Fine-Grained Control of Java Applets Using a Simple Constraint Language

by

Nimisha V. Mehta

Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the
Requirements for the Degrees of

Bachelor of Science in Computer Science and
Engineering and Master of Engineering in Electrical
Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1997

MIT-LCS-TR-713

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Fine-Grained Control of Java Applets

Using a Simple Constraint Language

by

Nimisha V. Mehta

Submitted to the

Department of Electrical Engineering and Computer Science

May 1997

In Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering and

Master of Engineering in Electrical Engineering and Computer Science

Abstract

The use of the internet has increased extensively with a growing number of inexperienced users surfing the Web. Lurking in Web pages, Java applets are automatically executed on users' machines. As a result, popular Web browsers are understandably conservative on what they allow Java applets to do. However, this places a heavy restriction on applets which drastically limits their capabilities. Therefore, we have developed a constraint language in which naive users can specify their fine-grained control over applets without needing to know the the intricacies of applet security. We have written an implementation of the Java Security Manager built into Sun's AppletViewer to demonstrate the feasability of this approach, addressing the many security issues that arise when opening the operating system to the public domain. This involves maintaining a log of applets' past accesses to determine the allowability of their future accesses, along with an account of which applets 'own' which files.

Thesis Supervisor: Dr. Karen R. Sollins
Title: Research Scientist, Laboratory for Computer Science

Acknowledgments

I would like to give special thanks to my thesis supervisor, Dr. Karen Sollins, for helping me find a topic that I truly enjoy. Her vision has fruited the basis of this thesis work. She has helped me focus in on the important issues and has always reminded me of the big picture. If it were not for her, this paper would be filled with split infinitives! She has not only advised me on this thesis, but, on everything from finding a job to creating a study group matchup program for Eta Kappa Nu. Thank you, Karen. :-)

Lewis Girod has also been an invaluable help in helping me formulate ideas through his innovation and experience. Special thanks must go to Edward Slottow who, always being a step ahead of me in completing our theses, has let me borrow his Framemaker templates and formats which have tremendously helped me in finishing up the writeup on time! Members of the Advanced Network Architecture group including Andrew Parker, have made the project even more fun while sharing cookies and laughs.

A hearty thanks to Anand Asthagiri for proof-reading this paper from start to finish and for providing invaluable comments and criticisms. He has taught me that work should be embraced and not discarded on Friday nights. Thanks to Jay Ongg who ever since sophomore year has been with me through the numerous foolish things I have done. His enthusiasm for CS has energized mine. His distractions in making me listen to his music compositions have kept me smiling throughout. Thanks to Nilesh Gandhi for his endless jokes, and to Steve Lin for reminding me to eat dinner. Thanks to the river folks for the fun times during my breaks from working.

Thanks to my faculty advisor, Prof. Peter Szolovits, who made sure that I could work on my Masters on campus this fifth year.

And an enormous thanks to my parents, Vasantlal and Jayana Mehta, who have helped me come far from the shy little girl I was. They have taught me to live life embracing religion, education, and love. Thanks to my sister, Purvi, and my brother, Krupesh for being patient with me through all my endevours.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. DABT63-94-C-0072.

Table of Contents

List Of Figures

Chapter 1
Introduction

Security has become an important issue in today's culture. This can easily be seen by how much money the United States government budgets to their military program, or by the number of locks people place on their main entrances to their homes. Such measures are also required when securing one's computer system to protect against theft, eavesdropping, and vandalism. With the advent of the Internet, more and more computers become vulnerable because of their wired connection to the external network - a computer attacker's heaven. Computer users must now face the challenge of keeping their doors open while discarding unwanted guests.

1.1 The World-Wide Web

The World-Wide Web (WWW) is a collection of interconnected, linked information which was first developed in 1989 by Tim Berners-Lee and CERN.

It consists of distributed Web pages or documents which contain hypertext links to other documents on the Internet. A network is thus created through arbitrary connections from one page to another. These documents are downloaded from Web servers onto the user's system (Web client) and viewed using applications known as Web browsers. Browsers are capable of accessing these documents using various protocols such as HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol), NNTP (Internet News Protocol), Telnet, and gopher.

1.2 Mobile Code - Java Applets

Along with the Internet, a new fad, mobile coding, now makes a computer system even more assailable than before. Mobile coding refers to the transmission of executable content over the network which is then run on the client's machine. Its popularity emerged when the World Wide Web began to attract more and more users. Mobile coding became a preferred alternative to server-side execution. It is used to run remote computations when the server is down, or to overcome latency and bandwidth issues by transporting the code over the network and then executing the code locally, without requiring any further network traffic. Figure 1 illustrates this transaction.

Sun Microsystems created a mobile code technology named Java in 1993. Java's platform independence and safety features resulting from its virtual machine environment (as is further described in Chapter 3) has made it widely adopted and supported by the computer industry. Being the prevalent

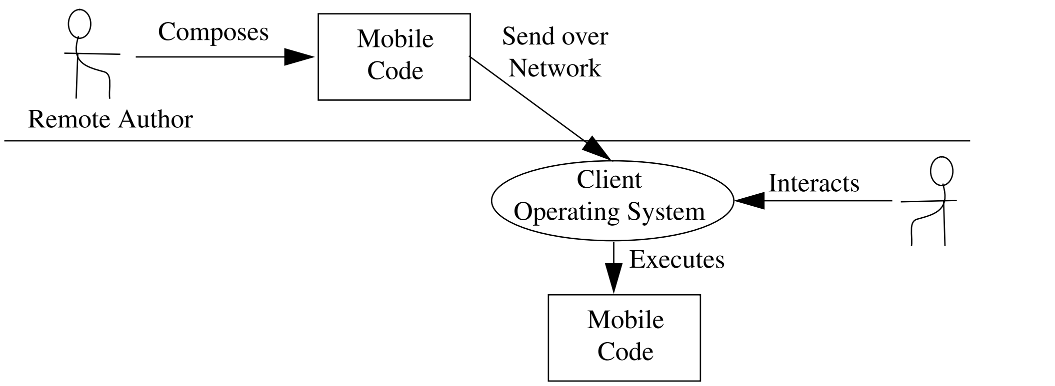

Figure 1. Mobile code architecture [14]

mobile code that have been lurking in Web pages, Java applets will be the focus of this paper. Java applets are Java applications that are embedded within Web documents. [23] However, be aware that there are also other mobile code in Web content including Sun's Tcl/Tk, Microsoft's ActiveX controls, Plug ins, application macros, etc.

