A New Approach to Mobile Code Security

Dan Seth Wallach

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

January 1999

Abstract

This dissertation presents a novel security architecture called *security-passing style* and motivates its application to security issues that arise in mobile code systems such as Java. Security-passing style, and its predecessor, stack inspection, allow the system to capture the complex security relationships that occur when trusted and untrusted code are run together and interact closely.

Where traditional security architectures can answer general questions of the form “can subject *X* use object *Y*,” they fail when considering problems where one subject may be acting on behalf of another, or may be acting on its own behalf. These systems generally have neither the mechanisms to capture the full security context of a request nor the policies expressive enough to be able to resolve whether these requests should be allowed or denied. Issues such as these arise in mobile code systems, requiring new security mechanisms to address their security.

While a number of traditional security architectures, including capability systems and process-structured systems, can be adapted to the secure execution of mobile code, this dissertation describes an architecture that addresses these issues and does it using an efficient implementation that requires no special hardware or language runtime support. Security-passing style has a well defined semantics describing how it works and allowing for proofs of its soundness. These semantics also allow us to produce an implementation that has extremely low overhead (in principal, just over one instruction per method invocation) based on static analysis of the program to be run and dynamic caching to make common-cases execute faster.

Acknowledgments

It is an amazing thing that this dissertation exists, due in no small part to help and cajoling from a large cast of friends and colleagues. My advisor, Ed Felten, was an unending source of useful and pragmatic advice. Andrew Appel gave expert direction in the programming language aspects of my research and provided helpful intuition into analyzing my results. Appel also coined the phrase ‘security-passing style’ and helped give my system the name ‘SAFKASI’ (pronounced *saff-KAH-zee*, the security architecture formerly known as stack inspection). My Secure Internet Programming colleagues Drew Dean and Dirk Balfanz have also been a great help to me throughout my research. Section 5.2.3, on name-space management, describes work done by Balfanz. The four of us spent far too much time finding flaws in the security of the Java system and building sneaky applets to take advantage of those flaws. Thanks also to Ken Steiglitz for reading my dissertation and catching all the silly mistakes that managed to slip by everybody else.

In the course of doing my work, I have gotten all kinds of useful commentary from anonymous conference and journal referees. I also thank Martín Abadi, John Wilkes, Dave Gifford, Norman Hardy, Olin Sibert, and Mark Miller for their feedback and explanations of how traditional systems have worked.

The technique of stack inspection was invented during a summer internship I spent at Netscape and was designed and built with Jim Roskind, Raman Tenneti, and Tom Dell. Thanks also to Warren Harris, Nick Thompson, Jim Gellman, Bob Relyea, Tom Weinstein, John Hines, and Tara Hernandez for all their help making my internship and my code successful. I probably still owe Tara a bottle of tequila for breaking the build.

Along the way, stack inspection has seen the tug and pull of the industry’s Java battles. For all the flames and invective, many people gave me extremely useful feedback and clarifications of how things were really working. Thanks to Li Gong, Marianne Mueller, Larry Koved, Bob Blakley, Jeff Bisset, Mike Jerger, and Mike Toutonghi. Thanks also to Kenneth Zadeck and David Chase for their useful and entertaining perspectives.

To actually get security-passing style working, I eventually gave up on the bugs and strange behavior of Sun's JDK and turned to Kenneth Zadeck, David Chase, and Roger Hoover of NaturalBridge. They allowed me to use their company's unreleased Java compiler and runtime and provided invaluable debugging help, encouragement, and constructive criticism.

My implementation of security-passing style was built around Geoff Cohen's JOIE (Java Object Instrumentation Environment) library. Geoff's fantastic understanding of the intricacies of the Java class file format were invaluable to me. Thanks also go to Paul Martino, Greg Humphreys, and Benji Jasik of Ahpah Software for allowing me free use of their SourceAgain Java decompiler, which proved to be a fantastic debugging tool.

Thanks to David Dobkin for supporting me while I transitioned from computer graphics, my original academic specialty, to computer security. The Princeton CS department is a wonderful place, and I'll miss it and its exceptional facilities. Thanks to Jim Roberts, Chris Tengi, Chris Krantz, and Matt Norcross for putting up with my ranting and keeping everything humming along.

Finally, a special thanks goes to my parents and my sister for supporting me during my years in grad school. My father's library of ancient computer arcana was particularly useful.

Research in the Secure Internet Programming Laboratory has been funded by the National Science Foundation, by the Alfred P. Sloan Foundation, and by donations from Bellcore, Intel, Merrill Lynch, Microsoft, Sun Microsystems, and Trintech. I did portions of this work at Netscape Communications Corp.

To Mom and Dad

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 My Thesis

Type-safe language runtimes are sufficient to securely execute untrusted mobile code.

Whereas traditional security required intervention from the operating system — and hardware memory protection — to isolate and protect one program from another, modern type-safe languages allow separation and resource controls to occur within the context of a single operating system process.

Actually maintaining the integrity of the language runtime in the face of hostile code that wishes to compromise it has turned out to be remarkably difficult, as researchers have discovered a stream of security-compromising problems in Java and other such systems over the past several years.

Once type safety is addressed, the major remaining difficulty in a language runtime is determining the identity whose authority must be checked before performing a security-relevant operation. Every "applet" is running together in a single system heap, so the applets all share a common set of system classes responsible for various dangerous operations such as access to the network and file system. As each applet might have different privileges, the system classes must impose some form of access control.

This dissertation analyzes a wide range of available access control mechanisms on a number of well-established security criteria. In addition to a number of traditional mechanisms that have existed in numerous systems over the past thirty years, this analysis considered a mechanism called “stack inspection,” designed and implemented at Netscape by myself, Jim Roskind, and Raman Tenneti.

Stack inspection turns out to have a number of interesting properties, including that it can be built easily above any type-safe language runtime and that it extends naturally to support secure remote procedure calls. This dissertation contributes a new form of stack inspection called “security-passing style,” defined as a code-to-code transformation that may then execute on an unmodified language runtime. Security-passing style is proven to be equivalent to the original stack inspection system, having the same desirable security properties without interfering with optimizing compilers as many other mechanisms do.

In addition to defining security-passing style, this dissertation contributes an implementation and measures its performance using a recent optimizing Java runtime. Current Java compilers and runtimes still have a way to go before achieving the efficiency of hand-tuned assembly code, so we also consider what the overhead of security-passing style *should* be, given a hand-tuned implementation.

1.2 Structure of the Dissertation

This dissertation presents information which has appeared in a number of earlier publications [WRF96, WBDF97, WF98, DFW96, DFWB97], as well as previously unpublished material. This section briefly describes each chapter and cites its influences.

- The remainder of this chapter discusses traditional and pragmatic security measures as they are often implemented in the commercial world. This chapter points out how mobile code violates many of the assumptions made by traditional commercial security. This material borrows from [DFWB97].

- Chapter 2 defines what exactly we mean by "security" and quotes the relevant parts of the Orange Book [Nat85]. The Orange Book defines what security generally means for an operating system and defines specific features that a system should support in order to receive various security ratings.

- Since we are interested in security mechanisms for Java, chapter 3 gives some background on the Java system and describes its promises and pitfalls. Rather than presenting each and every flaw we found, this chapter presents a taxonomy of problems with examples drawn from our experience studying Java's security [DFW96, DFWB97].

- Many of the security mechanisms proposed for Java have been tried in one form or another in traditional operating systems. Chapter 4 describes how operating systems have traditionally build security architectures, both on local machines and across computer networks. This chapter reviews a broad array of prior art, from capability-based and ring-structured systems [Lev84, Org72, SS72] through modern innovations like proof-carrying code [NL96].

- Chapter 5 shows how traditional security architectures apply to Java and introduces some new mechanisms that have been successfully applied to Java.

We originally designed and built two such systems (name-space management and stack inspection) [WBDF97]. This chapter also discusses capability-based Java systems and process-structured systems [BTS+98, Cv98] and evaluates all four architectures against each other.

- We want to analyze stack inspection in more detail, but to do that we need a logical framework. Chapter 6 introduces a logic of belief, designed by Abadi, Burrows, Lampson, and Plotkin [ABLP93], which we use later in the thesis.
- Chapter 7 uses this logic to design security-passing style and prove its equivalence with stack inspection. Chapter 7 also discusses how security-passing style applies to remote procedure calls in addition to local ones. We originally presented this work in [WF98].
- Chapter 8 describes an implementation we built of security-passing style and quantifies its actual performance running benchmarks as well as its ideal cost, based on the minimum number of instructions we expect might be able to implement these primitives. This material has not been previously published.
- Finally, chapters 9 and 9.1 present future work and conclusions.

1.3 Traditional Computer Security

Any discussion of computer security necessarily starts from a statement of requirements (*i.e.,* what it really means to call a computer system "secure"). In general, secure systems will control, through use of specific security features, **access to information** such that only properly authorized individuals, or processes operating on their behalf, will have access to read, write, create, or delete information.

— *Trusted Computer System Evaluation Criteria* (the "Orange Book") [Nat85]

There is no easy way to define computer security. Different environments have radically different ideas of what security means to them. Military organizations are concerned with restricting sensitive information to those authorized to see it. Commercial organizations are often more concerned that data cannot be modified without sufficient authorization. In academia, the largest concern may well be that scarce resources are consumed only by authorized users. And, just about everybody would like to have enough of an audit trail to identify their attackers after the fact, and present evidence to the police. These are the four pillars of computer security: secrecy, integrity, auditability, and protection from theft of service.

In traditional systems, at least, computer security seemed to be straightforward to manage. If you limited who could log in to your system to people who you trusted, you were 99% done. Once inside the system, each user could be controlled in their ability to read and write files on the system. They could be billed for their usage of disk space and CPU cycles. Sure, some system utilities may have had bugs that users could exploit, but those were few and far between. It was not really all that important, since only employees could get anywhere near the computer, and they generally had it in their best interests not to attack their employer! Even in university environments, where some users may have been malicious toward their peers, information about system vulnerabilities was carefully guarded ("security through obscurity"), providing adequate security against all but the most determined attackers.

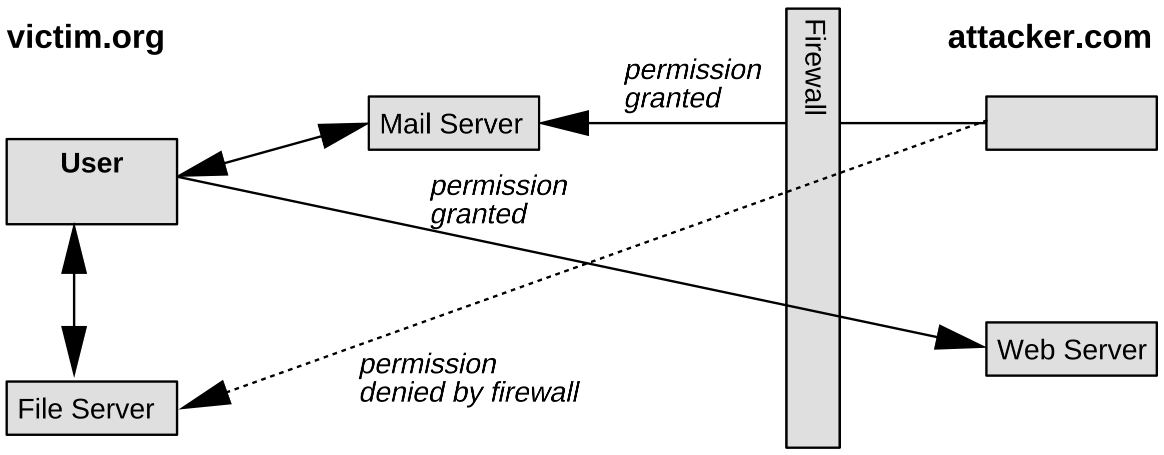


Figure 1.1: A typical firewall architecture.
Firewalls protect against traditional security attacks by blocking access to sensitive internal machines from the outside. Some machines will continue to allow connections, such as servers that must receive and process e-mail. Internal users are generally allowed to initiate connections to any machine on the Internet, or at least to any Web server.

In practice, the threat of attack from insiders was and is still a serious issue. In conversations I have had with early timesharing computer users, many have described discovering and exploiting security holes with the same pride one might describe finding a $20 bill on the street. The security of these timesharing systems relied as much on their users' lack of malice as on any of their security mechanisms.

1.4 Network Security

When organizations evolved from having central mainframes to networks of computers and those networks were eventually linked to the Internet, the rules began to change. Many organizations were fairly permissive in their internal security, operating under the assumption that their employees were not malicious, and that all internal machines were sufficiently well controlled that no normal use of the system utilities would result in a security problem. Of course, no organization could safely make such assumptions about arbitrary machines on the Internet. To safely connect a corporate network to the Internet, an organization would deploy *firewalls*, gateways between the inside and outside that would hopefully not hinder internal people trying to get their job done, yet would restrict external malcontents from wreaking havoc (see figure 1.1). While a number of different techniques exist for building firewalls [CB94], they all work fundamentally by blocking external attempts to connect to all but a handful of carefully chosen internal machines. Firewalls generally allow internal machines to make connections outside (either directly or through proxy servers), but external machines are generally allowed to connect only to a small number of internal machines.

If secure information must travel over an insecure network, or just over the Internet backbone, a very real danger is that the traffic might be *sniffed* — observed by a malicious third-party. This is an issue when a consumer wishes to buy goods from an online service like Amazon.com, and it is also an issue when a company wishes to connect to its field offices without expensive leased lines. All the deployed solutions involve *cryptography*, encrypting sensitive data before it goes on the network and decrypting it after it arrives safely. If the cryptographic protocol is well implemented, an attacker would be required to do an enormous computation in order to learn even a single bit of the encrypted message.

When properly deployed, firewalls and cryptography have the potential to reduce the network security problem to be no worse than the traditional security problem. Of course, the traditional security issues have not gone away.

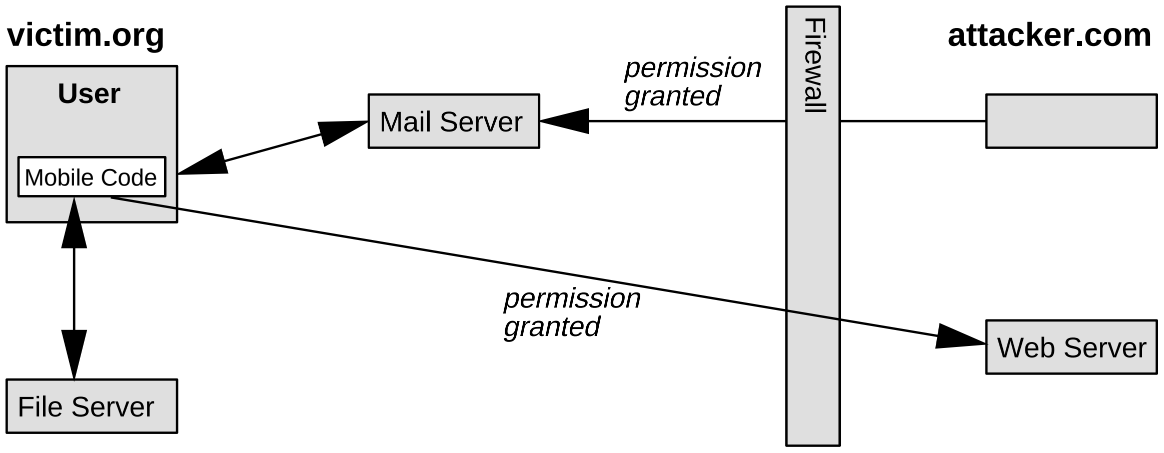


Figure 1.2: Firewalls do not protect against mobile code.

Once inside the network and running on a user's machine, the firewall offers no protection against the mobile code accessing sensitive internal data and perhaps leaking it to the outside. The security must happen at the user's machine, rather than at the firewall.

1.5 Mobile Code Security

The firewall model was quite a success commercially, and the firewall industry continues to grow today with numerous vendors fighting each other over features and prices. Unfortunately, the firewall model makes a fundamental error: it assumes that all the programs running inside the network are acting only on the requests of internal users and the data that passes through the firewall is inconsequential. After all, no internal user would accidentally choose to leak a sensitive corporate document to the open Internet. And, there should be no harm in letting them download arbitrary files from the network. If files are all plain ASCII text then there is clearly no danger. Even if a user downloaded a binary program, it had to be explicitly installed in the system. Because this process was relatively laborious, there was relatively lower danger of installing a dangerous program by mistake.

With the modern Internet, users are not just getting their documents and e-mail in plain text. Instead, documents are themselves programs or contain programs within themselves [Sib96]. Such documents are sometimes said to have "active content," containing exciting new gizmos like "scripting," "applets," "custom controls," "plugins," or other strange marketing phrases like "crossware." Fundamentally, they are all *mobile code systems* and they violate the assumptions made by firewalls. Previously, a firewall could assume that an attacker could only be on the outside. Now, with mobile code, an attack might originate *from the inside* as well, where a firewall can offer no protection (see figure 1.2).

Programs can arrive as attachments to e-mail messages or to Web pages. With mobile code technologies, the program can install itself and begin running with, at most, a single mouse click. Following that one click, it might be possible for mobile code, now running on a user's machine, to act as if it were the user and attack the corporate network.

Likewise, cryptography offers very little to help with mobile code. By using digital signatures, cryptography can be used to certify the *origin* of mobile code, and provide guarantees that the code received was not tampered with in transit. Cryptography can make no guarantees about what the code might do when executed.

In order to properly protect the users and their networks against attacks from mobile code, either all mobile code support must be turned off, with the consequent loss of functionality, or the users' platforms must maintain adequate safeguards to control the actions of mobile code. Turning off mobile code is certainly a simpler solution. Why not do it? The users will scream! If a company blocks its employees from seeing the latest snazzy Web plugins, they may be preventing their employees from interacting with suppliers and studying competitors. Like it or not, an increasing amount of "content" needs mobile code to be viewed properly.

In essence, the biggest concern with mobile code is that it might be a way to create *Trojan Horses*. When the invading Greeks wanted to sack Troy, they did it by building a large wooden horse as a gift for the Trojans and pretending to abandon their siege of the city. Unbeknownst to the Trojans, a number of Greek soldiers hid inside the horse, which the Trojans brought inside their walls as a trophy. At night, the soldiers emerged from their hiding place and opened the city gates to the returning Greek armies. In the context of computer programs, we call any program a Trojan Horse that has apparently or actually useful features and also contains hidden malicious functionality that exploits any privileges the program may have when executing.

1.6 Securing Our Future

As mobile code becomes increasingly a part of our computer infrastructure, we need to build systems that can safely run mobile code. A number of very broad issues will need to be addressed if we are to feel safe using our computers. Probably the single largest issue is known bugs in older existing systems. Most people who attack computers are not computer researchers discovering new vulnerabilities. Rather, they are skilled librarians, collecting known exploits like tools on a tool belt. An attacker only needs to learn what software their target is using and see if they have an appropriate exploit ready to go. While users can gain some assurance by always upgrading their systems to the latest software versions and applying security patches when available, much of the blame rests on the software vendors who continue to ship buggy code. So long as feature lists and ship dates are sufficient to gain market share and sell products, the effort to fix bugs or design software correctly from the ground up will not be rewarded in the marketplace.

One promising trend is the shift away from the C and C++ programming languages to Java. Java, along with Modula-3, Scheme, ML, and most other modern programming languages support *type safety* and *strong abstraction*. This means that, while programs may continue to have bugs, it is not possible for a program to accidentally reference memory it has not properly allocated or been given a reference to, nor is it possible to confuse one data type for another. Type safety is a strong enough property that it completely thwarts attacks that attempt to overflow internal buffers and overwrite the system's memory with hostile code. While it is possible to write secure code in C or C++, a great deal more effort is required because there are many more opportunities to make mistakes.

Unfortunately, “legacy code” will continue to run our systems for many years to come. In order to secure our future, we must realistically assess the legacy systems of our past and investigate systems that can retrofit security around them.

Chapter 2

What is Security, Really?

While this dissertation is about the security of mobile code systems, it is important to begin by talking about traditional computer security and what it means for a system to be secure. With systems as early as Multics [BL76, Org72, Sal74] and later in operating systems of all kinds and variations, vendors have worked to build secure systems. The landmark "Orange Book" [Nat85], published by the U.S. Department of Defense in 1985, sought to define common profiles for secure systems, helping vendors reach consensus on what a secure system should be, and then widely distribute such systems in the commercial as well as military marketplace. Based on research beginning with a task force in 1967, the DoD and its contractors wrote down everything they believed they knew about building secure software. The full series of books ("the rainbow books") covers everything from how software should be physically delivered to how version control should be managed. The Orange Book introduced six criteria around which all secure systems could be judged, and specified a number of classifications, ranging from "D" through "A1" to which a system would be evaluated.

According to the Orange Book, the basic criteria are:

Security Policy – There must be an explicit and well-defined security policy enforced by the system. Given identified subjects and objects, there must be a set of rules that are used by the system to determine whether a given subject can be permitted to gain access to a specific object. Computer systems of interest must enforce a mandatory security policy that can effectively implement access rules for handling sensitive (*e.g.,* classified) information…

Marking – Access control labels must be associated with objects. In order to control access to information stored in a computer, according to the rules of a mandatory security policy, it must be possible to mark every object with a label that reliably identifies the object's sensitivity level (*e.g.,* classification), and/or the modes of access accorded those subjects who may potentially access the object.

Identification – Individual subjects must be identified. Each access to information must be mediated based on who is accessing the information and what classes of information they are authorized to deal with. This identification and authorization information must be securely maintained by the computer system and be associated with every active element that performs some security-relevant action in the system.

Accountability – Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party. A trusted system must be able to record the occurrences of security-relevant events in an audit log. The capability to select the audit events to be recorded is necessary to minimize the expense of auditing and to allow efficient analysis. Audit data must be protected from modification and unauthorized destruction to permit detection and after-the-fact investigation of security violations.

Assurance – The computer system must contain hardware/software mechanisms that can be independently evaluated to provide sufficient assurance that the system enforces [**the above requirements**]**.** In order to assure that the four requirements of Security Policy, Marking, Identification, and Accountability are enforced by a computer system, there must be some identified and unified collection of hardware and software controls that perform those functions. These mechanisms are typically embedded in the operating system and are designed to carry out the assigned tasks in a secure manner. The basis for trusting such system mechanisms in their operational setting must be clearly documented such that it is possible to independently examine the evidence to evaluate their sufficiency.

Continuous Protection – The trusted mechanisms that enforce these basic requirements must be continuously protected against tampering and/or unauthorized changes. No computer system can be considered truly secure if the basic hardware and software mechanisms that enforce the security policy are themselves subject to unauthorized modification or subversion. The continuous protection requirement has direct implications throughout the computer system's lifecycle.

The security analysis of a system traditionally begins by studying its reference monitor. A *reference monitor* is defined to be the portion of code that checks each and every object reference and validates it against the system's security policy [And72]. In order to be trustworthy, the reference monitor must be tamperproof, always invoked, and small enough to be analyzed and tested. Reference monitors have traditionally been implemented as part of a *security kernel* [AGS83]. This security kernel, and any external utilities it depends on to enforce system security are referred to as the *trusted computing base* or TCB.

Security is still meaningless without a *security policy*. In commercial systems, it is often sufficient to have *discretionary access controls* (DAC) such as access control lists or Unix-style file permissions, although some commercial systems will also want the ability to build restrictions about which applications can access which data [CW87]. In contrast to the civilian sector, the military is very concerned about maintaining its multilevel security (MLS) — classified, secret, top secret, and so forth. A user with "secret" clearance should not be permitted to read a "top secret" document or even learn of the existence of such a document. Likewise, there should be no way for a user to "declassify" a document except through carefully limited procedures. The MLS policy has been formally modeled [BL73, BL76] and is an example of *mandatory access control* (MAC), so called because no user or program should be able to circumvent the policy, no matter how privileged. Numerous other security policies have been proposed, such as the Chinese Wall policy [BN89], which gradually builds a wall around users as they view sensitive information to prevent them from tainting other information considered to be compartmentalized. This may be applicable in "clean room" design environments as well as certain legal settings.

In order for a system to achieve the highest Orange Book ratings, it must have been *formally verified*. Formal verification has had its greatest successes in studying cache-coherence protocols [PD96], cryptographic protocols [BAN90, MMS97], cipher systems, hardware verification [HYHD95], and compiler design [GW95]. While some operating systems designers have attempted to create provably secure systems [NBF+80], to require this for operating systems in 1985 was quite ambitious and, even today, is considered somewhat impractical in commercial settings. As a compromise, the Orange Book is satisfied that a mathematical model of the system's security policy exists and has been proven to be secure. Similar proofs about the validity of the system's components would also help a system's rating by increasing our assurance of the system's correctness.

Chapter 3

Java Security: Web Browsers and Beyond

3.1 Introduction

The continuing growth and popularity of the Internet has led to a flurry of developments for the World Wide Web. Many content providers have expressed frustration with the inability to express their ideas in HTML. For example, before support for tables was common, many pages simply used digitized pictures of tables. As quickly as new HTML tags are added, there will be demand for more. In addition, many content providers wish to integrate interactive features such as chat systems, dynamic stock market charts, and other animations.

