Java applets and good separation of resources, but its complexity is troubling and and its implementation is very specific to Sun's Java virtual machine.

We believe the solution is to combine elements of these techniques. Type hiding 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.

6 Acknowledgments

Thanks to Jim Roskind, Raman Tenneti, and Tom Dell of Netscape for sharing many of the details of their design, including Netscape's support of Dan Wallach as a visiting member of the design and implementation group.

We also thank Li Gong for helping us solidify our ideas. Thanks to Marianne Mueller, Ben Renaud, Frank Yellin, and Mark Miller for many interesting conversations about our work. Steve Wallach, Simson Garfinkel, and Dan Boneh gave valuable feedback on drafts of this paper.

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

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

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

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

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

[5] BORENSTEIN, N. S. 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* (1994).

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

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

[8] DROSSOPOULOU, S., AND EISENBACH, S. Is the Java type system sound? In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages* (Paris, Jan. 1997).

[9] ELECTRIC COMMUNITIES. *The Electric Communities Trust Manager and Its Use to Secure Java*, Sept. 1996. `http://www.communities.com/company/papers/trust/`.

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

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

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

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

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

[15] GONG, L. New security architectural directions for Java. In *Proceedings of IEEE COMPCON '97* (Feb. 1997).

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

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

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

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

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

[21] KARGER, P. A., AND HERBERT, A. J. 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, May 1984), pp. 2–12.

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

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

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

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

[26] MICROSOFT CORPORATION. *Proposal for Authenticating Code Via the Internet*, Apr. 1996. `http://www.microsoft.com/intdev/security/authcode/`.

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

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

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

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

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

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

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

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

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

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

[37] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of 2nd ACM Symposium on Operating Systems Design and Implementation* (Oct. 1996), pp. 213–227.

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

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

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

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

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