1.2.1 Security Issues with Mobile Code

Previously, executables could only be downloaded using protocols such as *FTP*, where the user was required to explicitly specify the name of the file to be downloaded. The user was aware of all the files (including executables) that were retrieved from the network. In order to run an executable, the user had to run the code explicitly. However, with mobile code, the executable is not only implicitly downloaded by the Web browser, but also spontaneously executed thereafter on the client machine using the user's access rights. As opposed to CGI-scripts which are executed on the host Web server, mobile code uses the resources on the client machine to execute its program.

Furthermore, the World Wide Web eases an attacker's troubles since it automates the process of code distribution. Previously, a malevolent hacker could distribute malicious code only by covertly placing the executable on the client's storage system, falsely publicizing it on an FTP site, or forwarding it on a floppy disk. Now, it can be inherently and automatically distributed through Web documents.

Although the automocity of downloading, executing, and distributing of the code makes Web surfing user-friendly, the user is now blind to the code being run on his/her own machine. For example, an off-the-shelf Web browser accessing a web page may download an applet that is causing malicious side-effects on the client's machine. It may be covertly reading or writing private files on the machine, or sending derogatory mail to the user's boss. This lack of awareness of foreign code, the easy accessibility to the client system, and the automation of code distribution make mobile coding on the WWW unsafe. It is difficult to maintain the following three tenents of computer security since the client system is now vulnerable to the intrusive mobile code:

1. **Privacy** - Ensures the protection of sensitive information located on one's system. One's private information should not be read or accessed by others. There is a need to thwart intruders and eavesdroppers from confidential data.
   i.e. Mobile code should not publicize the contents of private files.

2. **Data Integrity** - Ensures the preservation of the contents of the data stored on one's system. One expects to find information intact without modification by others. There is a need to thwart vandals from modifying the data.
   i.e. Mobile code should not modify protected data.

3. **System Reliability** - Ensures the predictability of one's system so it functions as expected. One expects the system to respond to the same input in a consistent manner. There is a need to thwart intruders from causing any damage to the system. *i.e. Mobile code should not selfishly use all the CPU cycles.*

1.2.2 Naivete of New Users

The widespread use of mobile code among new and inexperienced users requires even more caution and protection. For instance, the number of Web sites has been increasing from 130 in June 1993 to 650,000 in January 1997 with a doubling period of less than 6 months. [12] The number of Internet users in the United States alone had reached 5.8 million by November 1995 [24] and is now over 40 million. [3] As the Internet welcomes the ever increasing number of users, more and more of these users are naive of the intricacies of the Web. These inexperienced users are understandably ignorant of the existence of mobile code within Web documents and the consequences of executing them on their systems. An attacker can easily lure ordinary users to a malicious Web site by advertising appealing merchandise or information. While, unbeknownst to the users, their machines may be executing malicious code when they download the document. Consequently, for global protection, the mechanism used to secure computer systems from malicious applets should be easily understood by ordinary users.

Surprisingly, Web authors themselves are not knowledgeable about the security aspects of Java applet programming. In a survey conducted from October 1996 through November 1996 of 2,664 Web authors, 45.7% said they did not know anything about Java security. "Authors are not becoming more educated about Java's security even though more are planning to use it." At that time 24.4% of the authors had programmed in Java, since they regarded its platform independence as a major advantage. [17]

1.3 Current Web Browser Limitations

To guard against malicious applets, Java-enabled Web browsers have averted the security problems by simply placing harsh restrictions on all applets. The current most popular Web browser, Netscape's Navigator disallows all access to the local file system and restricts network access only to the originating host. It further prohibits applets from executing system commands, accessing system properties, printing files, or starting other programs on the client. [15] The only system accesses given to applets are the client's CPU to execute the code and the capability to create graphical windows to provide graphical user interfaces. Even the window itself is labeled with a warning to denote that it is owned by an applet. Figure 2 summarizes the restrictions on Java applets.

These restrictions provide safety against malicious applets so they do not compromise the privacy, reliability, and integrity of the client system. For example, when an applet is downloaded, the user is not aware of what it really does. Although it is probably benign, without security restrictions, it has the potential to erase or alter data, infect the system with a virus, crash the system, or damage it in another way. However, in using cautionary mea-

Reading Local Files	Not allowed.
Writing Local Files	Not allowed.
Deleting Local Files	Not allowed.
Exiting the Program	Not allowed.
Executing System Commands	Not allowed.
Dynamically linking to other libraries	Not allowed.
Listening for socket connections	Not allowed.
Accepting connections from hosts	Not allowed.
Connecting to hosts	Can only connect to the originating host.
Printing Files	Not allowed.
Accessing Threads	Can only access threads that are owned by applets.
Accessing System Properties	Can access the following properties: *java.version, java.vendor, java.vendor.url, java.class.version, os.name, os.arch, os.version, file.separator, path.separator, line.separator* But, cannot access the following properties: *java.home, java.class.path, user.name, user.home, user.dir*
Accessing other packages	Cannot access *sun* and *netscape* packages. Cannot define *sun*, *netscape*, or *java* packages.
Creating a Top-level window	Can create a top-level window, but includes a visual warning label.
Accessing the Clipboard	Not allowed.