Rather than creating new HTML extensions, Sun Microsystems popularized the notion of downloading a program (called an *applet*) that runs inside the web browser. Such remote code raises serious security issues; a casual web reader should not need to be concerned about malicious side-effects from visiting a web page. Languages such as Java [GJS96], JavaScript [Fla97], Safe-Tcl [Bor94], Limbo [Com97], Phantom [Cou95], Juice [FK97] and Telescript [Gen95] have been proposed for running untrusted code, and each has varying ideas of how to thwart malicious programs.

After several years of development inside Sun Microsystems, the Java language was released in mid-1995 as part of Sun's HotJava web browser. Shortly thereafter, Netscape Communications Corp. announced they had licensed Java and would incorporate it into their Netscape Navigator web browser, beginning with version 2.0. Microsoft later licensed Java from Sun, and incorporated it into Microsoft Internet Explorer 3.0. With the support of many influential companies, Java effectively became the standard for executable content on the web. This also made it an attractive target for malicious attackers, and demanded external review of its security.

Drew Dean and I first looked at Java in November, 1995 [DFW96]. Since that time, the Princeton Secure Internet Programming group has found a number of bugs in Netscape Navigator through all its various releases and later in Microsoft's Internet Explorer. As a direct result of our investigation, and the tireless efforts of the vendors' Java programmers, we believe the security of Java has significantly improved since its early days [DFWB97]. In particular, Internet Explorer 3.0, which shipped in August, 1996, had the benefit of nine months of our investigation into Netscape's Java. Still, despite all the work done by us and by others, no one can claim that Java's security problems are over.

3.2 Java Semantics

Java is similar in many ways to C++ [Str94]. Both provide support for object-oriented programming, share many keywords and other syntactic elements, and can be used to develop stand-alone applications. Java diverges from C++ in a number of ways: it is type-safe, supports only single inheritance (although it decouples subtyping from inheritance), requires a garbage-collected memory heap, and has language support for concurrency, exception handling, and object persistence.

Java compilers produce a machine-independent bytecode. While a program can be executed on a local machine, it may also be loaded across a network. The bytecode is lazily loaded and dynamically linked as it is needed by the program's execution. A Java runtime system may either interpret the bytecode directly or compile it to native machine code [LY96]. Some newer Java systems statically compile and link whole Java programs at once [Nat98].

The standard Java distribution contains a large and growing collection of utility code for every purpose, from basic data structures (hash tables, vectors, queues) to string parsing routines, graphics and GUI functionality, networking and remote procedure call support, database connectivity, cryptography, internationalization and localization, and printing.

3.2.1 Packages and Modifiers

Java programmers may combine related classes into a `package`. These packages are similar to name spaces in C++, modules in Modula-2 [Wir83], or structures in Standard ML [MTH90]. While package names consist of components separated by dots, the package name space is actually flat: scoping rules are not related to the apparent name hierarchy. In Java, `public` and `private` have the same meaning as in C++: Public classes, methods, and instance variables are accessible everywhere, while private methods and instance variables are only accessible inside the class definition. Java `protected` methods and variables are accessible in the class or its subclasses or in the current (package, origin of code) pair. A (package, origin of code) pair defines the scope of a Java class, method, or instance variable that is not given a `public`, `private`, or `protected` modifier. Colloquially, methods or variables with no access modifiers are said to have *package scope*. Unlike C++, `protected` variables and methods can only be accessed in subclasses when they occur in instances of the subclasses or further subclasses. For example:

```
class Foo {
  protected int x;
  void SetFoo(Foo obj) { obj.x = 1; } // Legal
  void SetBar(Bar obj) { obj.x = 1; } // Legal
}

class Bar extends Foo {
  void SetFoo(Foo obj) { obj.x = 1; } // Illegal
  void SetBar(Bar obj) { obj.x = 1; } // Legal
}
```

The definition of `protected` was the same as C++ in some early versions of Java; it was changed during the beta-test period to patch a security problem[Mue96] (see also section 3.4.2).

3.2.2 Dynamic Checking and Static Verification

The Java runtime system is designed to enforce the language's access semantics. Unlike C++, programs are not permitted to forge a pointer to a function and invoke it directly, nor to forge a pointer to data and access it directly. If a rogue applet attempts to call a private method, the runtime system throws an exception, preventing the errant access. Thus, if the system libraries are specified safely, the runtime system is designed to ensure that application code cannot break these specifications.

Java's type safety can be mostly verified statically by the *bytecode verifier* which examines every class before it is loaded by the Java system. Unfortunately, the Java language has some features that prevent completely static verification. The type system uses a covariant [Cas95] rule for subtyping arrays, so array stores require run time type checks[1] in addition to the normal array bounds checks. Cast expressions also require runtime checks. In addition to their performance penalties, dynamic checks stretch the trusted computing base beyond the bytecode verifier. In practice, the dynamic verification of Java classes is remarkably subtle and bugs have been found regularly in the bytecode verifier [MF97, Sir97] and the class loading architecture [Dea97].

3.2.3 Security Mechanisms

The original version of Java distinguished *remote* code from *local* code. While local code was permitted to do do anything it wanted, remote code was restricted to the Java "sandbox" security policy, which roughly states that applets may not access the local file system at all and may only make network connections to their host of origin.

Since local code and remote code could co-exist in the same Java virtual ma-

[1]For example, suppose that `A` is a subtype of `B`; then the Java typing rules say that `A[]` ("array of `A`") is a subtype of `B[]`. Now the following procedure cannot be statically type-checked:

```
void proc(B[] x, B y) {
  x[0] = y;
}
```

Since `A[]` is a subtype of `B[]`, `x` could really have type `A[]`; similarly, `y` could really have type `A`. The body of `proc` is not type-safe if the value of `x` passed in by the caller has type `A[]` and the value of `y` passed in by the caller has type `B`. This condition cannot be checked statically.

chine (JVM), and can in fact call each other, the system needed a way to determine if a sensitive call, such as a network or file system access, was executing "locally" or "remotely," since the security policy allowed more freedom for local code. The original JVMs have two inherent properties used to make these checks:

- Every class in the JVM that came from the network was loaded by a ClassLoader, and includes a reference to its ClassLoader. Classes that 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, these two JVM implementation properties allow the security system to search for remote code on the call stack. The system would actually count how many "system" stack frames existed between the security check and the first "remote" stack frame. This value, called the *ClassLoader depth* was used in making a number of security decisions.

Later versions of Java replaced ClassLoader depths with a more general checking mechanism called *stack inspection*. This will be introduced in chapter 5 and formalized later in chapter 7.

To enforce the sandbox policy, all the potentially dangerous methods in the system were designed to call a centralized SecurityManager class that checks if the requested action is allowed (using the mechanisms described above), and throws an exception if the request violates the policy.

3.2.4 System Architecture

Because Java attempts to protect the local system from potentially malicious mobile code, and it also tries to protect one such program from another, Java begins to take on the appearance of an operating system instead of just a language. As an operating system, the most conspicuously missing feature of Java is the use of separate address spaces to separate processes. Instead, Java relies strictly on the memory safety it gets from being type safe. This may have important ramifications as computer architectures evolve and the proportional costs of traditional operating systems mechanisms grow worse [Ous90, ALBL91]. Still, the Java VM, by itself, does not provide *all* the guarantees of a traditional operating system, such as fair scheduling, resource usage limits, multiple users, and more. Some recent research projects have begun to address these issues [GWTB96, Dig97, MRR98, BTS$^+$98, TL98, Cv98, HCC$^+$98, BG98], but no commercial products yet offer a complete solution. These issues are discussed in more detail in chapter 5.

3.3 An Attacker's Methodology

Different attackers may have different goals. Some may wish to steal secrets from you. Others may wish to delete your files and crash your machine. Some may simply wish to be annoying. In order to build a secure Java system, it is necessary to understand how attackers will go about breaking the system. This dissertation is not meant to be a comprehensive list of every bug ever discovered in the Java VM (most of which were never published) nor of every possible kind of Internet-based security attack (although Howard [How97] has a nice summary), but instead presents examples we and others have found that demonstrate each category.

3.3.1 Overflowing Buffers

Popularized by the Morris Worm in 1988 [ER89], an extremely prevalent form of security attack has been to overflow a fixed buffer. A common practice among C programmers had been the use of `gets()` to read one line of text, terminated by a newline, into a buffer. Because `gets()` has no argument specifying the maximum length of its output, it will happily overflow buffers that were allocated with insufficient space. Similar issues occur with numerous C utility functions like `sprintf()`. Attacks like this have been aimed at every conceivable Unix and Windows utility. If the attacker knows something about where the buffer is allocated (either on the stack or on the heap), it becomes possible to write executable machine code directly into the system's RAM and (particularly when overflowing stack-allocated buffers) easily arrange for it to be executed.

Even when an application is written in a memory-safe language like Java, making it impossible to overflow buffers, the C runtime below Java may still be vulnerable. In early alpha releases of Java, we found numerous cases where Sun used fixed buffers and unsafe routines to write to them. Most of these problems were fixed in the beta releases.

While an operating system can largely address this class of attacks by requiring executable memory pages to be read-only, remarkably few operating systems actually do this.

3.3.2 Violating the Type System

The goal in violating the type system is the same as in overflowing a buffer: induce the system to execute arbitrary machine code and thereby work around any restrictions imposed by the Java environment. To violate the type system, we need a mechanism that allows an *unchecked type cast*. In C or C++, the programmer is free to cast any object type to any other type. In fact, C's `union` structure makes this fast and convenient. Java goes to great lengths to prevent this (see section 3.2.2) because unchecked type casts would allow an untrusted program to write a memory address into an integer, treat it as an object reference, and then write arbitrary data anywhere in memory.

Type system attacks often involve an attacker building a custom ClassLoader, which is officially against the sandbox security policy, yet a number of tricks have been discovered that have allowed them to be created anyway [MF97]. Since ClassLoaders are responsible for the consistency of the Java type system [Dea97], it is only natural that a malicious ClassLoader could arrange for an inconsistent type system. With this, it becomes possible to treat a reference of one type as if it is any other type in the system. Given this type caster, it is possible to write native code into memory and execute it, although it is tricky and unportable. An easier solution is to get a reference to a critical system class, such as `java.lang.System` and make security-compromising alterations. Just as an unchecked type caster can treat an integer as if it is an object reference, it can likewise treat a reference to a class with private members as if it is a reference to a class with all public members. A simple but effective attack is to set the SecurityManager to `null`, effectively disabling all system security and allowing a previously untrusted applet full access to the system.

3.3.3 Exploiting Library Weaknesses

The system libraries need to exercise a number of dangerous privileges that are normally denied to "sandboxed" applets. If you can pass unusual arguments to buggy system code, you may be able to make it act on your behalf.

One such attack focused on the way Java handles fonts. Font properties are stored in the the Properties subsystem along with a fair amount of privileged information, such as the identity of the user and other browser settings. Normally, an unprivileged applet is not allowed to read these privileged properties. However, if an applet asks to load a font called “user.name,” the value returned contains the restricted information. This bug was initially discovered in 1996, but still appeared as late as Sun’s JDK1.2beta4 release in mid-1998. The bug was fixed in later releases of JDK1.2.

This attack, reading the system properties through the font subsystem, is an example of an *indirect* attack. This class of attacks is extremely difficult to stop because the designers of each and every subsystem must be aware of the security implications of every line of code they write. One of the main goals of stack inspection (discussed later in this dissertation) is to thwart this class of attacks.

3.3.4 Denials of Service

Simple infinite for-loops or infinite memory allocation can destabilize and crash the Java virtual machine. Even allocating thousands of windows can also crash some window systems. Simply playing an annoying tune might drive users away from their machines. Mark LaDue has written a number of “hostile applets” that demonstrate these attacks [LaD96].

Denial of service attacks are fundamentally difficult to contain because, as far as the system is concerned, there is little difference between an applet that plays pleasant music and one that plays annoying noise. Similar issues apply to CPU and memory usage.

3.3.5 Spoofing the User

One of the most frightening attacks is one where Java is used to simulate the pop-up dialog boxes normally used to ask a user to authenticate themselves to a Web server, a dialup connection, or any other place where they might type a password or credit card number. Even though Java's graphics are constrained to be within the Web browser's internal window, it is still possible to draw a simulated window with simulated borders that can be dragged and resized! The only constraint is that it cannot be dragged outside the browser's internal window. This constraint generally is not sufficient to protect users.

Windows NT normally solves this problem by implementing a *trusted path* between the user and the password system. This is done by requiring the user to hit Control-Alt-Delete before an official password dialog will appear. NT will not let any application intercept Control-Alt-Delete, thereby guaranteeing the trusted path between user and authentication system.

Unfortunately, this works for NT login, but most other password systems have no such concept, particularly login screens for commerce-related Web sites, whether running on NT or any other system. Some recent commercial trends toward *single sign-on* might address this problem on the local system, but deploying such a system where it can run compatibly across the Web and across different operating systems will be a fascinating challenge.

3.4 Analysis

Through our studies of Java, beginning with the alpha releases of HotJava and continuing through the latest releases from Sun, Netscape, and Microsoft, we have found all kinds of security problems [DFWB97]. More instructive than the particular bugs we and others have found is an analysis of their possible causes. Policy enforcement failures, coupled with the lack of a formal security policy, make interesting information available to applets, and also provide channels to transmit it to an arbitrary third party. The integrity of the runtime system can also be compromised by applets. To compound these problems, no audit trail exists to reconstruct an attack afterward. In short, the Java runtime system is not a high assurance system.

3.4.1 Policy

The present documents on Netscape Navigator [Ros96b], Microsoft Internet Explorer [Mic97a, Mic97b], and HotJava [Sun95] do not formally define a security policy beyond the roughly stated “sandbox” policy. This contradicts the first of the Orange Book’s Fundamental Computer Security Requirements, namely that “There must be an explicit and well-defined security policy enforced by the system.” [Nat85] Without such a policy, it is unclear how a secure implementation is supposed to behave [Lan81]. In fact, Java has two entirely different uses: as a general purpose programming language, like C++, and as a system for developing untrusted applets on the web. These roles will require vastly different security policies for Java. The first role does not demand any extra security, as we expect the operating system to treat applications written in Java just like any other application, and we trust that the operating system’s security policy will be enforced. Web applets, however, cannot necessarily be trusted with the full authority granted to a given user, and so require that Java define and implement a protected subsystem with an appropriate security policy.

3.4.2 Enforcement

Java fundamentally bases its protection on the type safety of the language. An interesting issue is that the Java's bytecode representation is strictly more expressive than its source language; legal bytecode exists for which there is no equivalent Java source. Tools like *jasmin* make it easy to write any desired bytecode, legal or malicious [MD97]. Lindholm and Yellin [LY96] discuss many of the restrictions that are placed on bytecode and implemented by the Java bytecode verifier, but their specification is completely ad-hoc. An attempt to create an independent bytecode verifier [Sir97, SGB$^+$98] discovered a number of inconsistencies.

These inconsistencies are significant because the programmer writing a Java system class generally reasons about the code's security using the semantics of the Java source language. For example, a private variable may not be read or written by code outside of the class that contains the variable. If this property were enforced only by the compiler and not by the Java runtime, then hand-coded malicious bytecode would be able to freely attack the system. More complex properties relating to how a class calls its superclass constructor or when two classes are considered to be in the 'same' package are also relied upon for security, yet are not discussed in the Java language specification [GJS96]. If any one of these security-relevant properties turns out to be unenforced by the Java VM, then the code that relied upon it would be vulnerable to attack.

A more formal analysis of Java's bytecode may be found in Stata and Abadi [SA98], and the Java language's type system has been studied by Drossopoulou and Eisenbach [DE97a, DE97b]. More research will be necessary to address the safety of the language, in theory, and the bytecode restrictions, in practice.

Java's security must be enforced at a higher level as well. The Java "sandbox" security policy, such as it is, specifies certain restrictions on a Java applet's authority to make network connections, open files, and learn system properties. To enforce this, the Java `SecurityManager` is intended to be a reference monitor [Lam71]. Recall that a reference monitor has three important properties:

1. It is always invoked.
2. It is tamperproof.
3. It is verifiable.

Unfortunately, the Java SecurityManager design has weaknesses in all three areas. It is not always invoked: programmers writing the security-relevant portions of the Java runtime system must remember to explicitly call the SecurityManager. A failure to call the SecurityManager will result in access being granted, contrary to the security engineering principle that dangerous operations should fail unless permission is explicitly granted. It is not tamperproof: attacks that compromise the type system can alter information that the SecurityManager depends on. Finally, the SecurityManager code is the only formal specification of policies. Without a higher-level formal specification, informal policies may have incorrect implementations that go unnoticed. For example, the informal policies about network access were incorrectly coded in JDK 1.0 and Netscape Navigator 2.0's SecurityManager, allowing an applet, in collusion with a malicious DNS server, to connect to any computer on the network [DFWB97].

Furthermore, the original SecurityManager based many of its policies on a mechanism called "ClassLoader depths" (see section 3.2.3). In practice, this design proved insufficient. The ClassLoader depths made the SecurityManager impossible to analyze. Consider the code in figure 3.1. The Java authors identified two

```
/**
 * Applets are not allowed to link dynamic libraries.
 */
public synchronized void checkLink(String lib) {
  switch (classLoaderDepth()) {
    case 2: // Runtime.load clas
    case 3: // System.loadLibrary
      throw new AppletSecurityException("link", lib);
    default:
      break;
  }
}
```

Figure 3.1: ClassLoader depth excerpt from Java 1.0's SecurityManager. This method is used to check whether a request to load a new dynamic library should be granted. The ad-hoc checks of `classLoaderDepth()` are disturbing.

entry points from which applet code might request the dynamic loading of a library and encoded some fairly fragile information about these entry points in the SecurityManager. The security check is fragile because other system classes might also provide a way for an applet to load a library, directly or indirectly. These could possibly lead to other values of the ClassLoader depth.

One of the new features of Netscape 4.0, later followed by Microsoft's Internet Explorer 4.0 and Sun's JDK 1.2, was *stack inspection*, which was designed specifically to generalize and clean up the ClassLoader depth issues. This is described in detail in chapter 5. Stack inspection is actually a very interesting mechanism and its design, formalization, analysis, and performance is one of the main contributions of this dissertation.

3.4.3 Integrity

The Java runtime has a substantial amount of code written in C. Sun's JDK 1.0.2, for example, had 121K lines of C or C++ code compared to 107K lines of Java code.

This means that over half of the JDK is potentially vulnerable to buffer overflow attacks. We do not have access to current Java source, but the native code size has certainly grown larger in newer releases. If more of the Java runtime were written in Java itself, these potentially vulnerabilities would not exist.

In contrast, the architecture of HotJava (where the Web browser is implemented in Java and runs in the same Java machine as applets) is inherently more prone than that of Netscape Navigator or Microsoft Internet Explorer to accidentally reveal internal state to an applet because the HotJava browser's state is kept in Java variables and classes. Variables and methods that are `public` are potentially very dangerous: they give the attacker a toe-hold into HotJava's internal state. Static synchronized methods and public instances of objects with synchronized methods lead to easy denial-of-service attacks, because any applet can acquire these locks and never release them. These are all issues that can be addressed with good design practices, coding standards, and code reviews.

The Java system does not include an identified trusted computing base (TCB). Substantial and dispersed parts of the system must cooperate to maintain security. The bytecode verifier, and interpreter or native code generator must properly implement all the checks that are documented. The HotJava browser (a substantial program) must not export any security-critical, unchecked public interfaces. This does not approach the goal of a small, well defined, verifiable TCB. An analysis of which components require trust would have found the problems we have exploited, and perhaps solved some of them. More recent work has begun to address the TCB. This is detailed in chapter 5.

3.4.4 Accountability

The fourth fundamental requirement in the Orange Book is accountability: "Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party." [Nat85] The Java system does not define any auditing capability. If we wish to trust a Java implementation that runs bytecode downloaded across a network, a reliable audit trail is a necessity. The level of auditing should be selectable by the user or system administrator. As a minimum, files read and written from the local file system should be logged, along with network usage. Some users may wish to log the bytecode of all the programs they download. This requirement exists because the user cannot count on the attacker's web site to remain unaltered after a successful attack. The Java runtime system should provide a configurable audit system.

Solving this problem would appear to be an opportunity for a third-party developer, such as a firewall vendor. While a number of vendors sell products that purport to *block* applets, fewer offer support to *log* them, despite the relative ease with which it could be added to their products. Martin *et al.* [MRR97] and LaDue [LaD96] have shown that blocking is somewhere between difficult and impossible to accomplish. For example, an applet might be delivered over an encrypted channel. Without cooperation from the Web browser, the firewall would not be able to distinguish the applet from any other encrypted data. Perhaps simply logging all Java classes would be easier, and could provide sufficient evidence to reconstruct an attack and identify its source. If the JVM was forced to load its classes through a Web proxy server, that proxy server could host the tamper-proof log. If the JVM sent an extra message to the proxy server before loading any class (*e.g.*, "now loading x/y/z.cls from foo.zip"), the server would know to log its cached copy.

3.4.5 Denial of Service

The Java VM's that ship with Netscape and Microsoft's products make no attempt to control attacks that simply consume resources such as using too many CPU cycles or allocating too much memory. Several researchers have proposed architectures where Java applets run on distinct machines, separated from a user's Web browser, and displaying graphics remotely [SGB+98, MRR98, Dig97]. These systems fundamentally place no trust in the JVM itself and instead use OS-level protection to sandbox the entire JVM. Other researchers have proposed Java architectures that run separate instances of the Java machine within the same address space and can control the resource usage of each JVM instance [Cv98, BTS+98, TL98]. We will discuss and analyze these architectures in more detail in chapter 5.

Chapter 4

Building Secure Services

When we talk about a "secure system," we are generally referring to a system whose fundamental job is to provide a number of services with restrictions on who can use them and what they can do. A bank may allow customers access to information on their accounts. A subscription Web site may allow known subscribers to download the latest news and gossip. Generally, such systems are designed with a large database containing the state of the server, wrapped with Web servers with software that limits the kinds of requests and queries that can be made against the database (sometimes called a "three tier architecture," referring to the web client, the web servers with their "application logic," and a database system backing the web servers). Obviously, a bank does not want its customers to arbitrarily access the database and edit the amount of money present in their accounts!

A *secure service* is a layer of software (or hardware) that takes a dangerous primitive service, such as access to the bank's database, and exports a safe interface, such as an end user Web interface.

Whether a secure service is implemented in one computer or with a network of computers, the complete system must still be secure against attacks from the outside. Many of the techniques we might use to build security in single-computer systems apply as well to distributed systems [Rus81].

To build a secure service, we need a *security architecture*. The security architecture provides us with the primitives we use to express the *access controls*, or who is allowed to access what. In the abstract, all security policies can be expressed as an *access control matrix* [Lam71], stating for each *subject* and *object*, what kinds of accesses are permitted from that subject to that object. In practice, a number of different techniques have evolved to express the access control matrix. These are discussed below.

4.1 Traditional OS Architectures

Much of the work in security architectures has occurred in the pursuit of secure operating systems, particularly for time-sharing applications. With hundreds or thousands of users simultaneously sharing the same computer, the operating system is responsible for ensuring an equitable division of the machine's resources among its users, and it must do so with a minimum of overhead.

4.1.1 Processes

The Unix systems we use today as well as Multics and numerous other systems that came beforehand [Tan92] support the notion of a *process*. The process encapsulates a thread of execution, a separate address space in which it runs, and the privileges available to it. Processes are themselves prohibited from reading or writing directly to any of the computer's devices. Processes are likewise prohibited from directly handling interrupts or other hardware-related events. A process may read and write only to memory mapped into its address space by the kernel. In order for a process to interact with the outside world, it must either use a shared memory buffer or make a *system call*. A number of different mechanisms are used to implement system calls. Fundamentally, the kernel provides interfaces to read and write files, make network connections, allocate and free memory, interact with devices such as keyboards and mice, communicate with other processes, and so forth.

In a process-structured system, it is the kernel's complete responsibility to enforce any security policy the system may have. This generally means the kernel must very carefully check the arguments passed to every kernel call and make a determination if the request is allowed. The typical Unix security architecture attaches security labels to every object in the system (*e.g.,* every file). Every process runs with a user ID and a group ID. Based on these two ID's, the kernel will evaluate whether the process has sufficient privilege to access the requested resource. Traditional Unix systems label each file with a user and a group and have *permission bits* associated with each object. These permission bits, nine in all, state whether the user may read, write or execute the file, whether a member of the group may read, write, or execute the file, and finally whether anybody else may read, write, or execute. Some newer Unix systems as well as Microsoft Windows NT, Multics, and other systems support *access control lists* (ACLs), which provide a more general specification of access rights than Unix's permission bits. An ACL can express an arbitrary list of user and group ID's and state what permissions are granted to each of them. An ACL represents one row of the access control matrix – saying, for one object, all the subjects that may access it.

When the kernel is evaluating the permissions on an object, it must be careful in how the arguments are read and written from the user process; it would be unacceptable for a multi-threaded user processes to change the name of a file while it is being simultaneously validated by the kernel. This could result in the security portion of the kernel approving a file to be read only to have another file read instead. Likewise, the kernel must be careful in how it passes results back to a user process to avoid accidentally leaking the secrets of another process that may have previously occupied the same buffer in the kernel.

The traditional solution to these security issues is to always copy the data. The kernel will copy all string arguments before parsing them, therefore ensuring that the file validated to be read is the file that is actually read. Then, when data is actually read from a file or network, it is copied from the kernel's disk buffer back into the user process's memory.

In practice, the two largest sources of overhead in a process-structured system are copying data between address spaces and updating the system's page tables [Ous90, ALBL91]. Both of these operations are quite expensive and will adversely effect the I/O performance of applications. Extraneous data copying reduces system throughput by requiring multiple cycles of the CPU reading and writing the data to memory. This may also cause useful data to be ejected from memory caches. Extraneous page table operations may also require cache flushing in some virtual-memory mapped caches.

Operating system designers and system architects have considered a wide range of solutions to reduce these costs. Data copying can be reduced by careful buffer management and by mapping hardware devices directly into a user process's address space. To avoid forcing a cache flush on every context switch, some CPUs with virtually-mapped caches include a process-ID tag as part of the cache address. Alternately, single address-space systems [CLLBH92, CLFL94] place all processes into a flat address space. When such a system switches between processes, it need only change protection bits on the relevant pages.

Still, many applications, such as packet filtering, require extremely fine-grained interaction between untrusted user code and trusted system code. Modern operating systems have begun to look at injecting user code directly into the kernel, where costs can be even smaller. We discuss these systems later in section 4.3.

4.1.2 Ring Structures and Multilevel Security

Multics was a revolutionary system. It was structured as a series of concentric rings, with more privileged code getting closer to the center and unprivileged code on outer rings [Org72, Sal74, SS72]. Calls to inner rings were restricted, much like Unix system calls. The GE mainframe supported ring crossing calls with no additional overhead relative to normal procedure calls. This led to a system where device drivers could run in a separate ring from the rest of the kernel. This separation could isolate a fault in one kernel subsystem and prevent it from damaging the rest of the system.

The benefit of ring structures is that they naturally follow the layering inherent in an operating system design and provide nice separation and protection between layers. Still, problems arise in deciding what portions of the system to assign to what layers. Inner rings are allowed to freely read and write the memory of their callers in outer rings, so code closer to the center must necessarily be more trusted.

Multics also supported the U.S. military's multi-level security policy [BL76], where a document that is classified *top secret* may not be read by a user with only a *secret* clearance. Where access control lists provide *discretionary access control*, Multics additionally supported a form of *mandatory access control*.

4.1.3 Other Grouping Structures

Systems that support domain and type enforcement (DTE) allow grouping together of users and of resources, and allow policies to be expressed on these groups [BSS$^+$95]. DTE allows the expression of ACLs as well as multilevel mandatory policies. Likewise, role-based access control systems [SCFY94], allow privileges to be granted to *roles* rather than individual users. If a user is added to the group corresponding to a specific role, then the user may exercise the privileges associated with that role.

DTE and role-based systems can be thought of as adding one level of indirection to the access control matrix, simplifying the expression of policies by either grouping subjects or objects together and expressing policies against the aggregate groups.

4.1.4 Capability Systems

In Unix-style operating systems, a process may name any system object it wants, whether a file or a system device, and the kernel then centrally decides whether the process making the request is sufficiently privileged to make the request. In a capability system, however, a process will not necessarily be able to name any file or resource. Instead, the ability to access a resource is mediated by the ability to *name* it. Traditional machines supported two possible implementations of capabilities: special pointers can be stored in tagged memory where the hardware will prevent an untrusted process from changing the value of the pointer, or the kernel can supply "handles" or "descriptors" that allow an untrusted process to indirectly invoke its capabilities [Lev84].

A process will run in an environment defined by its capabilities. A user's privileges will be a group of capabilities that are inherited by programs run by that user. A number of subtleties involved in designing such a system are described in Hardy [Har85].

Plan9 [PPTT90] can be thought of as an interesting variation on a capability system. An untrusted process can be run in an environment where it is only presented with a subset of the full filesystem. Because all system resources in Plan9 are represented by files, they represent capabilities to access those system resources.

As ACLs can be thought of as capturing the rows of an access control matrix (saying, for each object, the list of subjects allowed to access it), capabilities can be thought of as capturing the columns of an access control matrix, where each subject holds a list of the objects that is it allowed to access.

There have been two traditional criticisms of capability-based systems: capabilities were relatively expensive to invoke, creating issues with system performance, and the *confinement* of privileges is problematic. The performance issue is relatively easy to envision: either your hardware must support relatively exotic semantics with tagged memory, or you must add a level of indirection between a process and its capabilities. Either will constrain the performance of a system. The confinement issue is much trickier: once a capability has been passed from a user to a program or from one program to another, there must be a mechanism to eventually *revoke* that capability. Without revocation, the capability can no longer be said to “belong” to the user to whom it was originally granted. While a number of extensions to capability systems exist to address these concerns [KL87], they all either increase the cost of invoking a capability or increase the cost of sharing one.

4.2 Distributed Systems

In a single machine, there are a number of guarantees that fail to be automatically true in a distributed system. For example, while all tasks on a single machine can be reasonably controlled, there may be a malicious machine on the same network as the distributed system. Such attackers may be able to passively listen to all traffic on the network or even inject their own messages that pretend to be from an authorized source. While cryptography and firewall technologies are commonly considered to solve all these problems, and are in fact necessary, they are not alone sufficient.

4.2.1 Firewalls

The simplest way to protect a clustered computer system is to restrict access to it through a firewall. In this way, every machine in the cluster will implicitly trust all messages it receives. This is probably the dominant method used in building secure Web services.

4.2.2 Distributed Capabilities

Capabilities extend quite naturally across a network. If remote object references include an unguessable string of bits (*e.g.,* 128 random bits), then a remote capability reference has all the security properties of a local capability [TMvR86, Gon89] (assuming suitable use of cryptography to protect the confidentiality and integrity of communication). Unfortunately, the confinement issues discussed in section 4.1.4 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 Kerberos [KN93], SSL [FKK96] / TLS [DA97], or Taos [WABL94]), the channel's remote identity might be useful. Gong [Gon89], Deng, *et al.* [DBWL95], and van Doorn, *et al.* [vABW96] describe implementations of this.

4.2.3 Distributed ACLs

Instead of capabilities, it is also possible to make access control decisions strictly based on the identity of the caller. When two machines establish a cryptographic connection, they will likely perform some kind of cryptographic handshake that proves to each party the identity of the other party [Sch96]. Many remote object systems, including DCE and CORBA [Hu95, DBWL95, Sie96], allow for access control lists associated with each object to be consulted when an object is invoked.

Taos [WABL94, LABW92, vABW96] is distinctive because it maintains *compound principals*. So, as a request passes from one process to another and from one machine to the next, the full chain of identities is maintained and can be passed along. Likewise, Taos has a strong notion of *delegation*, whereby one principal can make a statement that another principal can act on its behalf. This statement could be sent across an encrypted channel or digitally signed and sent in the clear. Delegation statements can become quite complex, so Taos uses a theorem prover to reason about them and resolve access control questions.

4.3 Mobile Code Systems

Modern research operating systems have invested a fair amount of effort in supporting mobile code. One of the important motivations for mobile code, particular injecting code from user processes into the kernel, has been to overcome the performance barriers inherent to crossing from user mode to kernel mode and back. In addition to the need for a security architecture, such systems also need a way to prevent the injected code from having arbitrary memory access to the kernel. So, just as Java relies on a bytecode verifier and then an interpreter or compiler, other mobile code systems have used interpreters, machine code rewriting [WLAG93], trusted compilers [BSP$^+$95, PB96, SESS94, SESS96], or machine-checkable proofs of code safety [NL96].

Many interesting security architectures have also been explored by these systems. Vino [SESS94, SESS96] exports a transactional interface to its kernel extensions. If the extension misbehaves, the transaction can be cancelled and its effects undone. Flux [FHL$^+$96] allows recursive virtual machines to manage each others resources. Scout [MP96] allows user and kernel code to cooperate in groups called *paths* that are scheduled together and may be assigned resource limits as a group.

4.4 Building a Secure Service

To understand the problem of building a secure service, consider the program that users may invoke to change their password. In general, users should have sufficient privilege to change their own password, but should not have sufficient privilege to change other users' passwords. However, some administrative users should have sufficient privilege to override a normal user password, which might be necessary when a user forgets his or her password. In a Unix-style system, the password-changing utility is generally “setuid root,” meaning it has sufficient privileges to do an arbitrary amount of damage to the system. A paranoid system administrator would need to carefully study the source of the password program, searching for bugs that might allow a user to convince the password program to perform a dangerous action on its behalf (see also the discusson on “the confused deputy” in section 5.2.4). In a DTE system or a system with ACLs, the password program may be granted a more fine-grained privilege to edit specifically the password file but no other. In a capability system, you could imagine each user being given a customized instance of the password program that would only allow changes to that specific user’s password.

One of the characteristics we look for in a security architecture is how easy it is to specify a fine-grained policy. Following the *principle of least privilege* [SS75], our system will be more secure when we can grant each program the minimum necessary privileges for it to get its job done. As we will see in the next chapter, each security architecture has pros and cons in how easily it can capture desired security policies.

It is important to point out that not all security policies can be strictly expressed as access control matrices. A security policy may be arbitrarily complex. It may involve constraints on the time of day a request is made. It might require the authorization of multiple users before a right can be exercised. To capture arbitrarily complex policies, we need arbitrary code that can be trusted to implement the policy. We call this a *trusted subsystem*. The Unix password utility is one example of such a system. Clark and Wilson [CW87] discuss the concept of trusted subsystems in great detail, and suggest that secure systems in the commercial world are fundamentally built from trusted subsystems rather than traditional access controls.

4.5 The Bad Guys Are Out There...

Many attackers use tools like COPS [Far93] or SATAN [FV93], which automate the process of checking for known bugs in remote network systems. These freely available tools, as well as commercial tools such as ISS's Internet Scanner [Int98], are designed to help systems administrators audit their own networks, but are equally useful to an attacker.

Collections of individual attacks, such as Rootshell [Kno97], are also available from which potential attackers may download exploits.

Once an attacker has broken into the machine, a common next step is to install a back door to allow them back in more easily in the future. Software that implements sophisticated back door functionality, hiding the back door from the user of a system by replacing system utilities that might be used to notice it, is freely available for both Unix and Windows [O'B96, Cul98].

4.6 Why Secure Services are Hard To Get Right

One would like to believe that, with so many ways to build a secure system, that one of them must actually work. Certainly, it must be possible to build a system that can do everything it is supposed to do without having any security problems. Instead, we have anecdotal stories going back to the dawn of timesharing systems and articles on the front-pages of newspapers detailing security failures. Unfortunately, several forces seem to be pushing vendors that conflict with security.

From the companies developing software to those deploying their systems, there seems to be a strong preference for additional features instead of fewer bugs. From users to technology executives, from salespeople to advertisers, features sell products. Across the industry, development resources appear to be increasingly funneled into new features rather than maintenance on older code.

The complexity of commercial software is growing rapidly. The source code for Microsoft Windows NT 5.0 is rumored to contain over 50 million lines. In a system that large, it becomes impossible for any one developer to be aware of the whole system and every conceivable interaction among components. This leads to unforeseen control flow and data dependencies that may cause one module to violate the invariants held by another module. Normally, that would lead to a bug that should be caught in normal quality assurance testing of the product before it ships. Security-related bugs, however, are different because they do not necessarily cause quality assurance tests to fail or raise other alarms. Instead, the system may be quietly permitting something that it should not. Security bugs thus can lie within shipping software for years before being discovered.

Several software engineering techniques can be used to address the code bloat. When code is written in a type-safe language, including Java, Modula-3, ML, and Scheme but excluding C or C++, buffer overflow bugs largely disappear (see section 3.3.1). Likewise, if programs were structured with a security kernel to reduce their TCB (see chapter 2), large sections of the system that have no need to be trusted could happily run with lower privilege.

Even if our systems vendors had infinite development resources to devote to the best design and debugging their software, we would still have insecure systems. Security requires serious effort on the part of systems administrators to properly install and configure their systems. Security also requires users to follow safe practices. One user installing a malicious (or virus-infected) program from the Internet could foil all the efforts of their administrators to protect their domain.

Given the increasing sophistication of attacker's tools combined with the increasing complexity of commercial software, it becomes difficult to reach anything beyond a pessimistic conclusion about security in the future. We hope the increasing attention paid to software security and software engineering will result in more sophisticated techniques (such as those espoused in this dissertation) being used in industry. Likewise, with the market for security products such as firewalls growing steadily, application and operating systems vendors may see security as a way for their own products to stand out.

Chapter 5

Security Architectures for Java

In Sun's original Java system, there was no real security architecture. Instead, the Java system relied on a number of ad-hoc mechanisms from which to build secure services. A number of techniques have been proposed to address this concern, many of which have been borrowed from traditional operating systems (detailed in chapter 4).

In this chapter, we will describe how a number of security architectures for Java have been implemented and will evaluate them against a number of criteria relevant to security, performance, and flexibility. In the case of mobile Java applets, all of these architectures allow the traditional Java sandbox to be extended when the code is "trusted." Generally this trust is based on digital signatures that have been applied to the code before it was shipped.

5.1 Common Underpinnings

To implement a flexible security policy, steps must be taken to identify the "source" of any program, either based on network addresses or digital signatures. The system must then decide what privileges are appropriate, based on input from either the user or system administrator. Any flexible system will have these same basic requirements.

5.1.1 Digitally Signed Code

Microsoft Authenticode [Mic96] popularized the idea of applying digital signatures to programs, likening the digital signature to traditional software distribution in shrink-wrapped boxes. When a user purchases a shrink-wrapped program in a store, the user decides whether to trust the program before installing it based on the reputation of the program and its vendor. To guarantee the legitimacy of a program, vendors take numerous measures including holographic packaging and finely printed "certificates of authenticity." Authenticode allows shipping the program across a network rather than in a box. The packaging is replaced with a digital signature. The identity of a program's signer forms the basis of a trust decision for the user: do you trust Company X to *endorse* potentially dangerous code that you run on your computer? The digital signature also guarantees that the code that was downloaded is the same as the code that its endorser originally signed. The endorser does not necessarily have to be the same as the original developer of the program.

Digital signatures can provide the same benefits and opportunities for Java. However, Java offers the opportunity to give more fine-grained privileges than the all-or-nothing policy used in Authenticode. We wish to provide different privileges to different sources of code. We can consider the identity of the code signer, seen as a public key signed by a certification authority, to be a *principal* in much the same way that every user of a traditional operating system is also a principal[1]. Then, the system's security policy can maintain a table listing which principals are allowed to endorse access to which resources, based on the desires of the system's user and administrator.

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. Netscape Communicator 4.0 (NS 4.0) solves this with an algebra to combine permissions [Net97]. Microsoft Internet Explorer 4.0 (MSIE 4.0) and the Sun Java Developer Kit 1.2 (JDK 1.2) only support a single signature per program [Mic97a, GS98].

While all the Java systems maintain a table of which principals are allowed which privileges, MSIE 4.0 also allows a digital signature to endorse a Java program for only a specific set of privileges. This allows an endorser to say "I certify this program is safe to use unrestricted networking, but not file system access" rather than simply "I certify this program is safe." The Jar (Java archive) file format [Sun96, Net96], used by NS 4.0 and JDK 1.2, does not currently support limited privilege endorsements in its digital signatures.

5.1.2 Administration

In Java, a critical issue with the system's security policy is how to help non-technical users make security-relevant decisions about who they trust to access particular resources. One strategy to simplify the user interface, used in NS 4.0, pre-defines groups of common privileges and gives them user-friendly names. For example,

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

a “typical game privileges” group might refer to specific limited file system access, full-screen graphics, and network access to the game server. Another strategy, used by MSIE 4.0, is to classify principals together into groups, called *security zones*. These grouping strategies reduce the number of dialog boxes presented to a user and make the dialog boxes simpler than deciding whether each individual principal may be granted each individual privilege.

An interesting issue is *when* to ask the user to grant privileges to a principal. NS 4.0 leaves this decision up to the applet. An applet may request any privilege at any time. This would allow an applet to begin running in a demonstration mode, for example, and later ask the user for the privilege to save work to the disk. MSIE 4.0, on the other hand, requires the request for privileges to be encoded directly into the digital signature, forcing any user dialogs to occur before the applet begins running. While this allows a single user dialog to specify permissions at once, avoiding “dialog fatigue,” it also forces an applet to ask for every possible permission it might *ever* need, since it will never get a chance to ask for a privilege after beginning execution.

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 access to the Web browser’s policy mechanism either to pre-install and “lock down” all security choices or at least to pre-approve applications used 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 NS 4.0 and MSIE 4.0 have extensive support for centralized policy administration in their Web browsers. While a sophisticated user 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.

5.2 Architectures

In chapter 4, we described a number of traditional security architectures used in traditional operating systems. Many of these have, in fact, been applied to Java. The architectures we will describe are:

Processes Several researchers have proposed models analogous to the process model used in Unix as a way of managing resources consumed by Java programs.

Capabilities A number of traditional operating systems were based on unforgable pointers that could be safely given to user code. Java provides an efficient environment for implementing capabilities.

Name space management An interesting property of Java's dynamic linking 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.

Extended stack inspection The original Java method of searching the stack for unprivileged code can be extended to include code sources on the call stack.

5.2.1 First Approach: Processes

Processes are a seemingly attractive technique to build a Java security architecture. Much as Safe-Tcl runs untrusted scripts in a separate interpreter [LDOW98, Bor94], Janos [TL98], Alta [BTS+98], and the J-Kernel [HCC+98] run each Java applet in a separate instance of the Java virtual machine, although multiple such VM's will run in the same native operating system process. Alta allows memory to be shared across applets, whereas the J-Kernel requires applets to communicate with remote procedure call mechanisms. GVM [BTS+98] runs applets in effectively separate processes, although allows limited memory sharing. JRes [Cv98] uses bytecode rewriting to instrument Java's memory allocation primitives. This allows memory and CPU usage to be "charged" to the applet responsible for that usage, even when multiple applets are running in the same virtual machine.

Taking process separation a step further, the Digitivity Cage [Dig97], the Java Playground [MRR98], and Kimera [SGB+98] dedicate a separate physical machine for running untrusted Java applets and use remote procedure calls to display graphics inside the user's Web browser.

The main benefit of a process model is that resources allocated by each applet can be tracked exactly. If an applet is to be terminated, its threads can be immediately terminated and its entire heap can be deallocated at once. Because no two virtual machines have direct references to each other's data, there is no concern that a VM may contain a pointer to garbage. Instead, indirect references are used in exactly the same way as references to objects on physically separate machines [BNOW95, Sie96].

When a potentially dangerous primitive is to be used, a message must pass from the untrusted VM to a trusted "system" VM, which must then decide if the sender of the message is sufficiently trusted to grant access to the desired resource.

The two largest drawbacks to process models are the overhead of marshaling and copying data across processes, as well as the general overhead associated with running a distributed object system, even though all object references may be within the same physical machine. Microbenchmarks show that cross-process calls can be quite expensive. In [BTS+98] the authors measured the performance of various IPC mechanisms, using the Kaffe virtual machine [Tra98]. A normal method invocation takes approximately 0.16μs whereas an inter-process call takes anywhere from 2.7μs (for the J-Kernel) to 57μs for GVM. No benchmarks are presented for running real applications, but if these interprocess calls are in the critical path, the performance impact would be substantial.

5.2.2 Second Approach: Capabilities

In many respects, Java provides an ideal environment to build a traditional capability system [Fab74, Lev84]. Electric Communities [Ele96] and Sun [Gol96] have implemented such systems. This section discusses general issues for capabilities in Java, rather than specifics of the Electric Communities or Sun systems.

Dating back to the 1960s, hardware and software-based capability systems have often been seen as a good way to structure a secure operating system [WCC+74, NBF+80, Har85, TMvR86]. 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 invoking the service associated with another capability. Once a capability has been given to a program, the program may then use the capability as often as it wishes and may pass the capability, or a subset of the capability, to other programs, although some systems take steps to control the propagation of capabilities. This leads to a basic property of capabilities: any program that *has* a capability is *assumed* to have been permitted to use it.

Capabilities in Java

Early computer systems used a number of techniques to support capabilities [Lev84]. Some systems used tagged memory to differentiate capabilities from other data types and protect them from tampering. Other systems stored capabilities in the kernel, providing a limited interface to user programs. Unix file descriptors are an example of such capabilities. In Java, a capability is simply a reference to an object. As early systems used tagged memory to prevent the forgery of memory addresses, Java uses type safety to prevent the forgery of object references. It likewise blocks access to methods or member variables that are labeled `private`, allowing an object to hide data from programs that have references to it.

The current Java class libraries already use a capability-style interface to represent open files and network connections (`InputStream` and `OutputStream`). However, static method calls and class constructors 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, the initial capabilities would be passed as arguments to the program (or perhaps stored inside the `Applet` object). For example, if an applet wished to open a file, it would be forced to call `Applet.getFileSystem()`, which would consult the system's security policy and then either return a capability to access the file system or nothing at all. Calls to the `FileInputStream()` constructor and similar classes would need to be completely forbidden, since a pure capability model does not provide any machinery for the constructor to determine who called it.

Interposition

Since a capability is just a reference to a Java object, the object can implement its own security policy by checking arguments before using them to initiate a dangerous operation. A capability's object can easily contain another capability as a private member and invoke the internal capability according to its own security criteria. As long as both objects implement the same Java interface, the capabilities can be used interchangeably.

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 5.1. 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 that 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 `FileSystem` interface.

To apply a flexible security policy to Java, the `FileSystem` instance returned by `Applet.getFileSystem()` would vary with the privileges granted to the applet. If the security policy called for all file access to be logged, or if it called for interaction with the user, all such behavior would be written as a capability and the applet would use this new file system capability in exactly the same way as it used the original. Because both capabilities implement the same interface, they can be used interchangeably.

An important issue for capabilities in Java is compatibility with existing code. For example, removing the usual `FileInputStream` constructor from the class libraries would constitute a major change to the standard Java class interfaces. Sim-

```
// In this example, any code wishing to open a file must first obtain
// an object that 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 -- Java semantics restrict creation
// of these objects to other code in the same Java 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);
    }
}
```

Figure 5.1: Interposition of a restricted file system root with capabilities. Both `FS` and `SubFS` implement `FileSystem`, the interface exported to all code that reads files.

ilar issues apply with other major system services. Additionally, class constructors for the capabilities themselves must be carefully controlled. In figure 5.1, the `FS` constructor is marked "package" scope, blocking the constructor from being called by arbitrary classes. This provides protection only if an applet cannot declare its classes to be in the same package as the capability. Current JVMs treat system packages (those beginning with `java`, `sun`, `sunw`, `netscape`, and `com.ms`) specially, and also enforce separation when the call crosses from applet code to system code.

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.

5.2.3 Third Approach: Name-Space Management

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

As part of our study in Java security architectures [WBDF97], Dirk Balfanz implemented a full system based on name-space management, as an extension to both MSIE 3.0 and NS 3.0. His implementation did not modify the JVM itself, but changed several classes in the Java system libraries. The full system is implemented in 4500 lines of Java; much of the code manages the graphical 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 the implementation in detail.

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

Table 5.1: Different principals see different name spaces.
In this example, code signed by Alice cannot see the `File` class, and for Bob the File class has been replaced with a compatible subclass. The columns in this table represent name-space "configurations."

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 that is compatible with the original. This technique is used in Safe-Tcl [LDOW98, Bor94] to hide commands in an untrusted interpreter. Plan 9 [PPTT90] can similarly attach different programs and services to the file system seen by an untrusted process.

In an object-oriented language, classes represent resources we wish to control. For example, a `File` class may be used to access the file system and a `Socket` class may be used to access networking operations. If the `File` class, and any other class that 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 that did not exist at all; an error or exception would be raised.

To implement an interesting security policy for an applet, we can create an environment that replaces sensitive classes with compatible ones that check their arguments and conditionally call the original classes. We refer to an environment that maps class names to their implementations as a *configuration*. Since every class can potentially have a different name-space configuration, we can arrange for the new classes to see the original sensitive classes, and we can block applet classes from seeing the same sensitive class. For example, the `File` class could be replaced with one that prepended the name of a subdirectory (see table 5.1), much like the capability-based example in figure 5.1. In both cases, only a subtree of the file system is visible to the untrusted applet. In order for the program to continue working correctly, each substituted class must be compatible with the original class it replaces (*e.g.,* the replacement must be a subclass of the original and must be careful to never return an instance of the original class).

Name space management can be compared directly with a traditional capability system. If different programs see different versions of the system classes, the effect is identical to those programs being given different capabilities to system resources, with the exception that while a program may pass a capability to another program, it cannot pass its name space.

An interesting property of this design is that all security decisions are made statically, before an applet begins execution. By avoiding runtime security checks, the design does not hinder high-performance implementations.

Implementation in Java

Name space management in Java is accomplished through modifying the Java *ClassLoader*. A ClassLoader is used by the JVM to provide the *name* $\rightarrow$ *implementation* mapping. Every class keeps a reference to a ClassLoader that is consulted for dynamic binding to other classes in the Java runtime. Whenever a new class is referenced, the ClassLoader of the referencer provides the implementation of the new class (see figure 5.2). So, ClassLoaders will be used to implement the name-space configurations in our design.