Figure 2. Java Applet Restrictions as set by Netscape Navigator 3.x

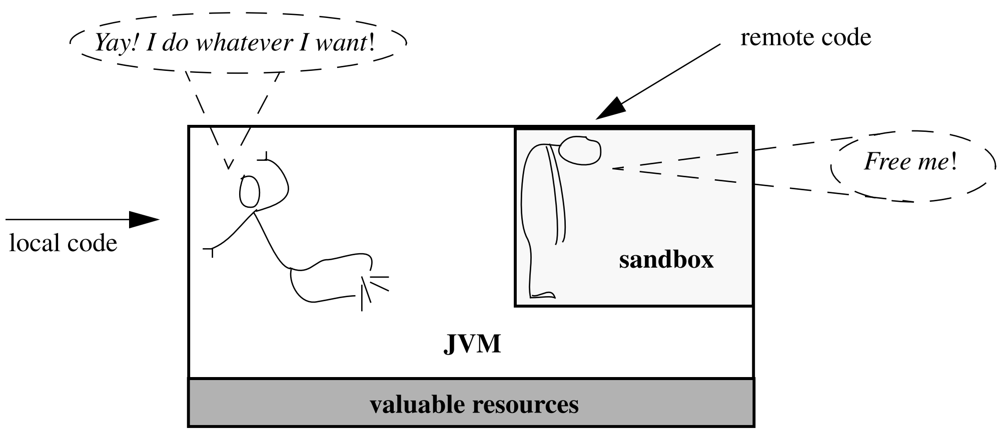

Figure 3. Sandbox - Remote code is Untrusted; Local code is Trusted.

"*Java provides the foundation for a secure environment in which untrusted code can be quarantined, managed, and safely executed. However, unless you are content with keeping that code in a little black box and running it just for its own benefit, you will have to grant it access to at least some system resources so that it can be useful. Every kind of access carries with it certain risks and benefits. The advantages of granting an untrusted applet access to your window system, for example, are that it can display information and let you interact in a useful way. The associated risks are that the applet may instead display something worthless, annoying, or offensive. Since most people can accept that level of risk, graphical applets and the World Wide Web in general are possible.*" [22]

sures against malicious applets, the Web browsers place rigorous limitations on all applets. Figure 3 depicts the confinement placed on applets.

This prevents applets from doing more useful services for the user such as maintaining the user's schedule book, creating an automatic birthday card delivery system, and updating the Web links in the bookmarks file. These beneficial services require access to the user's local file system as a storage resource, and access to the network for external information and communication. In fact, the applet's state can instead be stored on the parent host, and external network connections can also be achieved via the parent host. However, this inefficiently introduces extra latency and bandwidth costs which can be avoided if the applet were allowed to store information on the local file system and were permitted to connect to the network directly. To solve this, we need to find a way to loosen our leashes on applets while still ensuring the security of the client system.

1.4 Summary

The wide-scale popularity of Java applets, along with Web browsers' capability of automatically downloading, executing, and distributing them, have forced current Web browsers to excessively restrict their actions. In trying to devise a system which would loosen the leash on the applets, the following criteria should be maintained:

1. The system should not compromise the **security** of the client's computer in any way.

2. The system should be **straightforward** and easily understood by the wide distribution of naive users.

3. The system should be **configurable** with fine-grained control to accommodate the needs of the many users.

4. The system should be **flexible** to adapt to possible new uses and capabilities of Java applets.

Only those systems that are able to conform to the above constraints while giving applets a more lax policy can successfully solve the problem. This paper will describe the design and implementation of such a system that allows users to specify their policies over applets by using a simple constraint language. Permissions will be determined not only by the identity of the applet, but also by the applet's past accesses. In this paper, we will focus our attention to the UNIX operating system.

1.5 Preview

I have introduced the motivation for our work in this first chapter. Chapter 2 will briefly summarize and evaluate other approaches to addressing this problem. Chapter 3 will then provide background information before proceeding to the details of the proposed system. A design overview of the system will be described in Chapters 4 and 5, followed by a detailed explanation of its implementation in Chapter 6. We will then conclude in Chapter 7 with suggestions for future work.

Chapter 2

Related Work

Other individuals and organizations have tried to tackle the problems with setting access permissions on mobile code. The common approach is to build upon digital signatures and certificates to identify trusted applets. Microsoft, Netscape, and Sun have used this technique to create their applet policies. The World Wide Web Consortium has envisioned an even greater trust model using rating systems. And finally, a researcher at Carnegie Mellon University has devised a system whereby the code itself carries proof of its innocence.

2.1 Digital Signatures

To allow end users to decide which Web content they can trust, some have introduced the notion of digital signatures to identify the origin of the document and/or endorsers of the document. The signature may be embedded in the document, attached with the document, or received by querying a third party . [32] It is placed there by the originator of the applet, and it can't be altered or duplicated. A signature does not reveal anything about the content or quality of the applet, just the identity of its signer. It provides a way to authenticate the document so the user can give appropriate authorization. With this system, applets coming from a trusted source are granted higher privileges, while untrusted applets face the tightest restrictions.

The current digital signature techniques use RSA, a public key cryptography standard, where there are two keys, public and private, that are mathematically related to each other. A signature for a document is then a string of bits derived from the document itself and the private key of the signing party. The signer's public key is then used to verify the signature of the document. Note that since the signature depends on the document, it cannot be used to validate a different document. This protocol is illustrated in Figure 4. The current systems also rely on Key Certification Authorities (CAs), such as Verisign, that are responsible for issuing and/or certifying these public keys. [32]

2.2 Binary Trust Model - Microsoft's IE

The digital signature technique is currently used in Microsoft's Authenticode system in Microsoft Internet Explorer 3.0. The Authenticode technology allows a user to maintain his/her "Safety Level" at "High" to allow only digitally signed applets to be downloaded. Using the digital signature, the applet's code is verified to ensure that has not been tampered with. The user

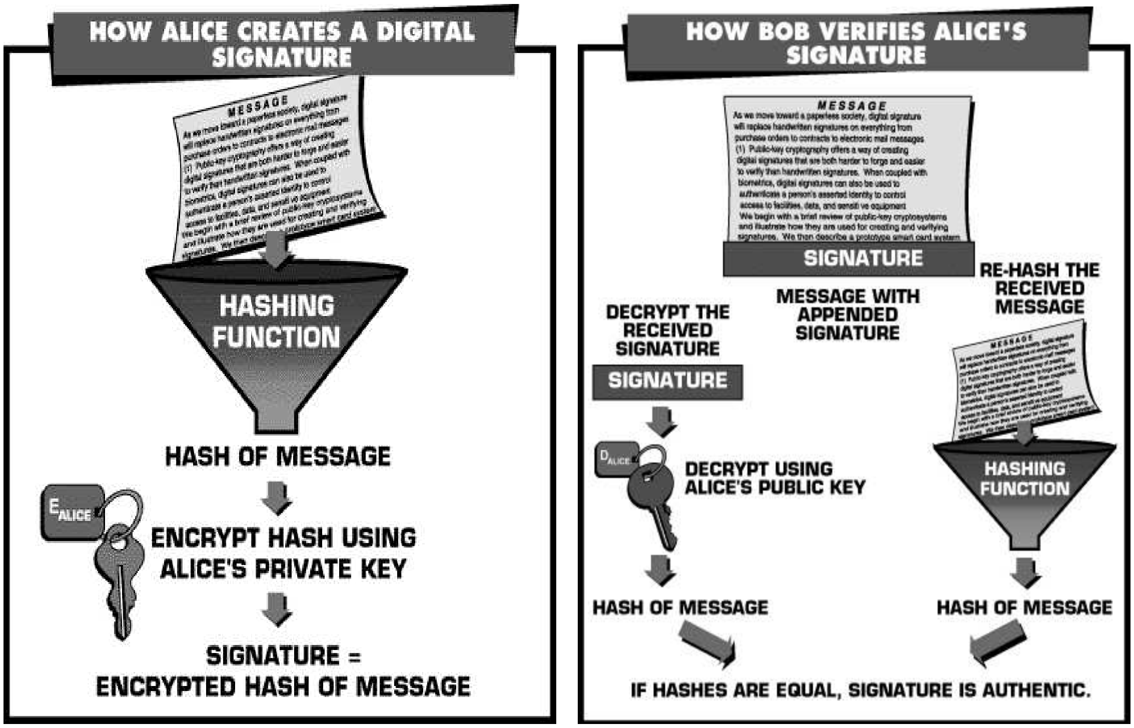

Figure 4. Creating and Verifying Digital Signatures [6]

then makes a decision as to whether he/she wishes to run the executable, taking into regard the name of the certificate authority and the author/endorser of the code. Unsigned applets can also be downloaded if the “Safety Level” is set to “Medium”. In both cases, however, the system interrupts the user by requesting permission each time prior to executing the applet code. [19]

However, although the digital signature initiative is a good mechanism for identifying and authenticating applets, this binary trusted/not-trusted model is not sufficient for setting applet policies. This provides an all-or-nothing approach, where a trusted applet gains all access to the system, while a non-trusted applet gains none.

2.3 Capabilities - Netscape’s Communicator

Netscape’s newest Web application, Communicator, includes a protocol called Netscape Object Signing which allows Java applets written with capabilities to request the ability to operate outside of their sandbox. Capabilities are requests to use the user's system resources (for example: writing to disk, reading from disk, establishing a remote connection, etc.). [21]

The identification of applets is similar to the Authenticode method in which the applets are digitally signed. However, the capabilities-based approach provides an extra level beyond the binary trust model. When a signed object is downloaded, the user receives information indicating the signer of the applet and the capabilities that the applet requests. Based on that identification, the user decides whether to give the applet the requested access to his/her computer. Since the user is pre-informed of all the resources to which the applet will need access, he/she can allow access to only those resources and need not blindly allow access to all resources as would be done with the Authenticode system. Unsigned applets, however, would still be entirely limited to their sandboxes. [21]

Even though this provides a some-or-nothing approach, it still requires users to give their permission each time by interrupting them as they surf the Web. Although, users may initially pay particular attention to the dialog boxes, after a while, users discard these messages haphazardly since they will routinely appear. Also, the users will need to reevaluate their policy each time to see if the applet conforms to it. Instead, if the system set the policy at one-time and then abided by it thereafter, a more user-friendly system would be achieved.

2.4 Configurable Applet Policy - Sun's HotJava

The HotJava Browser 1.0 allows the user to configure a one-time applet policy allowing an applet more or less access to local system resources based on the applet's digital signature. Users can tune security by controlling which applets can be downloaded, which system files can be read or written, which network hosts can be accessed, and which system permissions can be granted. [27, 16]

Restrictions are set according to a particular certificate, a particular Web site, or a grouping of sites and certificates. A check-list is provided where the user can specify which system accesses (opening windows without warning labels, accessing HotJava properties, accessing the clipboard, printing files, and launching applications) are granted or denied for applets from each group. The user can list which files and directories the applets can read from and write to, and similarly, which network hosts and ports the group can access. This graphical dialog is shown in Figure 5. [27]

This provides a much more configurable mechanism for specifying the user's applet policy. The policy is set once and configured through the use of check-boxes and list-boxes. However, the structure of the policy is limited to what is provided in the interface, namely the check-list of resources and the list of names. A more flexible, customizable tool is needed.