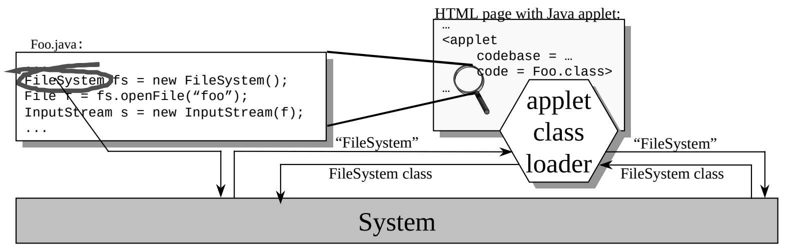


Figure 5.2: Behavior of a normal ClassLoader.
A ClassLoader resolves every class in an applet. If these classes reference more classes, the same ClassLoader is used.

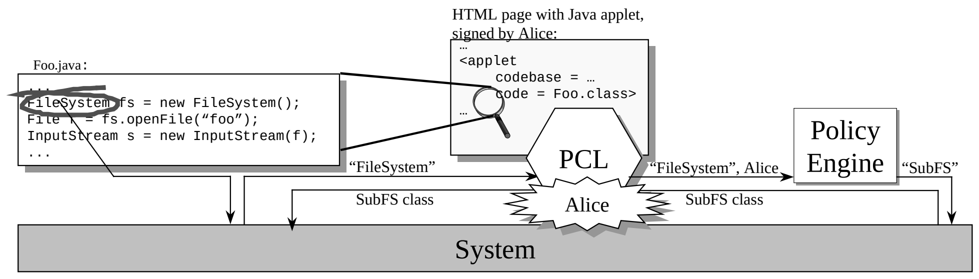

HTML page with Java applet, signed by Alice:
…
<applet
codebase = …
code = Foo.class>
…
Foo.java:
...
FileSystem fs = new FileSystem();
File f = fs.openFile("foo");
InputStream s = new InputStream(f);
...
PCL
Alice
Policy Engine
"FileSystem"
"FileSystem", Alice
"SubFS"
SubFS class
SubFS class
System

Figure 5.3: Changing the name space with a PrincipalClassLoader.
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.

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

    // Note that System.getPrincipal() would not be available in a system
    // purely based on name-space management.  In the current
    // implementation, getPrincipal() 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.getPrincipal().getName() + "/" + path;
    }

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

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

Figure 5.4: Interposition in a system with name-space management.
We assume that at runtime we can find out who the currently running principal is.

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.net.Socket`) that 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 get a different class because each applet's AppletClassLoader looks to separate locations for class implementations.

Our implementation works similarly, except that we replace each applet's AppletClassLoader with a PrincipalClassLoader that imposes the configuration appropriate for the principals attached to that class (see figure 5.3). 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 5.4 shows a class, `security.File`, that can replace `java.io.File`. When an applet calls `new java.io.File("foo")`, it will actually get an instance of `security.File`, that is compatible in every way with the original class, except it restricts file system access to a subdirectory. Note that, to achieve complete protection, other classes such as `FileInputStream`, `RandomAccessFile`, and so forth need to be replaced as well, as they also allow access to the file system. Likewise, there should be no way for an applet to ask for the superclass of `java.io.File`, thus getting a handle to the real File class instead of its renamed version. This would require some modifications to Java's class reflection, where any applet can call `Class.getSuperclass()`.

New Java features such as *reflection* (a feature introduced in JDK 1.1 that allows dynamic inquiry and invocation of a class's methods) could also defeat namespace management entirely unless the reflection system was specifically designed to respect the renaming. Also, common super-classes that are shared among separate subsystems would be difficult to rename without introducing incompatibilities (*e.g.,* both file system and networking use the same `InputStream` and `OutputStream` super-classes).

The Future of Name Space Management The ability to transparently replace a class with another class, as name-space management does, is useful only if the applet code continues to type-check properly after the replacement. Though there is no problem in this regard in NS 3.0, MSIE 3.0, and Sun's JDK 1.1, the semantics of dynamic linking in Java have changed with JDK 1.2 [LB98]. Though the change in JDK linking semantics is sound and well-motivated, it appears to have the side-effect of preventing many instances of transparent class-name replacement that worked in previous versions of the system. For example, if class `C` has a method `M` whose return type is `C`, and name-space management replaces `C` by `D` (a subclass of C) in some applet, calls to `M` in the applet will fail to type-check — the byte-code verifier will report them as type errors. The issue of compatibility of rewritten classes is discussed in detail (in the guise of class version upgrades) in the Java Language Specification [GJS96, chapter 13] and in Drossopoulou *et al.* [DWE98]. One possible solution is to rewrite classes as they are loaded [MRR98]. This would still require runtime support for Java reflection.

Despite the limitations on replacement of classes, *hiding* classes continues to be a sound and useful technique, and one that can easily be combined with other security mechanisms, so name-space management remains a valuable tool.

5.2.4 Fourth Approach: Extended Stack Inspection

This section presents an extension to Java's original stack inspection mechanism which was described in section 3.2.3. Variations on this approach are taken by NS 4.0[2] [Net97], MSIE 4.0 [Mic97b], and Sun JDK 1.2 [GS98][3].

[2]This approach is sometimes incorrectly referred to as "capability-based security" in vendor literature.

[3]At Netscape, I participated in the design and implementation of NS 4.0's Java security architecture along with Jim Roskind and Raman Tenneti in the summer of 1996. The Microsoft and Sun

Stack inspection has very little prior art. In some ways, it resembles dynamic scope of variables (where free variables are resolved from the caller's environment rather than from the environment in which the function is defined), as used in early versions of LISP [MAE$^+$62]. In other ways, it resembles the notion of *effective user ID* in Unix, where the current ID is either inherited from the calling process or asserted through an explicit *setuid* bit on the new program.

Java stack inspection's roots lie in the original Java system's use of "Class-Loader depth," discussed in more detail in section 3.2.3. This section will describe how it has been extended to support flexible security by each of the major Java vendors. Performance figures will be presented in section 8.3.

Simple Stack Inspection and the Confused Deputy

To explain how extensible stack inspection works, we will first consider a simplified model of stack inspection which resembles the stack inspection system used internally in Netscape Navigator 3.0 [Ros96a].

The problem faced by Netscape was what Norman Hardy calls *the confused deputy problem* [Har88]. Hardy describes a situation where the system Fortran compiler wished to log various statistics about its usage. The compiler was granted write-privileges to a directory which contained the statistics file, but also contained a file used to log system billing information. While no user had sufficient privileges to directly access the billing file, the compiler did, by accident. Asking the compiler to output debugging information to the billing file was sufficient to destroy the system's billing information. The Fortran compiler, granted a very broad privilege, could be tricked into abusing that privilege on behalf of any user. This

architectures were designed afterward and have different implementations while sharing the same basic structure.

problem was solved by granting the compiler a specific capability to access its statistics file, rather than editing access control information in the filesystem.

Early versions of Java suffered from similar problems. Applets needed to be restricted in their access to system resources, yet many system classes had privileges that were significantly more broad than was ever granted to applets. The simple stack inspection model addresses these concerns by allowing system code to optionally "enable its privileges". Before any file is opened or other restricted operation is performed, the system will check to make sure these privileges have been, in fact, enabled. If not, the system reverts to the restrictive applet security policy.

Fundamentally, stack inspection relies on being able to reconstruct the call stack and identify the source of every method or procedure on that stack. This generally relies on information already available in the runtime to support debugging, although language runtimes may need to do additional work in the presence of optimized code [Hen82, TA90].

To implement simple stack inspection, the operation of enabling privileges causes a flag to be set on its caller's stack frame. When the privileged stack frame returns, the privilege flag disappears with the rest of the stack frame. When privileges are checked, a search begins at the most recent stack frame and continues outward. If the search discovers the flag, privileges are granted. However, there might be a case where system code enabled a privilege then called back to untrusted applet code. This might occur in the event-dispatching loop of user-interface code. However, we do not want this event-handler to be able to take advantage of the privilege flag preceding it on the stack. Netscape calls this a *luring attack* and addresses it by considering the privileges of each stack frame as it performs the search. If the search discovers the stack frame of untrusted code before it reaches a flag set by trusted code, the search is terminated early and privileges are denied.

In Hardy's example, the Fortran compiler would enable its privileges before writing to its statistics file, but would not enable privileges when doing its normal output.

Stack inspection satisfies the *principle of least privilege* (see Section 5.3.4), allowing the Java system to operate with less than its full privileges active at all times and thus reducing exposure to attacks. This proved extremely useful in NS 3.0 [Ros96a]. An additional benefit of requiring explicit calls to enable privileges was that these calls could be quickly identified with text searching tools such as `grep` and then subjected to code auditing. With limited time to audit a large code base, this technique allows an audit to focus its efforts on code that will effect the security of the system.

Extended Stack Inspection

The stack inspection algorithm used in current Java systems can be thought of as a generalization of the simple stack inspection model described above. Rather than having only untrusted applets and fully trusted system code, the system supports code from an unbounded number of principals, each of which may be granted different levels of trust. Likewise, rather than having either full privileges or restricted applet privileges, a number of more specific privileges called *targets* are defined, so different principals may be granted different degrees of access to the system.

Four fundamental primitives are necessary to use extended stack inspection:[4]

- `enablePrivilege()`

[4]Each Java vendor has different syntax for these primitives. This dissertation follows the Netscape syntax.

- `disablePrivilege()`
- `revertPrivilege()`
- `checkPrivilege()`

As described in the previous section, privileges to use a target must be *enabled* before the target is used. This is done by calling `enablePrivilege(`*T*`)` on the desired target *T*, placing a flag on the caller's stack frame where it can be found later by a stack search. After calling `enablePrivilege()`, the program may continue normal execution. When execution reaches the code that manages a security-relevant resource, such as the file system or network, this code will wish to make a security check to validate that its caller and the rest of the callers on the stack have sufficient permissions to use the resource. This is done by calling `checkPrivilege(`*T*`)`, which searches on the stack for a method that called `enablePrivilege(`*T*`)`, using the algorithm in figure 5.5.

The remaining primitives of stack inspection are fairly straightforward. `disablePrivilege(`*T*`)` places a flag on the current stack frame in the same manner as `enablePrivilege(`*T*`)`, except this flag says to terminate the search immediately and deny access to the target. The final operation, `revertPrivilege(`*T*`)` is used to clear flags related to *T* from the current stack frame.

The generalized `checkPrivilege()` algorithm, used by all three implementations, 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 appropriately enabled its privileges. The search also terminates, forbidding access (and throwing an exception), upon finding a stack frame that is either forbidden by the local policy from accessing the target or that has explicitly disabled its privileges.

We note that each vendor takes different actions when the search reaches the end of the stack uneventfully (*i.e.*, if `enablePrivilege()` was never called and all stack frames are trusted classes): NS 4.0 denies permission, while both JDK 1.2 and MSIE 4.0 allow it. The NS 4.0 approach follows the principle of least privilege (see Section 5.3.4), since it requires that privileges be explicitly enabled before they can be used — this can help protect an applet against a hostile caller trying to take advantage of the applet's privileges. The MSIE 4.0 / JDK 1.2 approach, on the other hand, may be easier for developers, since no calls to `enablePrivilege()` are required in the common case where trusted applets are using trusted system libraries. It also allows local, trusted Java applications to run on an older JVM without support for stack inspection: they run as a trusted principal so all of their accesses are allowed by default. Because NS 4.0 stack inspection is currently used only in Netscape's Web browser, compatibility with existing Java *applications* is not considered to be an issue.

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 access the protected resource directly since it would be unable to create the necessary enabled privilege. Figure 5.6 demonstrates how code for a trusted subsystem may be written.

Several things are notable about this example. The classes in figure 5.6 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 could be used by a third party to provide new functionality to any untrusted applet.

With the trusted subsystem, an untrusted applet now has two ways to open

```
checkPrivilege (target) {
    // loop, newest to oldest stack frame
    foreach stackFrame {
        if (local policy forbids access to target by class executing in stackFrame)
            throw ForbiddenException;

        if (stackFrame has enabled privilege for target)
            return; // allow access

        if (stackFrame has disabled privilege for target)
            throw ForbiddenException;
    }

    // if we reached here, we fell off the end of the stack
    if (Netscape 4.0)
        throw ForbiddenException;
    if (Microsoft IE 4.0 || Sun JDK 1.2)
        return; // allow access
}
```

Figure 5.5: Java's stack walking algorithm.

a file: call through `TrustedService`, which will restrict it to a subset of the file system, or request `UniversalFileRead` privileges for itself. Because there are no other ways to enable privileges for the `UniversalFileRead` target (assuming correct implementation of the Java system classes), 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.

Details

The vendor implementations have many additional features beyond those described above. We will attempt to describe a few of these enhancements and features here.

Threads An interesting issue is how to manage privileges that must cross thread boundaries. Since separate threads have separate stacks, stack inspection might seem to have a problem managing this case. There are two cases to address: when one thread starts a child thread, and when two existing threads are communicating. In the case of child threads, MSIE 4.0 and JDK 1.2 have the child thread inherit the privileges of its parent, effectively copying the enabled privileges from the parent thread's stack to the child thread before it begins execution. NS 4.0, in contrast, starts the child thread with no privileges enabled, assuming it can always enable any privilege it needs.

"Smart Targets" Sometimes a security decision depends not only on which resource (the file system, network, etc.) is being accessed, but on what 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, each vendor has a

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

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());
    }
}
```

Figure 5.6: A `FileSystem` capability.

This example shows how to build a `FileSystem` capability (see figure 5.1) as an example of a protected subsystem using extended stack inspection. Note that figure 5.1's `SubFS` can work unmodified with this example.

form of “smart targets” that have internal parameters and can be queried dynamically for access decisions. JDK 1.2 implements these by creating subclasses of their Permission classes. Some subclasses, such as FilePermission, take an additional argument of the file name to be checked. A security policy may specify files to be granted or denied permissions. NS 4.0 has similar support for subclasses of Target that validate arguments.

Who can define targets? Each system offers a set of predefined targets that represent resources the JVM implementation wants to protect; they also offer ways to define new targets. MSIE 4.0 allows only fully trusted code to define new targets, while NS 4.0 and JDK 1.2 allow anyone to define new targets, while taking steps to guarantee new targets are not confused with existing ones. In NS 4.0, 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. The principal field effectively defines a name space for targets. Built-in targets belong to the predefined *System* principal. In JDK 1.2, targets are classes in the Java library (*e.g.*, `java.io.FilePermission`), using Java’s existing class name space to separate targets, and new targets may be written as subclasses of existing targets.

Allowing anyone to define new targets, as NS 4.0 does, allows third-party library developers to define their own protected resources and use the system’s stack inspection mechanisms to protect them. The drawback is that users may be asked questions regarding targets that the Netscape designers did not know about. If the third-party library designer wishes to answer these questions internally without querying the user or wishes to display a security dialog, that is up to them. Regardless, NS 4.0 does not allow third-party libraries to create a target that will be mistaken for a Netscape-internal target by virtue of the protected system name space.

JDK 1.2 extensions The stack inspection semantics of JDK 1.2 have changed several times during its beta release cycles. Here are JDK 1.2's important differences from the NS 4.0 and MSIE 4.0 systems.

JDK 1.2 does not support `revertPrivilege()` or `disablePrivilege()` operations. Also, `enablePrivilege()` takes no arguments. Instead, it may be thought of as enabling a *root target*. All targets in JDK 1.2 support an `implies()` method that encapsulates the idea that some targets are supersets of others. For example, a target representing the entire filesystem would *imply* a target representing a single file. The `implies()` method on targets defines a directed graph of all targets.

Sun has not clearly defined whether the `implies()` relationships are transitive, and if they are, whether this information may be used to optimize security queries. One useful optimization might be to compute the transitive closure of the `implies()` graph, which would then allow for constant-time queries. However, if a target is allowed to change its mind about what other targets it implies, the security system might be forced to reevaluate the `implies()` relationships on every security query.

Finally, between JDK 1.2beta3 and beta4, the `enablePrivilege()` call was replaced with a `doPrivileged()` method whose argument must implement a `PrivilegedAction` interface. So, code that would previously have been written as:

```
public void foo() {
    enablePrivilege();
    do_something();       // this will execute with privileges
    revertPrivilege();
    do_something_else();  // this executes normally
}
```

would now be written in JDK 1.2 as:

```
public void foo() {
    doPrivileged(new PrivilegedAction() {
        public void run() {
            do_something();
        }
    }
    do_something_else();
}
```

These two code examples are semantically identical. The new JDK 1.2 syntax, using inner classes, probably simplified the job of the compiler writer, since the stack searching procedure need only search for the stack frame corresponding to the `doPrivileged()` operation, rather than maintain security flags on all stack frames.

The downside of the JDK 1.2 semantics is that they encourage the use of inner classes, which can be quite dangerous in security-relevant code. The Java virtual machine has no fundamental support for inner classes. In particular, the JVM has no support for a class to be *nested* within another, and thus being allowed access to its parent's private members and methods. Instead, the Java compiler will create accessor methods as necessary, each of which could be a serious security hole.

Implementation size. NS 4.0's stack inspection system 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.

JDK 1.2's stack inspection system required about 5200 lines of Java source code [GS98] and an unspecified amount of C code. We do not have any size information for MSIE 4.0.

5.3 Analysis

Now that we have presented four systems, we need a set of criteria to evaluate them. The criteria used here are derived from the work of Saltzer and Schroeder [SS75] and include several non-security issues such as compatibility with existing code as well as performance. Table 5.2 summarizes the security properties of all four architectures.

5.3.1 Economy of Mechanism

Designs that are smaller and simpler are easier to inspect and trust.

Capabilities are fundamentally the simplest mechanism. They rely only on the existing Java type system and require no changes or extensions to the JVM. However, a more fully capability-based Java system would require redesigning the Java class libraries to present a capability-style interface — a significant departure from the current class library, although capability-style classes ("factories") are used internally in some Java classes.

Process models are relatively simple because they do not depend on the VM for any security. Instead, they either use external operating system mechanisms or simulate the OS interaction between a user and kernel. These mechanisms are

Criteria	**Processes**	**Capabilities**	**Name-space management**	**Stack inspection**
Economy of mechanism	straightforward processes but complex RPCs and reimplementation of security checks	no special mechanisms required	simple mechanism requires complex mappings to avoid type inconsistencies	moderately complex mechanism
Fail-safe defaults	Similar defaults on all systems (*)			
Complete mediation	Very strong	Problematic capability leaks	Very strong	Strong and more informative than others
Least privilege	Strong	Very strong	Hard to limit privileges	Strong (*)
Code auditability	Trust the process abstraction, not the Java VM	Must potentially audit the whole VM		Simplifies VM auditing
Least common mechanism	Strong	Lots of shared state		
Accountability	Easy to track	Problematic	Reasonable	Easy to track
Resource limits	Easy to enforce but more copying necessary	No known solution		
Psychological acceptability	Too early to really know			
Performance	cross-process calls 10x-500x slower than local calls	Fast and cheap		uses 1% to 9% of CPU
Compatibility (API)	Process separation may break cooperating or privileged applets	Complete incompatibility	Excellent	Good (*)
Compatibility (VM)	Some use a stock VM, some make major changes	Excellent, no VM changes	Small VM hacks to manage name spaces	Larger VM hacks to support stack inspection

Table 5.2: Evaluation of different security architectures.
Applying our evaluation criteria to the different security architectures results in no clear best strategy. (*) Note that different implementations of stack inspection trade off API compatibility against fail-safe defaults and least privilege.

fairly well understood but are not, in and of themselves, simple. Process models also require communication across processes to be compatible with existing implementations. This may be fairly complex. Furthermore, to implement the "sandbox" security policies in an environment where the Java VM is not considered trustworthy, the process security infrastructure must repeat the runtime checks normally done in Java. Some rules, such as the restrictions on Java network connections, are fairly subtle and could be easily misimplemented.

Name space management is fairly straightforward. 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.

The original stack inspection implementations require complex changes to the virtual machine, although security-passing style implementations will only require a front-end to rewrite Java bytecode (see chapter 8). As before, changes to the JVM could destabilize the whole system. Each method 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 that has not been modified to consult the security system can be an avenue for system compromise.

5.3.2 Fail-Safe Defaults

By default, access should be denied unless it is explicitly granted.

Name space management and stack inspection have similar fail-safe behavior. If a potentially dangerous system resource has been properly modified to work with the system, it will deny access to unauthorized callers. With name-space management, the protected resource cannot be named by a program, so it is not reachable. With NS 4.0 stack inspection, privileges must be explicitly enabled before a dangerous resource can be used. When no enabled privilege is found on the stack, access to the resource will be denied by default. (MSIE 4.0 and JDK 1.2 sacrifice this property for compatibility reasons.)

In a fully capability-based system, a program cannot do anything unless an appropriate capability is available. By restricting the default list of capabilities granted to a program, fail-safe defaults can be supported.

In a process-based system, specific privileges are granted to the process, as mediated by the underlying operating system or process abstraction. By forcing all access through such "choke points" (*e.g.,* system-call interfaces), it becomes easy to prevent a process from acting beyond its privileges.

5.3.3 Complete Mediation

Every access to every object should be checked.

Barring oversights or implementation errors (discussed in sections 5.3.1 and 5.3.2), all three systems provide mechanisms to interpose security checks between identified targets and anyone who tries to use them.

However, mediation is not complete if privileges are not *confined* to their rightful holders [Lam73]. 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 in Java; two programs that can communicate object references can share their capabilities without system mediation. This means that any code granted a capability must either be trusted to care for it properly or must be prevented from communication with an untrusted program. Many extensions to capabilities have been proposed to address these concerns. Kain and Landwehr [KL87] propose a taxonomy for extensions and survey many systems implementing 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 [Gon89], make the objects referenced by capabilities aware of "who" called them; an object can know who is supposed to invoke it and refuse to work for anyone else. The IBM System/38 [BTR80] associates optional access control lists with its capabilities, accomplishing the same purpose. Other systems use hardware mechanisms to block the sharing of capabilities [KH84]. For Java, any such technique would be problematic. To make an object aware of who is calling it, a certain level of inspection into the call stack must be available. To make an object reference 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 has good confinement properties. For example, if a program attempts to give an open `FileInputStream` to another program that 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 be addressed by judicious redesign of the Java class libraries.

Stack inspection 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 that is authorized to use them.

Current stack inspection systems actually compromise confinement somewhat for performance. These systems only use stack inspection to verify security for otherwise expensive operations such as opening a disk file or network connection, where much higher latencies are normal. Once the connection is open, the more common read and write operations do not invoke the `checkPrivilege()` operation. Instead, the `InputStream` and `OutputStream` objects are treated as capabilities. While a specific input or output stream could potentially be leaked across applets, the general ability to open a file or network connection would still be contained.

Processes also have excellent confinement. As discussed earlier, a process abstraction forces all attempts to access resources through a centralized "choke point" where privilege decisions can be made. Unlike stack inspection, there is never any ambiguity in a process model over where a request came from, simplifying the application of access controls.

An issue with any system is when privileged code maliciously acts as a "proxy" for unprivileged code. Such proxying is a general issue in any system where communication is allowed between programs of different privilege levels. Neither stack inspection nor name-space management have special provisions against proxy attacks. However, all but the capability systems at least guarantee that the privileged calls can be traced to the program that was originally granted the privileges, and a simple extension to the capabilities system could give similar guarantees.

5.3.4 Least Privilege

Every program should operate with the minimum set of privileges necessary to do its job.

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. However, because Java's dynamic linker is *lazy* (linking a class when it is first used), some flexibility is available before linking has occurred.

In contrast, capability systems 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 access modifiers provide the necessary semantics to prevent the intermediate code from using the capability.

Stack inspection provides an interesting middle-ground. The act of enabling a privilege is equivalent to taking responsibility for the subsequent use of that privilege. If a privilege is not enabled, the corresponding target cannot be used (in NS 4.0, whereas MSIE 4.0 and JDK 1.2 relax this requirement — see Section 5.2.4). Likewise, the lifetime and visibility of an enabled privilege are limited by the fact that privileges are automatically discarded when a privilege-enabled method returns. These limits reduce the system's exposure to damages from running with unnecessary privileges.

In a process-based system, privileges will likely be established before the process begins running. Because they exist outside of the Java VM, there may not be a way for a program to discard its privileges if they are not needed. The alternative would be to provide extensions to the Java system that allow it to manipulate the state of its controlling process. In contrast, one of the benefits of a process system, especially one that runs the untrusted code on a separate machine, is that most dangerous privileges are simply unavailable. There is no way for an applet to read a user's personal files if those files are not accessible from the remote machine. However, if a process-based system intends to support access to the file system for *some* applets (*i.e.,* applets that have been granted privileges by a user), then more privileges must be available. This lessens any benefit that might be gained from running applets on a separate machine.

5.3.5 Code Auditability

Large programs have bugs. The security architecture can help focus the efforts of auditors to find bugs that impact security.

Auditability is closely related to least privilege. If the security architecture can help a security audit rule out code that has no effect by virtue of its running with reduced privileges, the audit can be focused on more dangerous code.

Name space management effectively defines a list of classes that must be inspected for correctness. With both capabilities and stack inspection, 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 inspection, however, has stronger limits on the propagation of privileges: a privilege is only available to the transitive closure of code that is called after a privilege is enabled. If code is not in this control flow, it will not be able to exercise the privilege.

For example, Netscape ported Sun's RMI (remote method invocation) to run inside the Netscape VM. This was tricky as RMI was never designed to be used from within applets, as it wished to install its own SecurityManager and use its own ClassLoaders. When reviewing this port, we focused our efforts on studying the code immediately preceding and following calls to enable privileges and were able to rapidly identify security-relevant bugs. RMI was now available for any applet in the Netscape VM. This would not have been possible without stack inspection.

Process models are a very different challenge to audit. When the system does not depend on any of Java's security, it must then rely on either its own security mechanisms or those of the underlying operating system. As operating systems can be quite large, auditing them is non-trivial. Also, if the operating system was written in a non-type-safe language such as C, there may be hidden opportunities for buffer overflows that a malicious applet could try to exploit.

5.3.6 Least Common Mechanism

Anything shared among different programs can be a channel for communication and a potential security hole, so as little data as possible should be shared.

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 depending on it. This problem applies equally to capabilities, name-space management, and stack inspection. An example of this problem was Hopwood's `interface` attack [MF97], which combined a bug in Java's interface mechanism with a shared public variable to break the type system, and thus circumvent system security.

To truly separate Java programs, a process model may be appropriate, although channels between the processes will still be required to allow cooperation between applets.

5.3.7 Accountability

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

In the event that the user has granted trust to a program that 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 inspection system, every call to enable a privilege can be logged; an administrator can learn which principal enabled the privileges to damage the system. Because all the intermediary code is tracked when a privilege is checked, this could also be recorded, allowing for a very complete picture of how privileges are being used.

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 5.3.3), 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.

Process models have enough information to record which applet was responsible for using a privilege because of the strong separation between applets.

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

5.3.8 Resource Limits

Mechanisms should exist to fairly meter out finite system resources such as memory, CPU, and disk space.

Traditional Java systems have had no resource limits. There was nothing to prevent any Java applet from going into an infinite for-loop or from allocating memory without limit. Traditional Java schedulers were simple round-robin systems that, on some platforms, did not even support preemption.

The Java operating systems designed by the Flux project at Utah [BTS+98, TL98] are the only current systems to apply resource limits to Java, and they work with a fairly traditional process model. While this separation increases the costs of sharing between applets, it makes traditional resource management possible.

Systems like stack inspection can make binary access control decisions such as whether a specific request to make a network connection should be granted. However, they have no provisions to meter out limited resources. This will be an interesting area for future research.

5.3.9 Psychological Acceptability

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

All systems face a fundamental issue: the security policy must be specified *somewhere*. If the policy were expressed directly as an access control matrix, it would be quite large and unwieldy. The number of possible principals is potentially unbounded and the number of targets is also quite large.

The solution currently employed by both Netscape and Microsoft is a form of lazy evaluation. When a specific query is made to the security policy, if it has not been previously answered, a security dialog box will ask the user to decide whether the requested access should be allowed or granted. Asking the user questions too often can result in a condition called "dialog fatigue," where the user learns to ignore the dialogs, hitting the "OK" button automatically to continue what they were doing.

As discussed in section 5.1.2, dialog fatigue can be addressed by the grouping of privileges as well as principals. Likewise, providing hooks for a systems administrator to pre-load the system avoid requiring the user to answer security questions.

The security architectures presented here can all support these features to reduce a user's fatigue. A human-factors comparison between specific implementations (*e.g.,* NS 4.0 vs. MSIE 4.0) is beyond the scope of this dissertation.

5.3.10 Performance

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

Performance is one of the attractions of using type-safety for memory protection, so it would be undesirable if the addition of one of these security architectures had an adverse effect on system performance.

Process models impose considerable runtime overheads (see section 5.2.1). Whenever control flow crosses processes, all the data must be copied, since object references cannot be shared. If one process must reference the data of another, a distributed garbage collection algorithm must additionally run. Still, with 64-bit architectures, processes need not require the high overhead of context switching. Multiple Java processes could run in the same flat address space, gaining the same benefits associated with single address space operating systems [CLLBH92, CLFL94].

Depending on how stack inspection is implemented, the performance costs can vary widely. These costs are discussed in detail in chapter 8, and seem to vary from 1% to 9% CPU overhead running application benchmarks. 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 (*e.g.,* to validate or limit arguments to low-level system routines).

An interesting question to ask is what happens to the relative cost of making a call across protection boundaries as Java compilers become more aggressive in their optimization. We would be concerned if the security architecture required semantics that hindered traditional compiler optimizations.

Process-based systems may suffer the most in this regard. While the compiler is free to inline methods within a process, it cannot make any optimizations which cross processes. The security system must enforce a specific calling convention, allowing it to intercept and manage cross-process calls.

With a stack-walking implementation of stack inspection, the compiler may be allowed to perform any optimizations, since all the methods run within the same environment. However, the compiler must guarantee that all the stack information is preserved for later security checks. This is roughly equivalent to the problem of maintaining debugging information in optimized code [Hen82, TA90].

With a security-passing style implementation of stack inspection, with capabilities, or with name-space management, better compilers will result in faster systems all around. Because these systems are implemented on top of Java without underlying VM changes, any optimization that accelerates normal method calls will also accelerate cross-boundary calls as well.

5.3.11 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.

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

Systems with a process model should similarly be able to run unmodified applications. In cases where separate applets are intended to run in the same virtual machine and share memory, there could be compatibility issues because static variables will not necessarily be shared across the process boundaries. Likewise, “signed” applets that can request additional privileges may not work properly if the applet process is strongly separated from the user’s machine. An applet that tries to access a restricted browser resource would be denied.

On the other hand, as noted in section 5.3.1, a capability system would require a new class libraries and thus completely break compatibility with traditional Java classes.

5.4 Combinations

As demonstrated in the previous section, none of the Java security architectures is perfect. We now focus on how they can be combined practically in real systems to share their advantages and avoid their flaws.

5.4.1 Name-Space Management + Capabilities

While name-space management raises some concerns with *renaming* classes, it works well for *hiding* classes. This could provide the basis for implementing a strongly capability-based system. The classes implementing the capabilities would be allowed to see the original system classes, but other code would not.

5.4.2 Stack Inspection + Capabilities

Stack inspection can be expensive to use, but has nice security properties. Capability systems are quite fast, but require a radically different style of programming from Java's common usage. The combination gets many of the best features of both systems and it is already being used. As discussed in section 5.3.3, stack inspection protects "important" calls, such as opening files and network connections, while objects representing capabilities to use those files and network connections are returned. While this follows good object-oriented design, it can only be secure if one applet cannot leak its capabilities or steal the capabilities of another. If two applets are in a situation where one can get a reference to *any* data from the other applet, these leaks may be possible.

If a more strongly capability-based system were to be built, where the only way to open a file would be through a capability rather than through trying to construct an appropriate object directly, then stack inspection could be used to block anybody but the system from successfully calling Java's traditional file interfaces, while allowing the capability classes full access. For an example of how this might work, see figure 5.6.

If backward compatibility were not an issue, would there still be a place for stack inspection? Stack inspection offers guarantees about accountability and complete mediation that are not possible with pure capabilities (discussed in sections 5.3.3 and 5.3.5). Stack inspection can also be seen as a raw mechanism to extend a capability system to address the problems inherent with pure capabilities.

5.4.3 Processes + Capabilities

Existing systems that use processes to separate Java applets must support communication among these applets as well as with "kernel." The facilities to do this are closely related to remote procedure call systems. As discussed in section 4.2.2, capabilities are a simple way of adding security to a distributed system, so they would also work for process systems. In essence, any mechanism that provides security in distributed systems can be combined with processes to secure interprocess communication.

5.5 Conclusion

We have described four architectures which support interposition of security checks between untrusted code and important system resources. Each design has been implemented in Java and two of them (extended stack inspection and name space management) have been integrated in commercial Web browsers.

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

Name space management offers compatibility with existing Java applets but

Java's libraries and newer Java mechanisms such as Java reflection may limit its use.

Process models seem to offer good security, but they impose performance costs that question their usefulness.

Stack inspection has been adopted by the commercial Web browsers, offering relatively good security. The remainder of this dissertation will be focused on studying stack inspection and addressing its weaknesses.

Chapter 6

Access Control Logic

In order to gain a more sophisticated understanding of stack inspection, we found it necessary to build a *model* of the system. A good model would hide many of the details of the system and allow us to reason about it. In particular, we would like the model to capture the "security state" of the system at any time and let us express transitions from this state as mathematical operations.

There are no hard and fast rules of how one should model a system formally. Instead, the most expedient path is to find a formal model of a similar system and adapt it to stack inspection. The system that we decided to borrow from was originally used to describe authentication and access control in the Taos operating system [LABW92, WABL94]. In Taos, the operating system maintains information about every channel between processes on the same machine and across the network. When a process receives a request, the process may ask the system to identify who has connected to it. Because a channel may pass through multiple points of trust (the local operating system, the network, the remote operating system, etc.), the system explicitly puts these into the principal, creating a *compound principal*. Taos actually included a theorem prover inside the system which could, given these compound principals and a security policy, both expressed in the same formal logic, generate proofs of whether a given request is authorized to occur. The logic, a relatively simple propositional or modal logic with no negation of statements (and with certain restrictions on the form of statements), allows the theorem prover to run fast enough to not dramatically impact system performance. We decided to adopt this logic, originally specified by Abadi, Burrows, Lampson, and Plotkin [ABLP93] (hereafter, ABLP logic), to model stack inspection.

6.1 ABLP Logic

Stack inspection can be modeled using a subset of ABLP. This section will describe the subset and give a general flavor for how it can be used.

The logic is based on a few simple concepts: principals, conjunctions of principals, targets, statements, quotation, and authority.

- A *principal* is a person, organization or any other entity that may have the right to take actions or authorize actions. In addition, entities such as programs and cryptographic keys are often modeled as principals.

- A *target* represents a resource that we wish to protect. Loosely speaking, a target is something to which we might like to attach an access control list. (Targets are traditionally known as "objects" in the literature, but this can be confusing when talking about an object-oriented language.)

- A *statement* is any kind of utterance a principal can emit. Some statements are made explicitly by a principal, and some are made implicitly as a side-effect of actions the principal takes. In other words, we interpret P **says** s as meaning that we can act as if the principal P supports the statement s. Note that saying something does not make it true; a speaker could make an inaccurate statement carelessly or maliciously. The logic supports the informal notion that we should place faith in a statement only if we trust the speaker and it is the kind of statement that the speaker has the authority to make. Thus, if a speaker makes an inaccurate statement, we will not believe the statement. Also, speakers cannot make statements that lead to a logical contradiction (*e.g.,* $A \supset \neg A$) because negation is not allowed in ABLP.

The most common type of statement we will use looks like P **says** $Ok(T)$ where P is a principal and T is a target; this statement means that P is authorizing access to the target T. By saying an action is "*Ok*" the speaker is saying the action should be allowed in the current context but is not specifically ordering that the action take place.

- The logic supports *conjunctions of principals.* Specifically, saying $(A \wedge B)$ **says** s is the same as saying both A **says** s and B **says** s.

- *Quotation* allows a principal to make a statement about what another principal says. The notation $A \mid B$ **says** s, which we pronounce "A quoting B says s," is equivalent to A **says** (B **says** s). As with any statement, we must consider whether A's utterance might be incorrect, and our degree of faith in s will depend on our beliefs about A and B. When A quotes B, we have no guarantee that B ever actually said anything.

- We grant *authority* to a principal by allowing that principal to speak for another principal who has power to do something. The statement $A \Rightarrow B$, pronounced "A speaks for B," means that if A makes a statement, we can assume that B supports the same statement. If $A \Rightarrow B$, then A has at least as much authority as B. Note that the $\Rightarrow$-operator can be used to represent group membership: if P is a member of the group G, we can say $P \Rightarrow G$, meaning that P can exercise the rights granted to G.

When proving theorems, the $A \Rightarrow B$ means occurrences of A can be replaced with $A \wedge B$. Thus, when we hear a statement from A, we can act as if it were spoken jointly by A and B together.

6.2 ABLP Grammar

This section presents a grammar for valid ABLP expressions. Note that, while this grammar is not completely unambiguous, the axioms that operate on ABLP expressions are unambiguous.

There are two fundamental types in ABLP: statements and principals. A statement could be an atomic statement such as “the sky is blue.” It could also be a compound statement such as “Bob says the sky is blue” or “Alice speaks for Bob,” as indicated with the $\Rightarrow$ symbol. A statement may also be the conjunction of several independent statements, as indicated with the $\wedge$ symbol. Or, a statement may imply another statement, as indicated with the $\supset$ symbol.

Statement	$\rightarrow$	*AtomicStatement*
Statement	$\rightarrow$	Statement $\wedge$ Statement
Statement	$\rightarrow$	Statement $\supset$ Statement
Statement	$\rightarrow$	Principal **says** Statement
Statement	$\rightarrow$	Principal $\Rightarrow$ Principal

A principal can be an atomic principal such as “Alice” or “Bob.” It may also be a compound principal such as “Alice quoting Bob,” as indicated with the $|$ symbol or a conjunction of principals, as indicated with the $\wedge$ symbol.

$$\begin{aligned}\text{Principal} &\rightarrow \textit{AtomicPrincipal}\\ \text{Principal} &\rightarrow \text{Principal} \mid \text{Principal}\\ \text{Principal} &\rightarrow \text{Principal} \wedge \text{Principal}\end{aligned}$$

To avoid ambiguous statements or at least make statements more legible, parentheses are also acceptable in all the obvious places.

$$\begin{aligned}\text{Principal} &\rightarrow (\text{ Principal })\\ \text{Statement} &\rightarrow (\text{ Statement })\end{aligned}$$

Thus, it's perfectly reasonable to make a statement such as

$$\begin{aligned}&((Alice \wedge Bob)\ \textbf{says}\ (Charlie \Rightarrow (Alice \wedge Bob))) \wedge\\ &(Charlie|Alice\ \textbf{says}\ X) \wedge\\ &((Alice\ \textbf{says}\ X) \supset X)\end{aligned}$$

where the first part is a form of delegation (Alice and Bob delegating their privileges to Charlie), the second part is an assertion that "Charlie quoting Alice" wants to do X (*i.e.,* Charlie is claiming that Alice wants to do X), and the third part is an access control rule stating that when Alice says she wants to do X, we will believe her.

6.3 Axioms

Here is a list of the subset of axioms in ABLP logic used in this dissertation. We omit axioms for delegation, roles, and exceptions because they are not necessary to discuss stack inspection.

Axioms About Statements

$$\text{If } s \text{ is an instance of a theorem of propositional logic then } s \text{ is true in ABLP.} \tag{6.1}$$

$$\text{If } s \text{ and } s \supset s' \text{ then } s'. \tag{6.2}$$

$$(A \textbf{ says } s \wedge A \textbf{ says } (s \supset s')) \supset A \textbf{ says } s'. \tag{6.3}$$