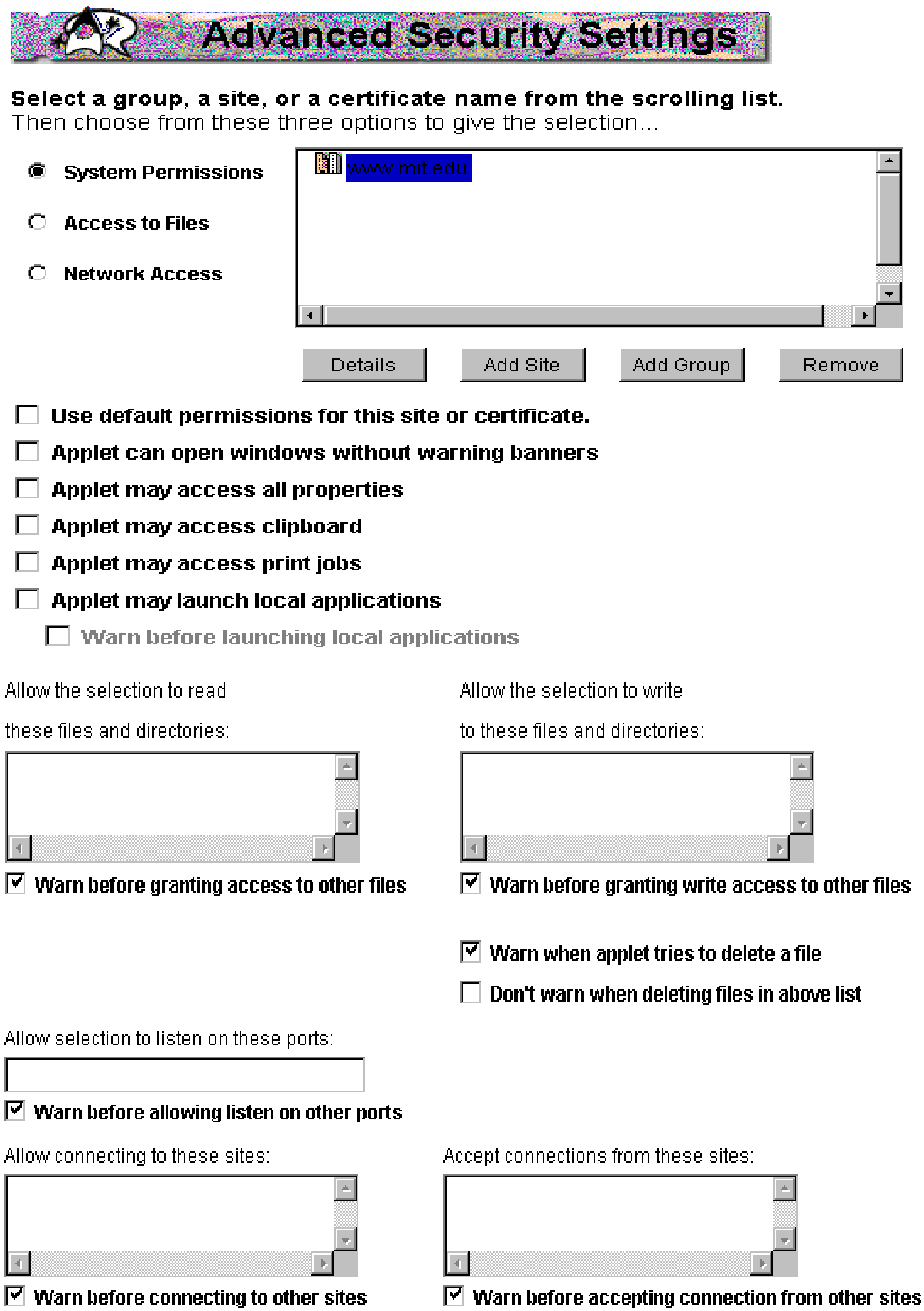

Figure 5. HotJava's Applet Security Preferences Page

2.5 X believes Y about Z - PICS Trust Model

Another protocol built upon the digital signature scheme is the Platform for Internet Content Selection (PICS) model developed by the World Wide Web Consortium. PICS is described as "an infrastructure for associating labels (metadata) with Internet content". [33] The labels specify the types of content included in the documents. These labels may be created and distributed by the authors of the documents themselves and/or by multiple, independent labelling services which would rate others' documents using their own rating system. When an end-user asks to see a particular URL, a PICS-compliable Web browser fetches the document and also makes an inquiry to the label bureau to ask for labels that describe that URL. Depending on what the labels say and where the labels originate, a filter configured by the user may block access to that URL. [33]

This system evolves into a community built upon a trust model often termed as the Web of Trust. Here is what Microsoft has to say about the trust model:

> "The only way to provide users with both the most positive computing experience and maximum security is to adopt a model based on trust. A trust model identifies the certified provider of a program and then allows users to decide for themselves whether to trust that provider. This model enables developers to create the most innovative applications, and enables users to realize the full potential of their computers while still maintaining an appropriate level of security." - *http://www.microsoft.com/security/*

However, although this trust model is adequate for content that cannot yet be judged by a computer, such as graphics or textual compositions, it is not sufficient for active code such as Java applets. Applets can be regulated by the browser because of the safety of the Java language as will be discussed later in Chapter 3. Therefore, such a trust model is not necessary for granting access to Java applets. Would you allow an applet to access your file system just because your friend Joe 'thinks' it is alright?

2.6 Proof-Carrying Code - CMU

George C. Necula and Peter Lee from Carnegie Melon University (CMU) are attacking this problem using a different approach. They have introduced the notion of proof-carrying code (PCC), a mechanism by which the Web server supplies a "safety proof" attached with the code that attests to the code's adherence to a previously agreed safety policy. The client can then validate the proof to assure that the code abides to the safety policy. Figure 6 illustrates this protocol. [20]

This approach promises a desirable outcome since no trust is assumed, rather, the trust is earned by proving the code's benevolence. However, a more user-friendly system needs to be achieved so ordinary users can configure their own policies. A system where users can declare their access permissions and where the restrictions would be based on trustworthiness and/or authentication of the applet.

2.7 Java Protected Domains - Java's Future

Developers at JavaSoft are restructuring the Java Virtual Machine (JVM) to help develop fine-grained control policies. They have introduced the

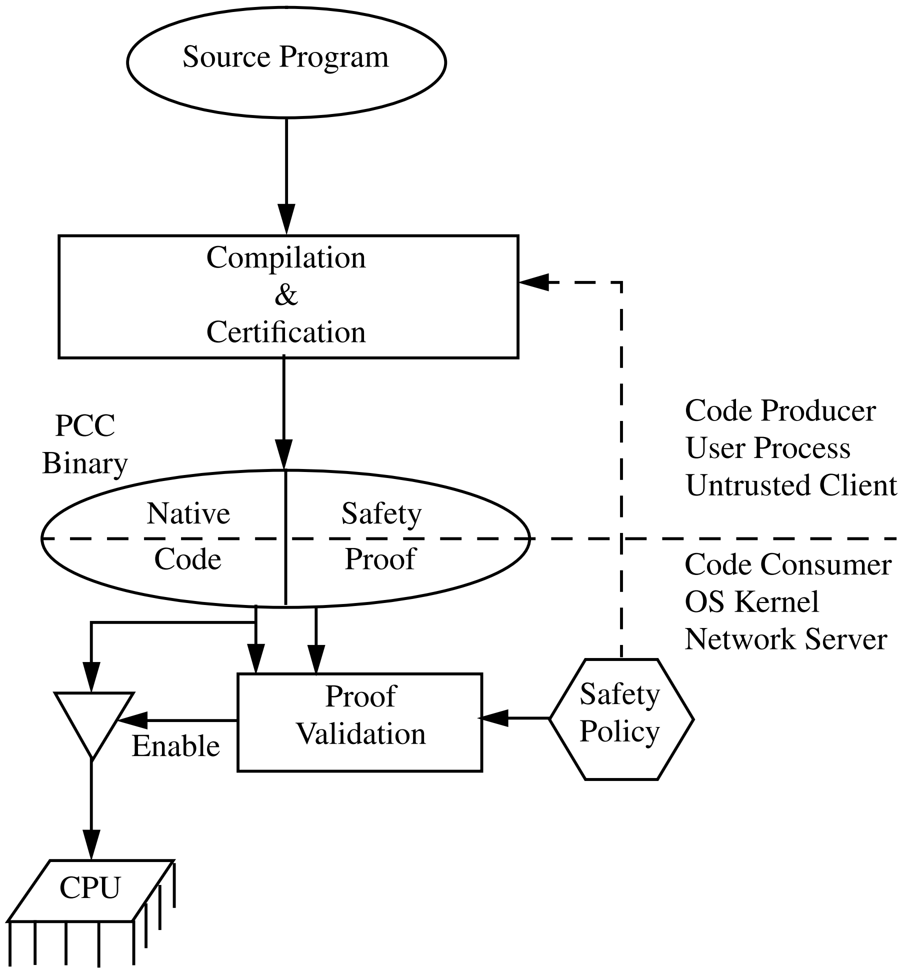

Figure 6. Proof-Carrying Code

idea of Java Protected Domains where the JVM is effectively partitioned into distinct protection components for the set of resources. Each component can impose restrictions on its internal resources. Underlying these domains, an inherent mechanism exists which ensures that applications do not falter from their security permissions during inter-domain interactions. [30]

This work will provide an inherent mechanism for easily setting permissions on resources and for securely protecting resources. When integrated with a mechanism for specifying policies and for deciding permissions, a secure system can perhaps be achieved.

2.8 Summary

Although the above approaches allow trusted applets more access to the client system, they either do not provide a less trust-requiring system (criteria 1), a more user-friendly interface (criteria 2), a finer granularity of control over applets (criteria 3), or a more flexible tool (criteria 4) as required in Section 1.4. Further, they do not keep track of information flow between an applet and a later execution of that applet or another applet as will be described in Section 3.3.3.

Chapter 3

Addressing Security

This chapter provides the background needed to further understand the security issues with Java applets running on the Unix operating system. We first describe the specifics of the UNIX system, paying particular attention to its security features. The safeness of the Java language and the design of the Java platform is then detailed. After that, the security concerns that arise when using Java and Java applets will be listed.

3.1 Unix Operating System

The Unix system was originally designed with emphasis on ease of use rather than on security. Its roots as a research platform in AT&T Bell Laboratories and in academia caused no great need for security until some time later when it extended to the commercial environment. [25,1, 10, 11] However, if used properly, it includes several security measures through its password security, file access control, and auditing capabilities. On the other hand, it has several user-friendly features that inadvertently increase the vulnerability of the system: the **set-UID** bit, file links, the *PATH* variable, the networking utilities, excessive resource usability, and user customization files.

3.1.1 Password Security

The password is the most vital part of UNIX' multi-user account security. Users have their own private passwords which the system uses for authentication. These passwords are encoded with a one-way encryption (Data Encryption Standard) and stored in the */etc/passwd* file preventing the need to keep the raw password in file storage. When a user logs in with his/her password, the password is encoded and then compared against the encrypted password in the user's entry within the */etc/passwd* file. [2, 1, 11]

Even though the passwords are encrypted, crackers can use a brute-force attack to discover users' original passwords if they manage to pirate a copy of the password file. Once the outsider cracks a user's password, he/she can easily log into the system with all the capabilities of that user. Furthermore, UNIX has a privileged account known as *root* which gives *"super user"* capabilities including reading from any file, writing to any file, and invoking privileged system calls. If the password to this *root* account is exposed, then <u>all</u> accounts are compromised.