$$\text{If } s \text{ then } A \textbf{ says } s \text{ for every principal } A. \tag{6.4}$$

Axioms About Principals

$$(A \wedge B) \textbf{ says } s \equiv (A \textbf{ says } s) \wedge (B \textbf{ says } s) \tag{6.5}$$

$$(A \mid B) \textbf{ says } s \equiv A \textbf{ says } B \textbf{ says } s \tag{6.6}$$

$$(A = B) \supset (A \textbf{ says } s \equiv B \textbf{ says } s) \tag{6.7}$$

$$(A \mid (B \mid C)) \equiv ((A \mid B) \mid C) \tag{6.8}$$

$$(A \mid (B \wedge C)) \equiv (A \mid B) \wedge (A \mid C) \tag{6.9}$$

$$(A \Rightarrow B) \equiv (A = A \wedge B) \tag{6.10}$$

$$(A \textbf{ says } (B \Rightarrow A)) \supset (B \Rightarrow A) \tag{6.11}$$

So, given the the following statement:

$$((Alice \wedge Bob) \textbf{ says } Charlie \Rightarrow (Alice \wedge Bob)) \wedge$$
$$(Charlie|Alice \textbf{ says } X) \wedge$$
$$((Alice \textbf{ says } X) \supset X)$$

we might try to prove X.

$Charlie \Rightarrow (Alice \wedge Bob)$	by axiom 6.11
$(Charlie \wedge Alice \wedge Bob) \vert Alice$ **says** X	by axiom 6.10
$(Charlie \vert Alice$ **says** $X) \wedge$	
$\quad (Alice \vert Alice$ **says** $X) \wedge (Bob \vert Alice$ **says** $X)$	by axiom 6.5
$Alice \mid Alice$ **says** X	by axiom 6.1
$Alice$ **says** $Alice$ **says** X	by axiom 6.6
$Alice$ **says** X	by axiom 6.3
X	by axiom 6.3 ∎

Now, in general, not all ABLP proofs are this easy. It is possible to encode problems in ABLP that are equivalent to the halting problem. However, by carefully choosing a subset of ABLP, we can not only guarantee that proofs are decidable, but we can also make efficient decision procedures for them. Chapter 7 presents the subset of ABLP that we use to model Java's stack inspection and presents an efficient decision procedure for it.

6.4 Applying ABLP

With an understanding of how ABLP logic works, we can explain how it can be used to model actual systems. A great amount of detail on this is available in Lampson, Abadi, Burrows, and Wobber [LABW92]. ABLP can be used to model the flow of control through a single system, from user to keyboard to motherboard to device driver to operating system to user process. It can also be used to model information passing across a network to the same level of detail. The key is quoting. When an application receives a keystroke, it might want to verify that the keystroke, in fact, came from the user. In the model, such an application would be required to validate

$$(\mathit{Kernel}|\mathit{DeviceDriver}|\mathit{Keyboard}\ \mathbf{says}\ \mathit{KeyPressed}(\text{‘}g\text{’})) \supset \mathit{KeyPressed}(\text{‘}g\text{’})$$

In order to do this, it must believe that each layer truthfully speaks for the layer below it:

$$\mathit{Kernel} \Rightarrow \mathit{DeviceDriver}$$

$$\mathit{DeviceDriver} \Rightarrow \mathit{Keyboard}$$

$$(\mathit{Keyboard}\ \mathbf{says}\ \mathit{KeyPressed}(\mathrm{x})) \supset \mathit{KeyPressed}(\mathrm{x})$$

Given the above beliefs and the axioms of ABLP logic, an application may safely believe in the authenticity of its keystrokes.

If we wish to add a network window server (such as X) to this model, we must prove that the window server speaks for the keyboard. Such a proof would require modeling the event dispatch mechanism inside the server. If the window server supported features like synthetic key events (where an application may simulate keystroke events to drive another application), this would also need to be taken into account in the model. As the model's complexity grows, our certainty of keystroke authenticity is dependent on our ability to make proofs as above.

ABLP can be applied to all kinds of authentication problems. A related logic, BAN logic [BAN90], has been applied to the underlying cryptographic protocols as well. In the next chapters of this dissertation, ABLP will be used to model the authentication and access control within Java.

Chapter 7

Understanding Java Stack Inspection

In chapter 5, we introduced the idea of stack inspection as a technique for managing access control in the Java environment. This chapter revisits stack inspection, presenting a model of stack inspection using ABLP logic. While the model is much simpler than the original stack inspection system, we prove the model is equivalent to the original specification and we present an efficient decision procedure for generating these proofs.

By examining the decision procedure, we demonstrate that many statements in the logic are equivalent and can thus be expressed in a simpler form. We show that there are a finite number of such statements, allowing us to represent the security state of the system as a deterministic pushdown automaton. We also show that this automaton may be embedded in Java by rewriting all Java classes to pass an additional argument when a procedure is invoked. We call this *security-passing style* and describe its benefits over previous stack inspection systems (chapter 8 discusses an implementation based on this). Finally, we show how the logic allows us to describe a straightforward design for extending stack inspection across remote procedure calls.

7.1 Mapping Java to ABLP

We will now describe a mapping from the stack, the privilege calls, and the stack inspection algorithm into ABLP logic.

7.1.1 Principals

In Java, code is digitally signed with a private key, then shipped to the virtual machine where it will run. If K_{Signer} is the public key of *Signer*, the public-key infrastructure can generate a proof[1] of the statement

$$K_{Signer} \Rightarrow Signer. \tag{7.1}$$

Signer's digital signature on the code *Code* is interpreted as

$$K_{Signer} \textbf{ says } (Code \Rightarrow K_{Signer}). \tag{7.2}$$

Using equations 7.1 and 6.11, this implies that

$$Code \Rightarrow Signer. \tag{7.3}$$

When *Code* is invoked, it generates a stack frame *Frame*. The virtual machine assumes that the frame speaks for the code it is executing:

$$Frame \Rightarrow Code. \tag{7.4}$$

The transitivity of $\Rightarrow$ (which can be derived from equation 6.10) then implies

$$Frame \Rightarrow Signer. \tag{7.5}$$

We define Φ to be the set of all such valid $Frame \Rightarrow Signer$ statements. We call Φ the *frame credentials*.

[1]Throughout this dissertation we assume that sound cryptographic protocols are used, and we ignore the extremely unlikely possibility that an adversary will successfully guess or otherwise acquire a private key.

Note also that code can be signed by more than one principal. In this case, the code and its stack frames speak for all of the signers. To simplify the discussion, all of our examples will use single signers, but the theory can support multiple signers without difficulty.

7.1.2 Targets

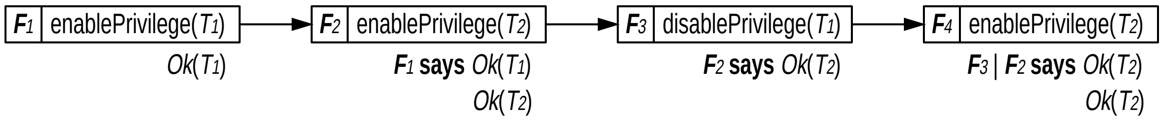
Recall that the resources we wish to protect are called *targets*. For each target, we create a dummy principal whose name is identical to that of the target. These dummy principals do not make any statements themselves, but various principals may speak for them.

For each target T, the statement $Ok(T)$ means that access to T should be allowed in the present context. The axiom

$$(T \textbf{ says } Ok(T)) \supset Ok(T) \tag{7.6}$$

says that T can allow access to itself.

Many targets are defined in relation to services offered by the operating system underlying the Java Virtual Machine (JVM). From the operating system's point of view, the JVM is a single process and all system calls coming from the JVM are performed under the authority of the JVM's principal (often the user running the JVM). The JVM's responsibility, then, is to allow a system call only when there is justification for issuing that system call under the JVM's authority. Our model will support this intuition by requiring the JVM to prove in ABLP logic that each system call has been authorized by a suitable principal.



F_1 enablePrivilege(T_1)
$Ok(T_1)$
F_2 enablePrivilege(T_2)
F_1 says $Ok(T_1)$
$Ok(T_2)$
F_3 disablePrivilege(T_1)
F_2 says $Ok(T_2)$
F_4 enablePrivilege(T_2)
$F_3 \mid F_2$ says $Ok(T_2)$
$Ok(T_2)$

Figure 7.1: Security contexts of successive stack frames.
Each rectangle represents a stack frame. Each stack frame is labeled with its name. In this example, each stack frame makes one `enablePrivilege()` or `disablePrivilege()` call, which is also written inside the rectangle. Below each frame is written its security context after its call to `enablePrivilege()` or `disablePrivilege()`.

7.1.3 Setting Policy

We use a standard access matrix [Lam71], implemented with with hashtables to achieve compact storage, to keep track of which principals have permission to access which targets. If *VM* is a Java virtual machine, we define A_{VM} to be a set of statements of the form $P \Rightarrow T$ where P is a principal and T is a target. If $(P \Rightarrow T) \in A_{VM}$, this means that the local policy in *VM* allows P to access T. We call A_{VM} the *access credentials* for the virtual machine *VM*.

7.1.4 Stacks

When a Java program is executing, we treat each stack frame as a principal. At any point in time, a stack frame F has a set of statements that it believes. We refer to this as the *security context* of F and write it S_F. We now describe where the security context comes from.

Starting a Program

When a program starts, we need to set the security context of the initial stack frame, S_{F_0}. In the Netscape model, $S_{F_0} = \{\}$. In the Sun and Microsoft models, $S_{F_0} = \{Ok(T) \mid T \in Targets\}$. These correspond to Netscape's initial unprivileged state and Sun and Microsoft's initial privileged state.

Enabling Privileges

If a stack frame F calls `enablePrivilege(`T`)` for some target T, it is really saying it *authorizes* access to the target. We can represent this simply by adding $Ok(T)$ to S_F.

Calling a Procedure

When a stack frame F makes a procedure call, this creates a new stack frame G. As a side-effect of the creation of G, F tells G the statements in F's security context. Thus, when F tells G a statement S, the statement F **says** S is added to S_G.

Disabling and Reverting Privileges

A stack frame can also choose to disable some of its privileges. The call `disablePrivilege(`T`)` asks to disable any privilege to access the target T. This is implemented by giving the frame a new security context that consists of the old security context with all statements in which anyone says $Ok(T)$ removed. `revertPrivilege()` is handled in a similar manner, by giving the frame a new security context that is equal to the security context it originally had. The latest JDK 1.2beta4 from Sun does not support any calls equivalent to these, so these calls need not be modeled for Sun's version of the architecture.

Example

Figure 7.1 shows an example of these rules in action. In the beginning, $S_{F_1} = \{\}$. F_1 then calls `enablePrivilege(`T_1`)`, which adds the statement $Ok(T_1)$ to S_{F_1}.

When F_2 is created, F_1 tells it $Ok(T_1)$, so S_{F_2} is initially $\{F_1$ **says** $Ok(T_1)\}$. F_2 then calls `enablePrivilege(`T_2`)`, which adds $Ok(T_2)$ to S_{F_2}.

S_{F_3} initially contains $F_2 \mid F_1$ **says** $Ok(T_1)$ and F_2 **says** $Ok(T_2)$. When F_3 calls `disablePrivilege(`T_2`)`, the latter statement is deleted from S_{F_3}. S_{F_4} initially contains $F_3 \mid F_2$ **says** $Ok(T_1)$. When F_4 calls `enablePrivilege(`T_2`)`, this adds $Ok(T_2)$ to S_{F_4}.

7.1.5 Checking Privileges

Before making a system call or otherwise invoking a dangerous operation, the Java virtual machine calls `checkPrivilege()` to make sure that the requested operation is authorized. `checkPrivilege(`T`)` returns true if the statement $Ok(T)$ can be derived from Φ (the frame credentials), A_{VM} (the access control matrix), and S_F (the security context of the frame that called `checkPrivilege()`).

We define *VM(F)* to be the virtual machine in which a given frame F is running. Next, we can define

$$E_F \equiv (\Phi,\ A_{VM(F)},\ S_F). \tag{7.7}$$

We call E_F the *environment* of the frame F.

The goal of `checkPrivilege(`T`)` is to determine, for the frame F invoking it, whether $E_F \supset Ok(T)$. While such questions are generally undecidable in ABLP logic, we now present an efficient decision procedure that gives the correct answer for our subset of the logic. `checkPrivilege()` implements that decision procedure.

The decision procedure **check** used by `checkPrivilege()` takes, as arguments, an environment E_F and a target T. **check**(T, E_F) examines the statements in E_F and divides them into three classes.

- Class 1 statements have the form $Ok(U)$, where U is a target.

- Class 2 statements have the form $P \Rightarrow Q$, where P and Q are atomic principals.
- Class 3 statements have the form

  $$F_1 \mid F_2 \mid \cdots \mid F_k \textbf{ says } Ok(U),$$

  where F_i is an atomic principal for all i, $k \geq 1$, and U is a target.

The decision procedure next examines all Class 1 statements. If any of them is equal to $Ok(T)$, the decision procedure terminates and returns *true*.

Next, the decision procedure uses all of the Class 2 statements to construct a directed graph which we will call the speaks-for graph of E_F. This graph has an edge (A, B) if and only if there is a Class 2 statement $A \Rightarrow B$.

Next, the decision procedure examines the Class 3 statements one at a time. When examining the statement $F_1 \mid F_2 \mid \cdots \mid F_k$ **says** $Ok(U)$, the decision procedure terminates and returns *true* if both

- for all $i \in [1, k]$, there is a path from F_i to T in the speaks-for graph, and
- $U = T$.

If the decision procedure examines all of the Class 3 statements without success, it terminates and returns *false*.

Theorem 1 (Termination) *The* **check** *decision procedure always terminates.*

Proof: The result follows directly from the fact that E_F has finite cardinality; there are a finite number of principals that the system knows about, a finite number of stack frames that must be considered, and a security policy of finite length. This implies that each loop in the algorithm has a bounded number of iterations; and clearly the amount of work done in each iteration is bounded.▮

In fact, the decision procedure runs quite efficiently. We can separately analyze the runtime complexity and space complexity of each phase, as presented above. If there are N rules in the access control matrix $A_{VM(F)}$ and a stack depth of D (*i.e.*, Φ has at most D elements), the cost of computing the transitive closure of the graph will be, at worst, $O((N+D)^2)$ and consume $O((N+D)^2)$ space. Then, if there are k statements in S_F (each of which can have at most D principals in its quoting chain), the cost of checking all statements will be $O(kD)$. The total cost of the decision procedure is thus $O((N+D)^2 + kD)$.

In practice, the transitive closure of the access control policy can be pre-computed (subject to the caveats in section 5.2.4) and the $O(kD)$ complexity of the security context can be lowered as well. In section 7.3.1, we describe optimizations that allow the decision procedure to execute in constant time.

Theorem 2 (Soundness) *If the* **check** *decision procedure returns* true *when invoked in stack frame* F*, then there exists a proof in ABLP logic that* $E_F \supset Ok(T)$.

Lemma 1 *If there is a path from* A *to* B *in the speaks-for graph of* E_F*, then* $E_F \supset (\mathrm{A} \Rightarrow \mathrm{B})$.

Proof: By assumption, there is a path

$$(A,\ v_1,\ v_2,\ \ldots,\ v_k,\ B)$$

in the speaks-for graph of E_F. In order for this path to exist, we know that the statements

$$A \Rightarrow v_1,$$

$$v_i \Rightarrow v_{i+1} \text{ for all } i \in [1, k-1],$$

and

$$v_k \Rightarrow B$$

are all members of E_F. Since $\Rightarrow$ is transitive, this implies that

$$E_F \supset A \Rightarrow B.$$

Proof of Theorem 2: There are two cases in which the **check** decision procedure can return *true*.

1. The decision procedure returns *true* while it is iterating over the Class 1 statements. This occurs when the decision procedure finds the statement $Ok(T) \in E_F$. In this case, $Ok(T)$ follows trivially from E_F.

2. The decision procedure returns *true* while it is iterating over the Class 2 statements. In this case we know that the decision procedure found a Class 2 statement of the form

   $$P_1 \mid P_2 \mid \cdots \mid P_k \textbf{ says } Ok(T),$$

   where for all $i \in [1, k]$ there is path from P_i to T in the speaks-for graph of E_F. It follows from Lemma 1 that for all $i \in [1, k]$, $P_i \Rightarrow T$. It follows that

   $$E_F \supset (T| \ T| \ \cdots \mid T \textbf{ says } Ok(T)). \tag{7.8}$$

   Applying equation 7.6 repeatedly, we can directly derive $E_F \supset Ok(T)$. ▮

Conjecture 1 (Completeness) *If the* **check** *decision procedure returns false when invoked in stack frame* F*, then there is no proof in ABLP logic of the statement* $E_F \supset Ok(T)$.

Although we believe this conjecture to be true, we do not presently have a complete proof. If the conjecture is false, then some legitimate access may be denied. However, as a result of theorem 2, no access will improperly granted.

If the conjecture is true, then Java stack inspection, our access control decision procedure, and proving statements in our subset of ABLP logic are all mutually equivalent.

Theorem 3 (Equivalence to Stack Inspection) *The* **check** *decision procedure is equivalent to the Java stack inspection algorithm of section 5.2.4.*

Proof: The Java stack inspection algorithm (Figure 5.5) itself does not have a formal definition. However, we can treat the evolution of the system inductively and focus on the `enablePrivilege()` and `checkPrivilege()` primitives.

We wish to prove that given a Java stack S and its ABLP-modeled equivalent $M(S)$, and for all targets T, `checkPrivilege(`T, S`)` $\equiv$ **check**$(T, M(S))$.

Our induction is over the number of *steps* taken, where a step is either a procedure call or an `enablePrivilege()` operation. Steps are defined as operations on environments. For clarity, we ignore the existence of `disablePrivilege()`, `revertPrivilege()`, and procedure return operations; our proof can easily be extended to accommodate them.

We also assume Netscape semantics. A simple adjustment to the base case can be used to prove equivalence between the decision procedure and the Sun / Microsoft semantics.

Base case: In the base case, no steps have been taken. In this case, the stack inspection system has a single stack frame with no privilege annotation; in the ABLP model, the stack frame's security context is empty. In this base case, `checkPrivilege(`T, S_0`)` and **check**$(T, M(S_0))$ will both return false.

Inductive step: We assume that N steps have been taken ($N \geq 0$) and we are in a situation where both `checkPrivilege(`T, S`)` and **check**$(T, M(S))$ would yield the same result. Now there are two cases:

enablePrivilege(T) step: In the stack inspection system, this adds an *enabled-privilege*(T) annotation on the current stack frame. In the ABLP model, it adds $Ok(T)$ to the current security context (a part of $M(S)$).

If this `enablePrivilege()` call is followed by a call to `checkPrivilege(`T`)`, the Java stack inspection algorithm will succeed because the *enabled-privilege*(T) flag is immediately discovered. Likewise, a call to **check**$(T, M(S))$ will succeed because $Ok(T)$ is found in $M(S)$, $Ok(T)$ is what **check** is trying to prove.

If this `enablePrivilege()` call is followed by a call to `checkPrivilege(`U`)` with $U \neq T$, the new stack annotation or statement will be irrelevant to the result of either `checkPrivilege()` or **check**, so we fall back on the inductive hypothesis to show that both systems give the same result.

Procedure call step: Let P be the principal of the procedure that is called. In the stack inspection system, this adds to the stack an unannotated stack frame belonging to P. In the ABLP system, it prepends "P **says**" to the front of every statement in the current security context.

When `checkPrivilege(`T`)` is called, two things occur. First, the call is treated as a normal procedure call, with the caller's principal being prepended to the statements in the security context. Then, there are two sub-cases.

- ***P* is not trusted for *T*.** In the stack inspection case, `checkPrivilege(`T`)` will fail because the current frame is not trusted to access T. In the ABLP case, the **check** will deny access because every statement starts with "P **says**" and

P does not speak for *T*.

- ***P* is trusted for *T*.** In the stack inspection case, the stack search will ignore the current frame and proceed to the next frame on the stack. In the ABLP case, since $P \Rightarrow T$, the "*P* **says**" on the front of every statement has no effect. Thus both systems give the same answer they would have given before the last step. By the inductive hypothesis, both systems thus give the same result.█

7.2 Extensions to the Model

There are a number of cases in which Java implementations differ from the model we have described. These are minor differences with no effect on the strength of the model.

7.2.1 Groups

It is natural to extend the model by allowing the definition of groups. In ABLP logic, a group is represented as a principal, and membership in the group is represented by saying the member speaks for the group. Deployed Java systems use groups in several ways to simplify the process of defining policy.

The Microsoft system defines "security zones" that are groups of principals. A user or administrator can divide the principals into groups with names like "local", "intranet", and "internet", and then define policies on a per-group basis.

Netscape defines "macro targets" that are groups of targets. A typical macro target might be called "typical game privileges." This macro target would speak for those privileges that network games typically need.

The Sun system has a general notion of targets in which one target can imply another. In fact, each target is required to define an `implies()` procedure, which can be used to ask the target whether it signifies a superset of the privileges associated with another target. This can be handled with a simple extension to the model.

7.2.2 Threads

Java is a multi-threaded language, meaning there can be multiple threads of control, and hence multiple stacks can exist concurrently. When a new thread is created in Netscape's system, the first frame on the new stack begins with an empty security context. In Sun and Microsoft's systems, the first frame on the stack of the new thread is told the security context of the stack frame that created the thread in exactly the same way as what happens during a normal procedure call.

7.2.3 Enabling a Privilege

The model of `enablePrivilege()` in section 7.1.4 differs somewhat from the Netscape implementation of stack inspection, where a stack frame F cannot successfully call `enablePrivilege(`T`)` unless the local access credentials include $F \Rightarrow T$. The restriction imposed by Netscape is related to their user interface and is not necessary in our formulation, since the statement F **says** $Ok(T)$ is ineffectual unless $F \Rightarrow T$. Sun JDK 1.2's implementation is closer to our model.

7.2.4 Frame Credentials

Java implementations do not treat stack frames or their code as separate principals. Instead, they track only the public key that signed the code and call this the frame's principal. As we saw in section 7.1.1, for any stack frame, we can prove the stack frame speaks for the public key that signed the code. In practice, neither the stack frame nor the code speaks for any principal except the public key. Likewise, access control policies are represented directly in terms of the public keys, so there is no need to separately track the principal for which the public key speaks. As a result, the Java implementations say the principal of any given stack frame is exactly the public key that signed that frame's code. This means that Java implementations need not have an internal notion of the frame credentials described here.

7.3 Improved Implementation

In addition to improving our understanding of stack inspection, our model and decision procedure can help us find more efficient implementations of stack inspection. We improve the performance in two ways. First, we show that the evolution of security contexts can be represented by a deterministic pushdown automaton; this opens up a variety of efficient implementation techniques. Second, we describe *security-passing style,* an efficient and convenient integration of the pushdown automaton with the state of the program.

7.3.1 Security Contexts and Automata

We can simplify the representation of security contexts by making two observations about our decision procedure.

1. Interchanging the positions of two principals in any quoting chain does not affect the outcome of the decision procedure.

2. If P is an atomic principal, replacing $P \mid P$ by P in any statement does not affect the result of the decision procedure.

Both observations are easily proven, since they follow directly from the structure of the decision procedure.

We also use the observation in section 7.2.4 that we need not consider frame credentials, but need only consider the signer of a given stack frame. This means that multiple stack frames corresponding to the same signature will be considered to have the same principal.

It then follows that without affecting the result of the decision procedure we can rewrite each statement in the security context into a canonical form in which each atomic principal appears at most once, and the atomic principals appear in some canonical order. After this transformation, we can discard any duplicate statements from the security context.

Since the set of atomic principals is finite, the set of targets is finite, and no principal or target may be mentioned more than once in a canonical-form statement, there is a therefore finite set of possible canonical-form statements. It then follows that only a finite number of canonical-form security contexts may exist.

While the number of possible security contexts can grow exponentially in the number of principals and targets, it is nonetheless finite. Therefore, we can represent the evolution of a stack frame's security contexts by a finite automaton, where each state in the automaton corresponds to a security context. Since stack frames are created and destroyed in LIFO order, the execution of a thread can be represented by a finite pushdown automaton, where calling a procedure corresponds to a `push` operation (and a state transition), returning from a procedure corresponds to a `pop` operation, and `enablePrivilege()`, `disablePrivilege()` and `revertPrivilege()` correspond to transitions on the finite state graph.[2].

[2]One more nicety is required. To implement `revertPrivilege()`, we need to remember what the security state was when each stack frame was created. We can encode this information in the finite state, or we can store it on the stack by doing another `push` operation on procedure call.

Representing the system as an automaton has several advantages. It allows us to use analysis tools such as model checkers to derive properties of particular policies. It also admits a variety of efficient implementation techniques such as lazy construction of the state set and the use of advanced data structures.

Furthermore, the results of a security check can be stored along with the security contexts. So in cases where the same security check may be made numerous times (such as when one program opens a multitude of files), only the first check would require invoking the decision procedure. Subsequent security checks could consult a local cache and execute in constant time.

One concern is that, because the space of all possible security contexts is exponential in the number of principals and targets, the amount of memory needed will be similarly exponential. This concern is addressed by noting that very few of these security contexts will ever be used. A lazy implementation, one which only allocates memory for security contexts as they are needed, would only allocate memory proportional to the complexity of its security needs. Thus, if the execution of a program only uses a handful of distinct security contexts, only those few contexts will be allocated. Conversely, if a program is truly exponential in its security complexity, it would need to run for an exponential amount of time in order to cause the full security context space to be instantiated. Such degenerate cases are unlikely to occur in practice.

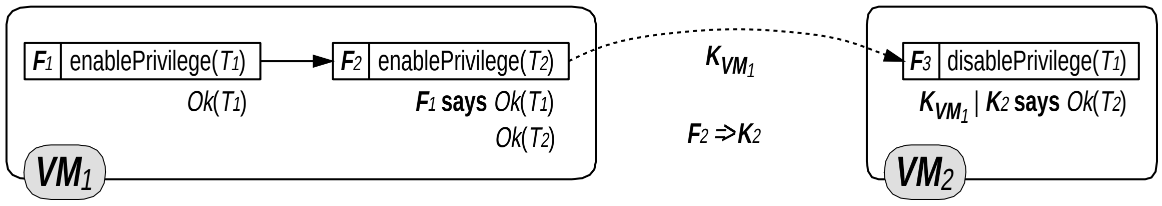
7.3.2 Security-Passing Style

The implementation discussed thus far has the disadvantage that security state is tracked separately from the rest of the program's state. This means that there are two subsystems (the security subsystem and the code execution subsystem) with separate semantics and separate implementations of pushdown stacks coexisting in the same Java Virtual Machine (JVM). We can improve this situation by implementing the security mechanisms in terms of the existing JVM mechanisms.

We do this by adding an extra, implicit argument to every procedure. The extra argument is a pointer into the finite-state space of the automaton. This eliminates the need to have a separate pushdown stack for security contexts or maintain stack annotations on the existing run-time stack. We dub this approach *security-passing style,* by analogy to continuation-passing style [Ste78], a transformation technique used by some compilers that also replaces an explicit pushdown stack with implicitly-passed procedure arguments. An implementation of security-passing style is presented in the following chapter.

The main advantage of security-passing style is that once a program has been rewritten, it no longer needs any special security functionality from the JVM. The rewritten program consists of ordinary Java bytecode that can be executed by any JVM, even one that knows nothing about stack inspection. This has many advantages, including portability and efficiency. The main performance benefit is that the JVM can use standard compiler optimizations such as dead-code elimination and constant propagation to remove unused security tracking code, or inlining and tail-recursion elimination to reduce procedure call overhead.

Another advantage of security-passing style is that it lets us express the stack inspection model within the existing semantics of the Java language, rather than requiring an additional and possibly incompatible definition for the semantics of the security mechanisms. Security-passing style also lets us more easily transplant the stack inspection idea into other language and systems.



F_1 enablePrivilege(T_1)
$Ok(T_1)$
F_2 enablePrivilege(T_2)
F_1 says $Ok(T_1)$
$Ok(T_2)$
K_{VM_1}
$F_2 \Rightarrow K_2$
F_3 disablePrivilege(T_1)
$K_{VM_1} \mid K_2$ says $Ok(T_2)$
VM_1
VM_2

Figure 7.2: Security contexts with a remote procedure call.
Example of interaction between stack frames via remote procedure call. Each rectangle represents a stack frame. Each stack frame is labeled with its name and its security context (after its call to `enablePrivilege()` or `disablePrivilege()`). The larger rounded rectangles represent separate Java virtual machines, and the dotted arrow represents the channel used for a remote procedure call.

7.4 Remote Procedure Calls

RPC security has received a good deal of attention in the literature. The two prevailing styles of security are capabilities and access control lists [TMvR86, Gon89, Hu95, Obj96, vABW96]. Most of these systems support only simple principals. Even in systems that support more complex principals [WABL94], the mechanisms to express those principals are relatively unwieldy.

This section discusses how to extend the Java stack inspection model across RPCs. One of the principal uses for ABLP logic is in reasoning about access control in distributed systems, and we use the customary ABLP model of network communication to derive a straightforward extension of our model to the case of RPC.

7.4.1 Channels

When two machines establish an encrypted channel between them, each machine proves that it knows a specific private key that corresponds to a well-known public key. When one side sends a message through the encrypted channel, we model this as a statement made by the sender's session key: we write K **says** s, where K is the sender's session key and s is the statement. As discussed in section 6.4, the public-key infrastructure and the session key establishment protocol together let us establish that K speaks for the principal that sent the message.

In order to extend Java stack inspection to RPCs, each RPC call must transmit the security context of the RPC caller to the RPC callee. Since each of the caller's statements is sent through a channel established by the caller's virtual machine, a statement S of the caller's frame arrives on the callee side as K_{CVM} **says** S, where K_{CVM} is a cryptographic key that speaks for the caller's virtual machine. The stack frame that executes the RPC on the callee is given an initial security context consisting of all of these arriving statements.

Note that this framework supports the intuition that a remote caller should not be allowed to access resources unless that caller's virtual machine is trustworthy. Any statement transmitted across the network arrives on the callee as a statement of the caller's virtual machine (or more properly, of its key); the callee will disbelieve such a statement unless it trusts the caller's virtual machine.

This strategy fits together well with security-passing style. The caller transmits its security context along with the normal RPC data. The security context is marshalled, transmitted, and unmarshalled like any other RPC data.

Figure 7.2 presents an example of how this would work. The Java stack inspection algorithm executes on the callee's machine when an access control decision must be made, exactly as in the local case.

7.4.2 Dealing with Malicious Callers

An interesting question is what an attacker can accomplish by sending false or misleading statements across a channel. If the caller's virtual machine is malicious, it may send whatever statements it wants, provided that they have the correct format. Regardless of the security context sent, each statement arrives at the callee as a statement made by the caller's virtual machine. If the callee does not trust the caller, such statements will not convince the callee to allow access.

Suppose a malicious caller's virtual machine *MC* wants to cause an access to target *T* on some callee. The most powerful statement *MC* can send to support this attempt is simply *Ok*(*T*)[3]; this will arrive at the callee as *MC* **says** *Ok*(*T*). Note that this is a statement that *MC* can make without lying, since *MC* is entitled to add *Ok*(*T*) to its own security context. Any lie that *MC* can tell is less powerful than this true statement, so lying cannot help *MC* gain access to *T*. The most powerful thing *MC* can do is to ask, under its own authority, to access *T*.

7.4.3 Dealing with Malicious Code on a Trustworthy Caller

Malicious code on a trustworthy caller also does not cause any new problems. The malicious code can add *Ok*(*T*) to its security context, and that statement will be transmitted correctly to the callee. The callee will then allow access to *T* only if it trusts the malicious code to access *T*. This is the same result that would have occurred had the malicious code been running directly on the callee. This matches the intuition that (with proper use of cryptography for authentication, confidentiality, and integrity of communication) we can ignore machine boundaries if the communicating processes trust each other and the platforms on which they are

[3]While the statement `false` would be more powerful than *Ok*(*T*), we assume the protocol for transmitting statements will not allow this. `false` is not a valid statement in our subset of ABLP.

running.

7.5 Conclusion

Commercial Java applications often need to execute untrusted code, such as applets, within themselves. In order to allow sufficiently expressive security policies, granting different privileges to code signed by different principals, the latest Java implementations now support a runtime mechanism to search the call-stack for code with different privileges and decide whether a given call-stack configuration is authorized to access a protected resource.

This chapter has presented a formalization of Java's stack inspection using a logic developed by Abadi, Burrows, Lampson, and Plotkin [ABLP93]. Using this model, we have demonstrated how Java's access control decisions correspond to proving statements in ABLP logic. We have reduced the stack inspection model to a finite pushdown automaton, and described how to implement the automaton efficiently using security-passing style. A new implementation of Java's stack inspection, based on security-passing style rather than stack annotations, is described in the next chapter.

We have also extended our model to apply to remote procedure calls and we have used the ABLP expression of this model to suggest a novel implementation for a Java-based secure RPC system. While the implementation of such an RPC system is future work, our model gives us greater confidence that the system would be both useful and sound.

Chapter 8

Security-Passing Style: Efficient Infrastructure for Access Control

Stack inspection has many benefits in structuring large systems with mutually untrusting authors. But its original definition, in terms of searching stack frames, had an unclear relationship to the actual achievement of security, over-constrained the implementation of a Java system, complicating the implementation of many desirable optimizations such as method inlining, tail recursion, and other interprocedural optimizations.

Our new semantics for stack inspection based on a belief logic and its implementation using the calculus of *security-passing style* solves all of these problems. We can efficiently represent the security context for any method activation, and we can build a new implementation by merely rewriting the Java bytecodes before they are loaded by the system. No changes to the Java runtime or bytecode semantics are necessary. With a combination of static analysis and runtime optimizations, our new implementation shows competitive performance, yet will not interfere with future aggressive optimizing compilers.

$$\begin{array}{llcl}
(1) & SPS_{\text{fun}}(\text{function } f(a_1,\ldots,a_n) = E) & = & \\
 & \multicolumn{3}{r}{\text{function } f(a_1,\ldots,a_n,s) = (\text{let } s' = \textbf{says}(\textbf{owner}(f),s) \text{ in } SPS(E,\ s'))} \\
(2) & SPS(p.g(x_1,\ldots,x_m),\ s') & = & p.g(x_1,\ldots,x_m,s') \\
(3) & SPS(E_1 + E_2,\ s') & = & SPS(E_1,s') + SPS(E_2,s') \\
(4) & SPS(\text{BeginPrivilege } E,\ s') & = & SPS(E, Ok()) \\
(5) & SPS(\text{CheckPrivilege}(T),\ s') & = & \textbf{check}(T,s')
\end{array}$$

Figure 8.1: “Caller-says” SPS conversion.

This chapter presents our new implementation and compares its performance to traditional stack inspection implementations using the `javac` compiler and our code transformer itself as benchmarks.

8.1 The Security-Passing Style Transformation

This section describes the design of the security-passing style transformation. Note that we are using a somewhat simpler version than that described in previous chapters, which is closer to the design of Sun’s JDK 1.2 system. This system only supports BeginPrivilege(), CheckPrivilege(), and function calls. Furthermore, in the simplified model, one may only enable privileges for a specific root target T_{root}, where $\forall targets T_x : T_{root} \Rightarrow T_x$. As a shorthand, we write BeginPrivilege() with no target argument and speak of $Ok()$ with no target.

8.1.1 SPS Conversion

For this chapter we assume a simple programming language with methods or functions:

$$P \rightarrow \text{function } f(a_1,\ldots,a_n) = E$$

$$P \rightarrow P\ P$$

$$\begin{aligned}
E &\rightarrow p.g(x_1, \ldots, x_m) \\
E &\rightarrow E + E \\
E &\rightarrow \text{let } v = E \text{ in } E \\
E &\rightarrow \text{BeginPrivilege } E \\
E &\rightarrow \text{CheckPrivilege}(T)
\end{aligned}$$

where a program is a collection of function definitions; a function body contains function / method calls as well as arithmetic expressions and (not shown here) sequencing statements. The (BeginPrivilege E) statement asserts privileges for the dynamic extent of the execution of E, and CheckPrivilege(T) checks whether the target T is currently accessible. We use lower-case letters to range over program variables and T to stand for target names.

We assume that each method f has an owner (principal) **owner**(f). Ownership is associated with code, not classes: in an object-oriented language, if object p belongs to class C_2, which inherits method-body f from superclass C_1, then **owner**($p.f$) is C_1, not C_2. In principle, this could go either way, but the designers chose to use the concrete implementation's owner because it is easier to ascertain at runtime, and it avoids the danger that a privileged method may call what it thinks is a method of the same class, yet is actually a method in a subclass. This would be an example of a *luring attack* which we wish to avoid (see section 5.2.4).

Figure 8.1 shows the rules for converting a program to security-passing style. The conversion SPS_{fun} is applied to each function; SPS is applied to each expression. Rule 1 involves the introduction of new local variables s and s' whose names are not used elsewhere. The function f is rewritten to take s as a new formal parameter, the *security context*, which will be the representation of a statement in the

$$
\begin{array}{llcl}
(1) & SPS_{\text{fun}}(\text{function } f(a_1,\ldots,a_n) = E) & = & \text{function } f(a_1,\ldots,a_n,s) = SPS(E,\ s) \\
(2) & SPS(p.g(x_1,\ldots,x_m),\ s) & = & p.g(x_1,\ldots,x_m,\mathbf{says}(\mathbf{owner}(p.g),s)) \\
(3) & SPS(E_1+E_2,\ s) & = & SPS(E_1,s)+SPS(E_2,s) \\
(4) & SPS(\text{BeginPrivilege } E,\ s) & = & SPS(E,Ok()) \\
(5) & SPS(\text{CheckPrivilege}(T),\ s) & = & \mathbf{check}(T,s)
\end{array}
$$

Figure 8.2: “Callee-says” SPS conversion.

ABLP logic. We then construct a new security context with $s' = \mathbf{owner}(f)$ **says** s, and SPS-convert the body of the function using s' for all outgoing function calls.

Rule 2 of Figure 8.1 shows the use of s' as the “extra” argument of an outgoing call; rule 3 shows that most statements are unaffected by SPS-conversion. Rule 4 shows that BeginPrivilege discards the security context s' and simply uses $Ok()$; rule 5 shows that CheckPrivilege invokes the decision procedure *check*, described in section 7.1.5.

To complete the definition of SPS conversion, we assume that the `main` function of the converted program is called with a security context allowing no access to any target (following Netscape) or full access to every target (following Sun).

8.1.2 Rewriting Java Bytecodes

Stack inspection was originally implemented at Netscape (and then at Sun and Microsoft) by adding support for it to the runtime system. These extensions required changing the stack frame representation, which in turn affected the garbage collector and JIT compiler.

With SPS conversion, we can express the stack-inspection security architecture in “vanilla” Java bytecodes (or source), without stack-frame marks or any other constraints on the Java virtual machine implementation. Every method has an extra parameter for passing the security context, but this parameter and its representation are just Java.

8.2 Optimization

Netscape and Sun's implementation of stack inspection — by marking frames at BeginPrivilege() and scanning frames in CheckPrivilege() — has very low cost for the vast majority of methods, which do not perform either operation. There is a difficult-to-measure cost of their scheme, in that it may inhibit useful optimizations such as inlining, tail-call optimization, and certain interprocedural optimizations. Also, their system has a linear-time cost for CheckPrivilege(), which must scan a potentially unbounded number of frames to recover the security context. After this, with the current JDK 1.2 semantics, analyzing the security context could be potentially as bad as $O(N^2)$ in the number of targets because Sun allows the target speaks-for graph to change over time. If Sun allowed caching of the speaks-for graph (see section 5.2.4), then a transitive closure could be computed once, allowing the final security check to be linear in the size of the security context, which will typically be quite small.

Our semantics costs $O(1)$ per operation (with the same caveats about checking privileges), since a security context has a bounded-size representation. Even so, in order to achieve competitive performance we must minimize the constant-time overhead on each method call. We achieve this by a combination of static optimizations and dynamic caching. By applying static analysis to the full program before it begins running, we can optimize away many of the computations of a new security state.

8.2.1 Caller-says vs. callee-says

We have two variants of SPS-conversion; Figure 8.1 shows a "caller-says" convention, in which a call from g to f involves a computation by g (the caller) of $\mathbf{says}(\mathbf{owner}(g), s)$. Figure 8.2 shows a "callee-says" convention, in which a call from g to f involves a computation by g of $\mathbf{says}(\mathbf{owner}(f), s)$.

Either of these conventions is semantically equivalent to stack inspection, but slightly different compile-time optimizations apply, as we will show. For example, in caller-says, **owner**(this.f) can always be computed at compile time; but in callee-says SPS conversion of an O-O language, **owner**(p.g) requires either dynamic method lookup or static analysis.

8.2.2 Static Optimizations

Suppose a function body $g(\ldots, s)$ contains a call $f(\ldots, s')$, where s and s' are the security-context arguments. The method g must compute $s' = \mathbf{says}(\mathbf{owner}(g), s)$ (in a caller-says convention) or $s' = \mathbf{says}(\mathbf{owner}(f), s)$ (in a callee-says convention). Under certain circumstances, we can let $s' = s$:

- In caller-says, we know that f must compute $s'' = \mathbf{says}(\mathbf{owner}(f), s')$ and then perform operations on s''; f cannot use s' in any other way. If $\mathbf{owner}(g) \Rightarrow \mathbf{owner}(f)$, that is, if the privileges of g are a superset of the privileges of f, then
$$\begin{array}{rcll} s'' & = & \mathbf{says}(\mathbf{owner}(f), s') & \equiv \; F \;\mathbf{says}\; s' \\ & = & \mathbf{says}(\mathbf{owner}(f), \mathbf{says}(\mathbf{owner}(g), s)) & \equiv \; F \;\mathbf{says}\; G \;\mathbf{says}\; s \\ & = & \mathbf{says}(\mathbf{owner}(f), s) & \equiv \; F \;\mathbf{says}\; s \end{array}$$
by virtue of the fact that $G \Rightarrow F$, allowing substitutions based on the ABLP axioms. As a result, g can call f with the security context s instead of s',

eliminating the computation of the **says** function inside g.

- In callee-says, we know that s was constructed by the caller of g as $s = \textbf{says}(\textbf{owner}(g), t)$. If $\textbf{owner}(g) \Rightarrow \textbf{owner}(f)$, then

$$\begin{array}{lllll} s' & = & \textbf{says}(\textbf{owner}(f), s) & \equiv & F\ \textbf{says}\ s \\ & = & \textbf{says}(\textbf{owner}(f), \textbf{says}(\textbf{owner}(g), t)) & \equiv & F\ \textbf{says}\ G\ \textbf{says}\ t \\ & = & \textbf{says}(\textbf{owner}(g), t) & \equiv & G\ \textbf{says}\ t \\ & = & s & & \end{array}$$

  again using the ABLP axioms. As a result, g can call f with the security context s instead of s', eliminating the computation of the **says** function inside g.

In practice, it is very common for one function to call another with the same owner; in such cases, no **says** computation is necessary (since $\textbf{owner}(f) \Rightarrow \textbf{owner}(f)$).

Criterion for choosing caller-says vs. callee-says. Caller-says requires calculation of the owner of the currently executing code, and can be statically optimized if the caller is known to speak for the callee. Callee-says requires fetching the owner of the callee, and can be statically optimized if the callee speaks for the caller. Depending on how often these different speaks-for relations can be statically determined, and how often the owner of the callee can be determined statically, one convention or the other may turn out to perform best in practice. We only implemented the callee-says style.

Static identification of ownership in class-based object-oriented languages. In an object-oriented program, a program variable `p` declared to be of class `C` may point to to an object of any subclass of `C`. Therefore, the method call `p.f()` may invoke any of several actual method bodies, depending upon how `f` is overridden

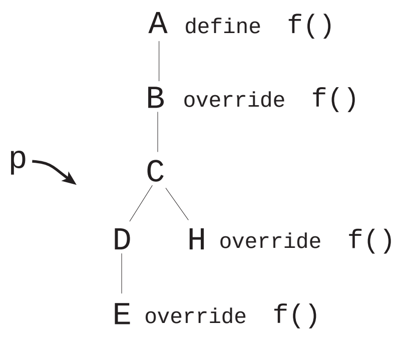


Figure 8.3: Class hierarchy analysis.
Variable p of static type C may point to an object of class C, D, E, or H; the owner of $p.f()$ may be the owner of B, E, or H.

in the subclasses of `C`.

Figure 8.3 illustrates a simple *flow-insensitive class hierarchy analysis*. Given a variable `p` of static type `C`, we analyze a call `p.f`() as follows. From `C`, we walk up the class hierarchy tree to find the lowest (improper) ancestor of `C` that implements or overrides `f`(), and put that ancestor into the set P. Then we examine all (direct and indirect) subclasses of `C`, and any of those that override `f`() are also put into P.

A static analysis of the program may be able to narrow the set of possible types that `p` may take on at the site of the call `p.f`(), and this in turn narrows the set of possible method bodies (callees) that this call site may invoke. Such an analysis can speed up a conventional object-oriented program because dynamic method lookup is more expensive than a static procedure call; if the set of callees can be narrowed to a singleton, then the call `p.f`() can be implemented without run-time lookup. Other kinds of program optimization – interprocedural dataflow analysis, or function-call inlining – also benefit from knowledge of which method-body is called.

For security-passing style, it is not necessary to narrow the set of possible method bodies to a singleton – it suffices to prove that all possible method bodies for this call to f have a common owner. In fact, an even weaker property will suffice: for caller-says, we require only that every possible owner of f have (nonstrictly) fewer privileges than the owner of its caller, g; for callee-says, we require that all owners of f have (nonstrictly) more privileges than the owner of g.

There has been much work on static analyses of object types. *Class hierarchy analysis* [Fer95, DGC95] simply examines all the subclasses of C to see if any of them overrides method f. If not, the definition of f in class C (or, if C does not define f, the definition of f in an ancestor class) must be the callee.

Information gained from dataflow analysis can prune the set of object types that p may contain at the call site; this in turn prunes the set of possible method bodies for f at this point, which in turn allows more precise dataflow analysis. This iterative process is called *interprocedural class analysis* and has been shown practical by at least two independent sets of authors [DMM96, DGC98].

System speaks for everyone. Some access-control matrices contain a principal *System* that has access to every target. Even without flow analysis or hierarchy analysis the compiler can use the rule $System \Rightarrow C$ to eliminate **says** computations when System code is calling other methods (in the caller-says convention) or other code is calling System code (in callee-says).

Leaf procedures. Many functions do not use their security context in any way. A *leaf procedure* is one that makes no other function call and contains no Check-Privilege operations; its security context argument is statically dead at all times. A *generalized leaf procedure* is one that neither calls CheckPrivilege nor any native methods, either directly or indirectly. Static analysis of the dynamic call tree can conservatively identify many generalized leaf procedures; these procedures do not require any security-context argument or a **says** computation.

The generalized leaf procedure analysis works by recursively following `invoke` bytecodes from every method (to a limited recursion depth) and is repeated until a fixed point is reached. In practice, thousands of methods can be analyzed this way in less than one CPU second.

This optimization is just a form of interprocedural dead code elimination, and can be done by a conventional object-oriented compiler (after SPS conversion) because security-passing style has expressed all the **says** computations in the underlying programming language. However, the compiler needs to know that **says** has a declarative/functional semantics: calls to **says** can be deleted if the result is dead, even though **says** might have internal side effects to lazily compute part of the transition graph.

Unfortunately, the `invoke` bytecode instructions are not sufficient for the leaf analysis. If a `getstatic` or `putstatic` bytecode references a class that has not yet been loaded, this will cause the class to be *initialized*, creating an implicit call to the target class's initialization method. Likewise, various runtime exceptions, such as indexing beyond an array's boundary or dereferencing a null pointer, will throw exceptions. To throw an exception, an implicit call to the method's constructor would occur.

These implicit method invocations are not directly visible from Java bytecode, making it difficult to pass the security context to the new method. To address class initializers, we observe that, when a class is first loaded, it cannot make any assumption about who caused it to be loaded. Class initializers should thus be pessimistic about any security context they might inherit. Our system gives class initializers the same security context they would receive after an EnablePrivilege operation. While not completely compatible with Sun's implementation, this solution simulates a *possible* outcome with which a class initializer should be prepared to cope.

The implicit calls to create runtime exceptions are much simpler. By observation, all of the exceptions that might be thrown make only one native method call, to fill in their stack trace, and are otherwise generalized leaf methods. Because the security context is never used by these exception constructors, it is safe to allow these implicit method invocations.

8.2.3 Dynamic Optimizations

There are a finite number of security contexts s, each corresponding to a subset of the n principals in the system. For a simple browser-applet system, $n = 2$. Each context can be represented with a finite table of labeled out-edges, so that **says**(o, s) is computed by looking up o in the table for s.

Although n is bounded, it may not always be tiny (*e.g.,* a stock market with thousands of principals), so we lazily compute the tables and represent only those security contexts that are actually reached. Following an untraversed edge requires (1) looking up a "new" subset in a global hash table to see if this context has been reached before, (2a) using the context-pointer from the table or (2b) creating a new context data structure, and (3) installing the edge into the context that had lacked the edge.

From a security context s there be many consecutive **says** computations by the same principal. In the representation of s we maintain a dynamic one-word cache (o, s') indicating that the most recent **says** calculation on s was **says**$(o, s) = s'$. This should speed up the common case.

8.2.4 Open vs. Closed World Assumptions

Our system, as we have described it, currently makes a fundamental assumption that we can inspect all code before execution begins. This is often called a “closed world assumption.” In systems where Java’s security features are often used, such as Java applets or servlets, new code may arrive at any time. Currently, all of our algorithms have been designed for a closed world. In particular, our class hierarchy analysis runs once, up front, and code is then generated based on properties true in the closed world. Dean *et al.* [DGC95] discusses precisely this issue and proposes a scheme for incrementally updating the analysis.

Keep in mind that the performance numbers in section 8.3 are based on a closed world. Generally speaking, an open world has strictly less information available from which to infer that an optimization is legal. This implies that, in general, an implementation of security-passing style built for an open world would have strictly worse performance than one built for a closed world, although it may be possible for an open world system to closely approach the performance of a closed world system. For example, in a Java system supporting dynamic code recompilation, such as Sun’s forthcoming HotSpot [Gri98], it would be possible for an SPS incremental analysis to determine that certain classes, compiled using optimizations that are now invalid, should now be recompiled.

8.3 Implementation and Performance

We have implemented SPS conversion as a transformation on a collection of Java class files. Our implementation was built using the JOIE library (Java Object Instrumentation Environment) from Duke University [CCK98]. This library presents a relatively high-level interface to parse and edit Java class files.

We have written approximately 1700 lines of Java code to do static analysis, and 2300 lines to do byte-code rewriting (SPS conversion). Our runtime support (implementing the **says** and CheckPrivilege functions) is 1900 lines. Our system loads, analyzes, and rewrites roughly 800 Java classes in 100 seconds. We made no effort to tune the performance of the rewriter itself; achieving an order of magnitude improvement in rewriting speed should not be unreasonably difficult.

At present we have implemented flow-insensitive class hierarchy analysis to eliminate **says** computations, and we remove **says** computations from both simple and generalized leaf methods. Because we require the full program for this analysis, we cannot presently support the dynamic loading features of Java (see section 8.2.2). Instead, we run the program from local disk with our specialized classes.

Our system runs by modifying the class libraries of the NaturalBridge BulletTrain Java compiler [Nat98]. BulletTrain uses a traditional static compiler to produce native machine executables, in contrast to the dynamic just-in-time compilation used by other Java implementations. BulletTrain currently requires the whole program to be available at compile-time, although their runtime can support dynamic loading in future releases. We chose to use BulletTrain because its authors offered us invaluable assistance with their unreleased product. Also, because BulletTrain has an aggressive code optimizer which uses whole-program analysis, we believe performance numbers measured today with BulletTrain will represent what other Java systems can achieve in the future.

8.3.1 Making SPS Work

Security-passing style has some very nice theoretical properties, but actually implementing it requires a number of difficult cases to be handled properly.

Native methods. Java programs can call *native methods* (functions not written in Java) that might then call back to Java methods. We cannot apply SPS conversion within the native methods. Instead, when calling from Java to native, we store the security context s into a per-thread global variable; when calling from native back to Java we fetch s as the security context for the Java code. If we assume that all native method calls have the owner, *System*, then since **says**$(System, s) = s$ this is the correct behavior.

We must also support up-calls, where native methods choose to call back into Java methods. The standard mechanism for this, JNI (Java Native Interface), requires the native code to specify a method's complete signature, including the types of its arguments and return value. This means the SPS-converter must generate stubs with the original signatures to receive a JNI up-call. A stub method will retrieve the per-thread stored security context and then invoke its SPS-converted sibling.

Reflection. Ideally, the security-passing transformation should not be visible in any way to an application. The Java *reflection* API allows a program to learn how many parameters each of its methods takes; since SPS conversion introduces extra arguments, this is a problem that would have to be fixed by modifying the implementation of reflection; we have not yet done this.

Bootstrapping. In practice, bootstrapping proved to be the most difficult aspect of implementing security-passing style. In the BulletTrain system, the majority of the bootstrapping code is written in Java itself. This makes the system extremely sensitive about the order in which classes begin execution, and many classes which appear to be normal are handled specially by the compiler. To address these concerns, an SPS-converted program must bootstrap in three stages.

Classes involved in the very beginning of bootstrapping the runtime were identified by hand and added to a list of classes that are not modified by the SPS converter. Instead, any calls to these classes are treated the same as calls to any native method, storing the security context into a per-thread global variable.

In the second phase of bootstrapping, some SPS-converted classes begin execution but the SPS runtime itself is not yet initialized. Still, SPS-converted classes require a non-null instance of security context be passed to them. To avoid this chicken-and-egg problem, a "dummy" security context, later subclassed to implement the real security context, is created.

Finally, when "real" security contexts are available, the application's main routine can be invoked with a proper security context and execution continues normally.

Consistency and inheritance. Because many system classes must not be SPS-converted, an issue arises when an SPS-converted class subclasses a non-converted class or vice versa. It is obviously important to maintain the consistency of the type system, and SPS-converting only a subset of the classes can cause confusion.

To solve the problem, we adopted a rule that, if a class is SPS-converted, then all subclasses of it must also be SPS-converted. Likewise, if an interface is SPS-converted, then all classes that implement that interface must also be SPS-converted.

This rule implies that, if a class *cannot* be SPS-converted, its superclass may not be converted, either. Therefore, several core classes, include `java.lang.Object`, execute unchanged. If a method in an SPS-converted class wishes to call a method in a non-SPS-converted class, it treats the call in the same way native methods are handled: the security context is saved and the method is invoked without the security context argument.

Several issues must still be resolved to make this work. One problem is that `java.lang.Thread`, which must not be SPS-converted, implement the `java.lang.Runnable` interface, which we want to SPS-convert. This issue does not occur anywhere else, so it was solved by adding a new method specifically to `java.lang.Thread`.

Another problem arises when an SPS-converted subclass inherits a method from a non-SPS-converted superclass without overriding it. If we do nothing, certain Java features (method invocation on interfaces and reflection) will fail. This happens frequently enough that we solve the problem by automatically generating stubs in the subclass which explicitly delegate to the superclass.

Portability. As mentioned above, Java systems are relatively fragile during the bootstrapping process. This requires a number of classes to be handled specially. Running SPS-converted code in a different Java environment would require assessing which classes need to be handled specially. Also, sufficient access to the system bootstrapping process is required such that the SPS system can be loaded as early as possible. Aside from these issues, the SPS runtime should be straightforward to add to any Java system.

	No Security (baseline)	Stack Inspection	Security-Passing Style
No **says** (leaf)	0	0	1-4 cycles
says$(o, s) = s$ (static opt.)			1-4
says(o, s) (cache hit)			33
says(o, s) (cache miss)			69
BeginPrivilege	24 cycles	2200 cycles	57

Table 8.1: Measured cost of SPS primitives.
The **says** function is shown as it would be implemented in a leaf method (no security-context argument), in a non-leaf with identical caller and callee owners, and (without static optimization) with and without a hit in the one-word cache.
BeginPrivilege includes the cost of invoking an interface method, as part of the latest JDK 1.2 semantics (see section 5.2.4.
Cycle counts were measured by timing microbenchmarks, then dividing by the computer's clock cycle.

SPS Converter Benchmark

	No Security (baseline)	Normal Stack Inspection	SPS Conversion (no checks)	SPS Conversion (with checks)