In addition to the encrypted password, the */etc/passwd* file contains the user's login name, home directory, and login shell. The password file therefore needs to be accessed by many other programs (e.g. '*cd*') that need to know the location of the user's home directory. This being the case, it is kept world-readable for easy access by these programs. However, the password file's world-readability and its well-known location make it trivial for an intruder to access the file. Therefore, some administrators create a non-world-readable shadow password file, */etc/shadow*, that contains the users' actual passwords, while the */etc/passwd* file contains the other information. [1, 8]

3.1.2 File System Security

Each file in the UNIX file system has associated with it an *inode* which is part of a *file descriptor* that describes the properties of the file. These properties include file access permissions, linking information, and other file attributes. Directories are treated as special types of files. [18]

3.1.2.1 Access Control Lists

Each file or directory has three sets of permission bits associated with it together with a user identification number (*UID*) and a user-group identification number (*GID*). The three sets of bits are for the ***owner*** of the file, for the users in the ***group*** associated with the file, and for all other users (the "***world***" permissions). File access for a user is not hierarchical: if the file is owned by the user, the ***owner*** bits are tested, otherwise if the user is in the file's group, the ***group*** bits are tested, otherwise the ***world*** permissions are tested. [25, 11] This means that a file giving access to the ***world*** does not necessarily give access to its ***owner***. Each set contains three identical permission bits that control the following [5]:

read : If set, the file or directory may be read. In the case of a directory, read access allows a user to see the contents of a directory (the names of the files contained therein), but not to access them.

write: If set, the file or directory may be written (modified). In the case of a directory, write permission implies the ability to create, delete, and rename files.

execute **:** If set, the file or directory may be executed (searched). In the case of a directory, execute permission implies the ability to climb into that directory or to use it as a component of a path name.

Note that write access does not imply read access. In addition, underlying directory permissions can adversely affect the safety of seemingly protected files. For example, a *file F* that has its write bit not set can still be modified if the permissions set by its parent *directory D* allow write permission: *file F* can be deleted and re-created using *directory D*'s write permissions without having to obey *file F*'s write permissions. [11]

The access mode can be modified with the *chmod* system command; the owner of the file can be changed with the *chown* command; and the group associated with the file can be changed with the *chgrp* command. Only the owner of the file/directory (or the *super user*) can modify these properties. [5, 13]

3.1.2.2 UMASK Value

In conjunction with the standard access controls, UNIX provides a system feature called *umask* which allows the user to specify the default file and directory creation mode. The bits in the *umask* value are <u>masked</u> with the permission bits specified by the system upon creation of a file. So it actually defines the permission bits that are <u>not</u> to be given by default. [5, 1] This allows users to easily avoid the creation of world-writable and/or world-readable files. The *umask* value can be specified in the user's start-up configuration files as described in Section 3.1.4.4. [5] Users should be aware of this value and set it conservatively.

3.1.2.3 Set-UID & Set-GID Bits

Along with the nine bits specifying a file's read, write, and execute permissions, each file has two additional bits that are used only when the file is executed as a program: **set-UID** and **set-GID** bits. Each process generated by or for the user has associated with it an *effective UID* and a *real UID*, and an *effective GID* and a *real GID*. The user process' *effective UID* and *effective GID* are used to determine the access permissions. If the **set-UID** bit is set for the file, the *effective UID* for the process is changed to the *UID* associated with the file. Similarly, the *effective GID* of a process is changed to the *GID* associated with the file when that file's **set-GID** bit is set. The *real UID* and *real GID* of the process do not change when the file is executed. [25, 5, 11]

The idea behind **set-UID** and **set-GID** bits is that one may write a program which is executable by others and which maintains files accessible to others but only through that program. A classic example is a game-playing program which records the player's scores in a file. The score file needs to be modified and updated by the game via players' processes, but cannot be modified directly by the players since it would give them freedom to alter the scores. This can be accomplished by setting on the game program's **set-UID** bit. [25]

One must use the **set-UID** (and **set-GID**) bits very carefully in order to maintain the security of the system. For example, a writable **set-UID** file can have another program copied onto it preserving the **set-UID** bit. This allows any user to execute commands using another user's access permissions. Therefore, one must ensure that all **set-UID** files are writable by no one other than their owners. Another security concern is when a **set-UID** file is owned by *root*. For those files, any user who executes that file is temporarily empowered with *super user* capabilities. Care must be taken to guarantee that these *root* owned **set-UID** files are bug-free and trusted. [25]

3.1.2.4 File Links

File links allow a user to give a single file (a single *inode*) multiple names. Links can be *hard* or *soft* (*symbolic*). *Hard links* create links directly to an *inode*, while *soft links* create new *inodes* which name the files to which they are linked. They are useful when one needs to have the same file located at multiple places. When the file or the link is modified, the other file is apparently modified. [18]

File links introduce a security concern since they may accidently lead access to a directory that was intended to be protected. For example, if one's */ftp/pub* directory had a link to a private directory for some reason, an outsider can travel to that private directory by following that link. Care must be taken to prevent such insecure links.

3.1.3 Audit Mechanism

The UNIX system audits many things including the last time a user logged in, commands that a user executed during a login session, which users attempted or gained access to root privileges, and which users were logged in at any given time. [1] These logs are maintained in the */usr/adm* and */etc* directories. Such information can usually be retrieved by issuing commands such as *ps, who*, *w*, l*astlog*, and *lastcomm*. [5, 1] One may want to prevent outsiders from accessing such information to ensure others' privacy and to limit the release of such system information.