Runtime (sec)	94.25	97.59	104.34	106.55
Stddev	0.75	3.06	1.46	0.95
Overhead	0%	3.55%	10.71%	13.05%

Javac Benchmark

	No Security (baseline)	Normal Stack Inspection	SPS Conversion (no checks)	SPS Conversion (with checks)
Runtime (sec)	15.53	15.77	17.93	18.13
Stddev	0.26	0.09	0.18	0.08
Overhead	0%	1.60%	15.50%	16.78%

Table 8.2: Runtime performance of benchmark programs.
We chose two programs which do a fair amount of computation as well as file operations. Security checks occur when files are opened for reading and writing.
The SPS Converter is the program which does the code rewriting, and it makes an interesting benchmark as well. Javac is the compiler from Sun's JDK 1.2 distribution. Both benchmarks were compiled with the NaturalBridge BulletTrain compiler. The table compares the NaturalBridge implementation of stack inspection with security-passing style. Likewise, each benchmark was run with the SecurityManager enabled and disabled. This difference represents the cost of privilege checking.

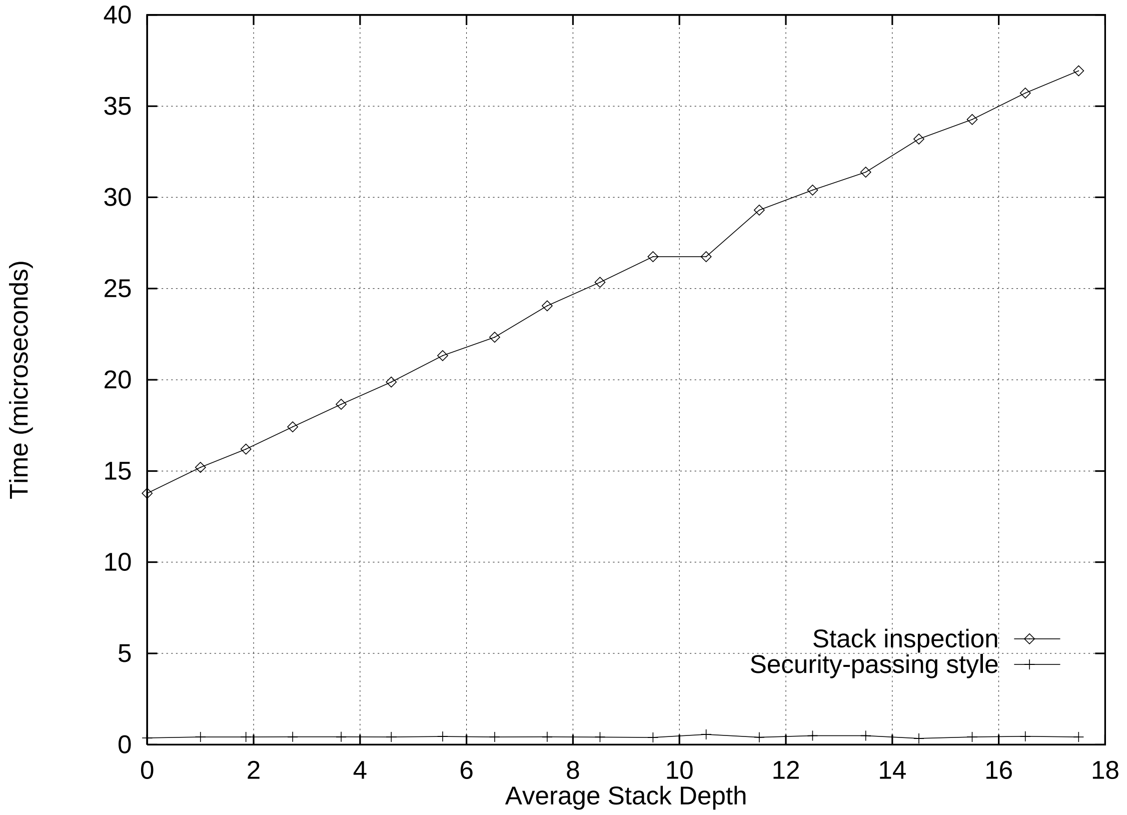


Figure 8.4: Cost of CheckPrivilege() microbenchmark.
The times for the SPS check privilege calls are approximately 0.5μsec. This microbenchmark compare NaturalBridge's internal stack inspection system with our security-passing style implementation. The microbenchmark is an implementation of the recursive solution to the Towers of Hanoi problem. Both implementations were measured on a 450MHz Pentium II, so 0.5μsec is approximately 225 CPU cycles.

8.3.2 Performance

To accurately measure the performance of the original stack inspection primitives as well as their SPS-converted equivalents, we created a series of microbenchmarks that repeatedly enable a privilege, perform a number of recursive method calls, then check the privilege.

All benchmarks were measured on a PC with 384MB of RAM and a 450MHz Intel Pentium II running Windows NT Workstation 4.0 and a pre-release version of the NaturalBridge BulletTrain Java compiler.

We used the measured run times of our microbenchmarks to calculate the cycle count of each stack-inspection primitive in each of three implementations: the null implementation (no security passing, no security checking); the BulletTrain implementation of stack inspection; and our security-passing style. Each microbenchmark was executed ten million times, allowing Java's millisecond-accurate timer to resolve single-cycle differences in execution time. Table 8.1 shows the results. Figure 8.4 shows the variable cost of the CheckPrivilege() primitive when using stack inspection compared to the constant-time cost of CheckPrivilege() with security-passing style. The performance difference varies from a factor of 35 to a factor of 88, depending on the stack depth!

Despite the success of security-passing style on microbenchmarks, the per-method overhead hurts it when running real applications. Table 8.2 compares SPS-converted code, with its cheap security checks, to normal code, performing expensive stack inspections for its security checks. As benchmarks, we used Sun's `javac` Java compiler and our own SPS converter. Both benchmarks do a fair amount of file reading and writing, requiring a security check for each file opened. Each benchmark was executed ten times; we show performance numbers with the average and standard deviation of their runtimes. On these benchmarks, the BulletTrain stack inspection system imposed an overhead that varied from 1.60% to 3.55% of the total runtime. In contract, security-passing style imposed an overhead from 10.71% to 15.50% without even making any security checks. The security checks themselves imposed an additional overhead ranging from 1.2% to 2.3%.

Currently, both our security-passing system and the BulletTrain stack inspection system are prototype implementations. Both "cheat" by returning `ProtectionDomain` structures which grant permission for any request and neither has been heavily tuned for performance. Kenneth Zadeck of NaturalBridge claims the BulletTrain implementation will be much simpler to tune for performance [Zad98]. Certainly, if our benchmarks represent typical usage patterns for security checking, stack inspection is probably the most efficient option. However, if an application requires a dramatically higher number of security checks, our microbenchmarks indicate security-passing style can offer orders of magnitude better performance.

8.4 Ideal Performance

We have measured the performance of security-passing style as a transformation on Java bytecodes, which are then compiled using the NaturalBridge BulletTrain compiler (see section 8.3). Although this is a good compiler, it is not as efficient as we might expect from hand-coded assembly language. Therefore, the cost of our security-passing operations and of ordinary program execution are both higher than they should be.

We therefore present an estimate of the cost of security-passing style in a com-

		SPS Converter		Javac	
		(*stat*)	(*dyn*)	(*stat*)	(*dyn*)
a	Leaf methods	18.42%	84.47%	18.90%	56.92%
b	Statically identifiable dominated callee	93.76%	41.16%	88.27%	70.40%
c	Save / restore cost	1 instruction *(estimated)*			
d	Fetch owner	1 instruction *(estimated)*			
e	Cache miss rate (one word cache)[a]	0.649	0.649	1.146×10^{-6}	1.146×10^{-6}
f	Cache test cost	3 instructions *(estimated)*			
g	Fetch target	1 instruction *(estimated)*			
h	Look in table	10 instructions *(estimated)*			
z	Total overhead (instructions / call)	1.368	1.146	1.287	1.068

Table 8.3: Estimated ideal costs for security-passing style.
stat indicates static frequencies in the program text, and *dyn* indicates measurements weighted by dynamic execution frequency.

[a]Static analysis cannot be used to estimate dynamic cache miss rates, so the dynamic number was used in the static column.

piler assumed to generate the smallest and fastest possible code for the SPS primitives. These estimates are based on dynamic performance counters which were added to the benchmarks discussed in section 8.3.2 and are summarized in table 8.3.

1. Static analysis will identify between 18% and 19% of methods as leaf methods which corresponds to between $a = 57\%$ and $a = 85\%$ of runtime method invocations; no security-context argument is needed to call a leaf method. Of the remaining non-leaf methods, class hierarchy analysis allows us to replace $b = 41\%$ to $b = 70\%$ of the **says** computations with the identity function.

2. The security context will be kept in a caller-save register; a procedure that makes more than one call must save its security context variable into the activation record before the first call, and fetch it back each subsequent call.

We approximate the register save / restore cost by assuming $c = 1$ instruction per call.

3. The security context is usually passed in the same register, so that a method `g`$(\ldots, s)$ needs no move instructions to call `f`$(\ldots, s)$. All **says** computations are inlined.

4. Each class descriptor contains one field that points to the owner of that class. When executing a dynamic method call `p.f()`, the class descriptor of `p` must be fetched anyway, so there is an estimated cost of $d = 1$ instruction to fetch the owner for a **says** computation.

5. When **says**(o, s) is calculated, a one-word cache (inside s) is used to remember the previous result (see section 8.2.3). Our dynamic traces indicate near perfect hit rates in the `javac` benchmark and fairly poor hit rates for the SPS converter (missing nearly $e = 65\%$ of all tests). The cost of testing the cache is estimated to be $f = 3$ instructions (fetch owner from context cache, compare, branch) and the cost of processing a hit is estimated to be an additional $g = 1$ instruction (fetch target from one-word cache).

6. Each security context contains a data structure (list, hash table, ...) holding the outgoing edges (labeled by owners) in the finite-state graph of security contexts. The average number of outgoing edges actually present (they are calculated lazily) is approximately one. Querying this data structure (and updating the context's cache) takes an estimated $h = 10$ instructions.

7. When there is no appropriately labeled outgoing edge already present in the (lazily computed) security context, it must be computed as described in section 8.2.3. Since this situation can happen only a bounded number of times

	No Security (baseline)	Method Invocations	Estimated Overhead
SPS Converter	94.25 sec	109×10^6	0.27 – 1.1 sec (0.28% – 1.1%)
Javac	15.53 sec	23×10^6	0.056 – 0.22 sec (0.36% – 1.4%)

Table 8.4: Estimated runtimes of benchmarks using "optimal" SPS code generation.

— see section 7.3 — its amortized cost is negligible.

Under these assumptions, the average cost of **says** computations per method invocation is

$$z = (1 - a)(c + (1 - b)(d + f + ((1 - e)g + eh)))$$

instructions, which should add just over one instruction per method, on average, based on our experiments.

8.4.1 Estimated Benchmark Performance

Based on these "optimal" overhead figures, we would like to estimate the runtime overhead while running real programs. This requires measuring the number of actual method calls made during a program's execution and then adding the estimated cost per method of SPS-converted code. The main difficulty is converting from some number of instructions to an equivalent number of CPU clock cycles. Modern CPUs can potentially execute multiple instructions per clock, but load and store operations may take additional time, depending on whether the desired memory address is cached or a host of other factors. In many cases, the latency necessary to compute an intermediate value can be hidden if the value is not yet needed and other instructions are ready to run. In the case of SPS-conversion, using caller-says semantics, the computed security context is not necessary until the first callsite is reached, so it is entirely feasible for the cycle cost of SPS to be low.

However, measuring this in practice is beyond the scope of this thesis (and could vary widely from one CPU to another). Instead, we present what the overhead would be if each instruction for SPS conversion consumed exactly one clock cycle (an optimistic estimate) or exactly four clock cycles (a pessimistic estimate).

We use our estimated overhead of approximately 1.1 instructions per method invocation to infer the actual runtime cost of the benchmarks presented in table 8.2. These results, summarized in table 8.4, are based on a 450 MHz system clock. With either benchmark, we estimate the overhead of an ideal SPS converter would add between a quarter of a percent to at most 1.4% to the program's runtime. This overhead would be competitive with implementations using stack inspection, in applications like our benchmarks, yet would likely maintain this lower overhead even in systems which made security checks more often.

8.5 Conclusion

Security-passing style is a simple semantics for the "stack inspection" security architecture. Users can reason about the security of their systems using ABLP belief logic, and implementors can reason about interprocedural optimization using the semantics of the original source language, in the SPS-converted program. The implementation cost is quite reasonable – even in our prototype with an unsophisticated and SPS-unaware compiler – and with better compilers it should be possible to implement security-passing with an overhead of approximately one instruction per call.

In the future we plan to try adding other information to the security context to control resource usage by callees. We will handle classes with multiple signers using a previously suggested scheme [WF98]. We also plan to use flow-sensitive class hierarchy analysis to improve the static optimization of SPS, and integrate our system into a (dynamically linking) Java ClassLoader (although this would require VM hooks to invalidate the optimizations when new classes are loaded that contradict our previous flow analysis).

Chapter 9

Future Work

This dissertation has described a number of different security architectures as applied to Java and similar languages and has focused on stack inspection as a valuable technique for access control. Security-passing style was shown to be an efficient and sensible way of implementing stack inspection, raising a number of interesting ideas for future work. Currently, the additional argument to each procedure is used strictly for passing the security context. An interesting extension would be to augment the security context with other information. Perhaps the security context could additionally help with resource allocation by specifying something about who is about to allocate memory and on whose behalf (similar to the per-thread resource tracking in JRes [Cv98]). Similarly, security contexts might prove useful for tracking CPU usage, allowing code to declare to a scheduler that it is consuming cycles itself, or on the behalf of its caller. A system that combines security-passing style with its scheduler and memory management may have interesting properties.

The static optimization presented here is fairly primitive, failing to use flow-variant code analysis or other more aggressive techniques. It would be useful to study the ability of a more aggressive optimizer to further reduce the overhead of security-passing code. Likewise, the system presented here does not account for dynamic loading of code. In current Java implementations, new code may arrive at any time and violate previously valid properties about the class hierarchy. It would be useful to build a system that maintained a dynamically changing static analysis of the class hierarchy and study how often properties we rely on are invalidated. Given a Java runtime with dynamic recompilation of code, it would additionally interesting to try regenerating a class's code when an optimization used in SPS-converting it becomes invalid.

It would also be useful to build an extension to an existing RPC system to support security-passing style and integrate that with a local security-passing system. As more distributed systems are deployed, it would be beneficial to apply the security we can achieve in a local system to the distributed case.

Ultimately, a number of research opportunities exist in the area between traditional operating systems and traditional language runtimes. Operating systems are increasingly adopting mobile code, dynamic linking, and object-oriented semantics. Languages are increasingly dealing with issues of access control and fair resource allocation. Interesting solutions will be found all across the spectrum between languages and operating systems. A number of commercial systems from the 1960s blurred the distinction between language and OS, and I expect these distinctions will begin to be blurred again as we address the computing needs of the Internet age.

9.1 Conclusions

My research makes the following contributions:

- An analysis of security architectures that might be applied to a type-safe language, such as Java.
- The design and implementation of stack inspection (at Netscape).
- The design of security-passing style, and a proof of its equivalence to stack inspection.
- An implementation of security-passing style and an analysis of its performance characteristics, both measured and ideal.
- A number of optimizations for security-passing style, both static optimizations based on a class hierarchy analysis, and dynamic optimizations based on caching of previous results.

Combining these, we now have an efficient, powerful, and theoretically sound system for access controls in mobile code systems. Security-passing style addresses concerns that mobile code systems must inherently be inefficient or must rely on traditional operating systems mechanisms. In fact, because security-passing style can work seamlessly across networks, it can go farther than traditional operating systems security mechanisms in allowing for detailed and principled access controls. As mobile code is increasingly deployed, whether in the form of active networks, shared virtual realities, or programmed stock trading, the importance of a sound security architecture, such as the one proposed in this dissertation, increases likewise.

Bibliography

[ABLP93] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[AGS83] Stanley R. Ames, Jr., Morrie Gasser, and Roger G. Schell. Security kernel design and implementation: An introduction. *Computer*, pages 14–22, July 1983. Reprinted in *Tutorial: Computer and Network Security*, M. D. Abrams and H. J. Podell, editors, IEEE Computer Society Press, 1987, pp. 142–157.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. 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.

[And72] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, Massachusetts 01730 USA, October 1972. Volume 2, pages 58–69.

[BAN90] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[BG98] Dirk Balfanz and Li Gong. Experience with secure multi-processing in Java. In *Proceedings of ICDCS'98*, Amsterdam, The Netherlands, May 1998.

[BL73] D. Elliot Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547-I, MITRE Corporation, March 1973.

[BL76] D. Elliot Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997 Rev. 1, MITRE Corporation, March 1976.

[BN89] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, California, May 1989.

[BNOW95] Andrew D. Birrell, Greg Nelson, Susan Owicki, and Edward P. Wobber. Network objects. *Software: Practice and Experience*, S4(25):87–130, December 1995.

[Bor94] Nathaniel S. Borenstein. 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*, Barcelona, Spain, 1994.

[BSP$^+$95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain, Colorado, December 1995.

[BSS$^+$95] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, California, 1995.

[BTR80] V. Berstis, C. D. Truxal, and J. G. Ranweiler. System / 38 addressing and authorization. In *IBM System/38 Technical Developments*, pages 51–54. IBM, 2nd edition, July 1980. IBM Document Number: G580-0237-1.

[BTS$^+$98] Godmar Back, Patrick Tullman, Leigh Stoller, Wilson C. Hseih, and Jay Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, August 1998. `http://www.cs.utah.edu/projects/flux/papers.html`.

[Cas95] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[CCK98] Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the 1998 Usenix Annual*

Technical Symposium, pages 167–178, New Orleans, Louisiana, June 1998.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12(4):271–304, May 1994.

[CLLBH92] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 397–413, Vancouver, British Columbia, Canada, October 1992.

[Com97] Computing Sciences Research Center of Bell Labs, Murray Hill, New Jersey. *Inferno: la Commedia Interattiva*, 1997. `http://www.lucent-inferno.com/Pages/Developers/Documentation/White_Papers/commedia.html`.

[Cou95] Antony Courtney. Phantom: An interpreted language for distributed programming. In *Usenix Conference on Object-Oriented Technologies*, June 1995.

[Cul98] Cult of the Dead Cow. *Back Orifice*, August 1998. `http://www.cultdeadcow.com`.

[Cv98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, Vancouver, British Columbia, October 1998.

[CW87] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, California, May 1987.

[DA97] Tim Dierks and Christopher Allen. *The TLS Protocol, Version 1.0*. Internet Engineering Task Force, November 1997. Internet draft, `ftp://ietf.org/internet-drafts/draft-ietf-tls-protocol-05.txt`.

[DBWL95] Robert H. Deng, Shailendra K. Bhonsle, Weiguo Wang, and Aurel A. Lazar. Integrating security in CORBA based object architectures. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 50–61, Oakland, California, May 1995.

[DE97a] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, Paris, France, January 1997.

[DE97b] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, Jyväskylä, Finland, June 1997.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, California, May 1996.

[DFWB97] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browsers and beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 241–269. ACM Press, New York, New York, October 1997.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95)*, Århus, Denmark, August 1995.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236. ACM Press, January 1998.

[Dig97] Digitivity Corp. *The Digitivity Cage*, 1997. `http://www.digitivity.com`.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically typed object-oriented programs. In *OOPSLA '96: 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, pages 292–305. ACM Press, 1996.

[DWE98] Sophia Drossoppoulou, David Wragg, and Susan Eisenbach. What *is* Java binary compatibility? In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 341–358, Vancouver, British Columbia, October 1998.

[Ele96] Electric Communities, Sunnyvale, California. *The Electric Communities Trust Manager and Its Use to Secure Java*, September 1996. `http://www.communities.com/company/papers/trust/`.

[ER89] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 326–343, Oakland, California, May 1989.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–411, July 1974.

[Far93] Dan Farmer. Cops (computer oracle and password system), May 1993. `http://www.trouble.org/cops/`.

[Fer95] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of ACM SIGPLAN '95 Conference on Programming Langauge Design and Implementation*, volume 30, pages 103–115, 1995.

[FHL$^+$96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 137–151, Seattle, Washington, October 1996.

[FK97] Michael Franz and Thomas Kistler. A tree-based alternative to Java byte-codes. In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*, 1997. Also appears as Technical Report

96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol: Version 3.0*. Internet Engineering Task Force, March 1996. Internet draft, `ftp://ietf.cnri.reston.va.us/internet-drafts/draft-freier-ssl-version3-01.txt`.

[Fla97] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, California, 2nd edition, January 1997.

[FV93] Dan Farmer and Wietse Venema. Improving the security of your site by breaking into it. `http://www.trouble.org/security/admin-guide-to-cracking.html`, December 1993.

[Gen95] General Magic, Inc., Mountain View, California. *The Telescript Language Reference*, October 1995. `http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html`.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[Gol96] Theodore Goldstein. *The Gateway Security Model in the Java Electronic Commerce Framework*. JavaSoft, Sunnyvale, California, November 1996. `http://www.javasoft.com/products/commerce/jecf_gateway.ps`.

[Gon89] Li Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, Oakland, California, May 1989.

[Gri98] David Griswold. *The Java HotSpot Virtual Machine Architecture.* Sun Microsystems, Palo Alto, CA, March 1998. `http://java.sun.com/products/hotspot/whitepaper.html`.

[GS98] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *The Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998. Internet Society.

[GW95] Joshua D. Guttman and Mitchell Wand. VLISP: A verified implementation of Scheme. *LISP and Symbolic Computation*, 8(1/2):5–32, March 1995.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Sixth USENIX Security Symposium Proceedings*, pages 1–12, San Jose, California, July 1996.

[Har85] Norman Hardy. KeyKOS architecture. *ACM Operating Systems Review*, 19(4):8–25, October 1985.

[Har88] Norman Hardy. The confused deputy. *ACM Operating Systems Review*, 22(4):36–38, October 1988. `http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html`.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998. USENIX.

[Hen82] John L. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(4):323–344, July 1982.

[How97] John D. Howard. *An Analysis of Security Incidents On The Internet*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1997. `http://www.cert.org/research/JHThesis/`.

[Hu95] Wei Hu. *DCE Security Programming*. O'Reilly & Associates, Inc., Sebastopol, California, July 1995.

[HYHD95] Richard C. Ho, C. Han Yang, Mark Horowitz, and David L. Dill. Architecture validation for processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 404–413. Association for Computing Machinery, June 1995. Santa Margherita Ligure, Italy, 22-24 June 1995.

[Int98] Internet Security Systems, Atlanta, Georgia. *Internet Scanner User Guide, version 5.2*, 1998. `http://download.iss.net/eval/NT_Scanner.pdf`.

[KH84] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, California, May 1984.

[KL87] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, SE-13(2):202–207, February 1987.

[KN93] John T. Kohl and Clifford Neuman. The Kerberos network authentication service (V5). Technical Report RFC-1510, Internet Engineering Task Force, September 1993. `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1510.txt`.

[Kno97] Kit Knox. Rootshell, 1997. `http://www.rootshell.com`.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[LaD96] Mark LaDue. Hostile applets home page, 1996. `http://www.rstcorp.com/hostile-applets/`.

[Lam71] Butler W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in *Operating Systems Review*, 8(1):18–24, January 1974.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[Lan81] Carl E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44, Vancouver, British Columbia, October 1998.

[LDOW98] Jacob Y. Levy, Laurent Demailly, John K. Ousterhout, and Brent B. Welch. The Safe-Tcl security model. In *USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998. USENIX.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[MAE$^+$62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Tompthy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2nd edition, 1962.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly & Associates, Inc., Sebastopol, California, March 1997.

[MF97] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, New York, New York, 1997.

[Mic96] Microsoft Corporation, Redmond, Washington. *Proposal for Authenticating Code Via the Internet*, April 1996. `http://www.microsoft.com/security/tech/authcode/authcode-f.htm`.

[Mic97a] Microsoft Corporation, Redmond, Washington. *Microsoft Security Management Architecture White Paper*, May 1997. `http://www.microsoft.com/ie/security/ie4security.htm`.

[Mic97b] Microsoft Corporation, Redmond, Washington. *Trust-Based Security for Java*, April 1997. `http://www.microsoft.com/java/security/jsecwp.htm`.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–153, Oakland, California, May 1997.

[MP96] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 153–167, Seattle, Washington, October 1996.

[MRR97] David M. Martin Jr., Sivaramakrishnan Rajagopalan, and Aviel D. Rubin. Blocking Java applets at the firewall. In *Internet Society Symposium on Network and Distributed System Security (NDSS '97)*, San Diego, California, 1997.

[MRR98] Dahlia Malkhi, Michael Reiter, and Avi Rubin. Secure execution of Java applets using a remote playground. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, California, May 1998.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[Mue96] Marianne Mueller. Personal communication, January 1996.

[Nat85] National Computer Security Center, Fort Meade, Maryland. *Department of Defense Trusted Computer System Evaluation Criteria (The Orange Book)*, December 1985.

[Nat98] NaturalBridge, LLC. *BulletTrain Java Compiler*, 1998. `http://www.naturalbridge.com`.

[NBF+80] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, 2nd Ed., SRI International, May 1980.

[Net96] Netscape Communications Corporation, Mountain View, California. *The JAR Format*, 1996. `http://developer.netscape.com/library/documentation/signedobj/jarfile/`.

[Net97] Netscape Communications Corporation, Mountain View, California. *Introduction to the Capabilities Classes*, August 1997. `http://developer.netscape.com/library/documentation/signedobj/capabilities/index.html`.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without runtime checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Seattle, Washington, October 1996.

[O'B96] David O'Brien. Recognizing and recovering from Rootkit attacks. *Sys Admin: The Journal for Unix Systems Administrators*, 5(11), November 1996. `http://perry.prnaccess.com/sys_admin_1996/sacdrom/htmfiles/0511/x0011.htm`.

[Obj96] Object Management Group. *Common Secure Interoperability*, July 1996. OMG Document Number: orbos/96-06-20.

[Org72] Elliot I. Organick. *The Multics System: An Examination of its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[PB96] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, Washington, October 1996.

[PD96] Seungjoon Park and David L. Dill. Verification of flash cache coherence protocol by aggregation of distributed actions. In *Symposium on Parallel Algorithms and Architectures*, pages 288–296, June 1996.

[PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, London, England, July 1990.

[Ros96a] Jim Roskind. *Evolving the Security Model For Java From Navigator 2.x to Navigator 3.x*. Netscape Communications Corporation, Mountain View, California, August 1996. `http://developer.netscape.com/library/technote/security/sectn1.html`.

[Ros96b] Jim Roskind. Java and security. In *Netscape Internet Developer Conference*, Mountain View, California, March 1996. `http://home.netscape.com/misc/developer/conference/`.

[Rus81] John M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 12–21, December 1981.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 149–160. ACM, January 1998.

[Sal74] Jerome H. Saltzer. Information protection and the control of sharing in the Multics system. *Communications of the ACM*, 17(7):388–402, July 1974.

[SCFY94] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control: A multi-dimensional view. In *Proceedings of the 10th Computer Security Applications Conference*, pages 54–62, Orlando, Florida, December 1994. IEEE Computer Society Press.

[Sch96] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, New York, New York, 2nd edition, 1996.

[SESS94] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. An introduction to the architecture of the VINO kernel. Technical Report 34-94, Harvard University, Dept. of Computer Science, 1994.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–227, Seattle, Washington, October 1996.

[SGB$^+$98] Emin Gün Sirer, Robert Grimm, Brian N. Bershad, Arthur J. Gregory, and Sean McDirmid. Distributed virtual machines: A system architecture for network computing. In *Eighth ACM SIGOPS European Workshop*, September 1998.

[Sib96] W. Olin Sibert. Malicious data and computer security. In *19th National Information Systems Security Conference*, Baltimore, Maryland, October 1996.

[Sie96] Jon Siegel, editor. *CORBA Fundamentals and Programming*. John Wiley and Sons, New York, New York, 1996.

[Sir97] Emin Gün Sirer. Kimera: A Java system architecture. `http://kimera.cs.washington.edu`, 1997.

[SS72] M. D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[Ste78] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, Massachusetts, 1978.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Sun95] Sun Microsystems, Sunnyvale, California. *Frequently Asked Questions - Applet Security*, 1995. `http://java.sun.com/sfaq/`.

[Sun96] Sun Microsystems, Mountain View, California. *JAR Documentation*, 1996. `http://java.sun.com/products/jdk/1.1/docs/guide/jar/`.

[TA90] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 1–12, New York, June 1990. ACM Press.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[TL98] Patrick Tullman and Jay Lepreau. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop*, September 1998.

[TMvR86] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*, pages 558–563, Cambridge, Massachusetts, May 1986.

[Tra98] Transvirtual Technologies, Inc., Berkeley, CA. *Kaffe OpenVM*, 1998. `http://www.transvirtual.com`.

[vABW96] Leendert van Doorn, Martín Abadi, Michael Burrows, and Edward Wobber. Secure network objects. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.

[WABL94] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 116–128, Saint-Malo, France, October 1997.

[WCC$^+$74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, California, May 1998.

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 2nd edition, 1983.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, North Carolina, 1993.

[WRF96] Dan S. Wallach, Jim A. Roskind, and Edward W. Felten. Flexible, extensible Java security using digital signatures. In *DIMACS Workshop on Network Threats*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, December 1996.

[Zad98] Kenneth Zadeck, November 1998. Personal communication.