3.1.4 Usability vs. Security

Since UNIX designers placed greater emphasis on the usability of the system, they inadvertently sacrificed its security. Along with the concerns introduced by **set-UID** bits and file links, other user-friendly features exist that must also be carefully used.

3.1.4.1 The PATH Environment Variable - Trojan Horses

The *PATH* variable lists a sequence of directories in which the system should search when it looks for an executable. If the executable is not located in any of the directories listed in the *PATH*, a *"Command not found"* error is returned. This allows users to input simply an executable's name and to let the system search for it, rather than specifying its full path name. For example, if */usr/bin* and */bin* are in the *PATH*, the user can simply type *ls*, *who*, or any other command located in those directories to execute them. [13]

A common practice is to place the current working directory, ".", in the *PATH* so that any program in the current directory could be executed. However, along with this convenience comes the danger of easily placed Trojan horses. Trojan horses are programs that apparently perform a useful or interesting task, but are actually executing malicious activities in the background. For example, if a malicious version of the command *ls* (Trojan horse *ls*) is planted in the current working directory and "." is placed in the *PATH* prior to */bin*, then the Trojan horse *ls* will be executed instead of */bin/ls*. Similarly, Trojan horses can be easily planted in <u>any</u> writable directory that appears prior to the system directories in the *PATH*; it need not be in the current directory. In addition, having an excessively long *PATH* list increases the opportunity for planting a Trojan horse in any one of those directories.

3.1.4.2 Networking - Viruses

UNIX provides friendly network features where one can specify a list of "trusted" hosts (and users). Remote logins (using *rlogin*) and remote command executions (using *rsh* or *rcp*) from these hosts would be permitted without requiring the guest to enter a password. This provides a convenience for users when they use the network since they do not have to repeatedly enter their passwords. [5, 1, 13]

The system administrator (having *super user* access) lists the trusted hosts along with their trusted users in the */etc/hosts.equiv* file (hosts equivalency file). If a user attempts to access the system remotely from one of the hosts listed in *hosts.equiv*, is listed as a trusted user from that host, and has an account on this local system with the specified login name, then the access is granted without requesting a password. Similarly, each user can have a local *.rhosts* file in his/her account which can list further trusted hosts and users not listed in the administrator's *hosts.equiv* file. [5, 1]

Unfortunately, this customizable enumeration of trusted hosts, though convenient, is extremely insecure. The 'trust host' concept gives intruders and viruses (programs which spread themselves secretly) unhindered opportunity to break into other machines in the network. For example, the Internet Worm of November 1988 made use of the cascaded logins to spread quickly throughout the network. [5, 1] Once it broke passwords, it easily navigated through other hosts using the users' *.rhosts* and the */etc/hosts.equiv* files. Knowing this, UNIX administrators are usually very careful when using the */etc/hosts.equiv* file. However, the *.rhosts* files are not under their control and can be created by normal users granting access to whomever they choose, without the administrator's knowledge. [5] Naturally, care must also be taken to prevent outsiders from creating *.rhosts* files.

3.1.4.3 Excessive Resource Usage - Denial Of Service

"The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls .. but rather in lack of checks for excessive consumption of resources." [25]

UNIX has no built-in limits on the amount of resources allowed per user, giving users the freedom to excessively consume disk space, files, swap space, and processes. [25] Although providing no restrictions is user-friendly, this gives easy rise to denial of service attacks where attackers hog system resources to the point of unusability by others. [4] The following is the classic shell program that consumes all the inodes or disk blocks on the system preventing others from writing to the disk [25]:

```
while ; ; do
        mkdir x
        cd x
done
```

The following is a C program that consumes all the system processes forcing the user to reboot the machine:

```
main() {
    fork ();
    main ():
}
```

3.1.4.4 User Configuration Files (Dotfiles)

Users can maintain their own configuration files in their home directories to be accessed by programs that have user customizations. These files include *.cshrc*, *.kshrc*, *.profile*, *.login, .rhosts*, and so on. They are also known as dotfiles since their names begin with a dot ("."). They are not listed as part of the directory contents when using the ordinary *ls* command. These hidden files can only be seen by using the "-a" option with *ls*. This feature provides the capability not to be overwhelmed with these customization files during normal usage of the file system. However, this allows attackers to place other dotfiles covertly in the users file space which would normally not be seen. Care must be taken in preventing outsiders from creating dotfiles if one wants to be aware of all files that are created and of the usage of the file system.

3.1.5 Summary of Safeguards

The following list summarizes the safeguards needed when allowing foreign code to be executed on the local system:

- Protect the directories: */etc*, */adm, /usr*, */var*, */dev*, */bin.*
- Disallow the creation and modification of dotfiles by untrusted sources.
- Place limits on and audit resource usage.
- Disallow the creation and modification of *.rhosts* files.
- Disallow having the same file names as those in */bin*: *ls, more*, ...
- Disallow the setting of SUID and SGID bits on files.
- Disallow writing to SUID and SGID files.
- Allow the following of file links only carefully.
- Disallow the execution of *ps*, *who*, *w*, *lastlog*, *lastcomm*,... by untrusted sources.

3.2 Java Safety

The Java programming language was released in May of 1995 by Sun Microsystems as a platform-independent language that is both safe enough to traverse networks and powerful enough to replace native executable code. [22] Its 'Write Once/Run Anywhere' capability is a result of the Java Virtual Machine which sits in between the native operating system and Java applications as show in Figure 7. The Java source code is compiled into an intermediate byte code which is run inside the Java interpreter. [22, 34, 9] Since Java applications are interpreted, the program can be executed in a secure run-time environment. Both the safeness of the language design and the system architecture help ensure the safeness of Java applications.

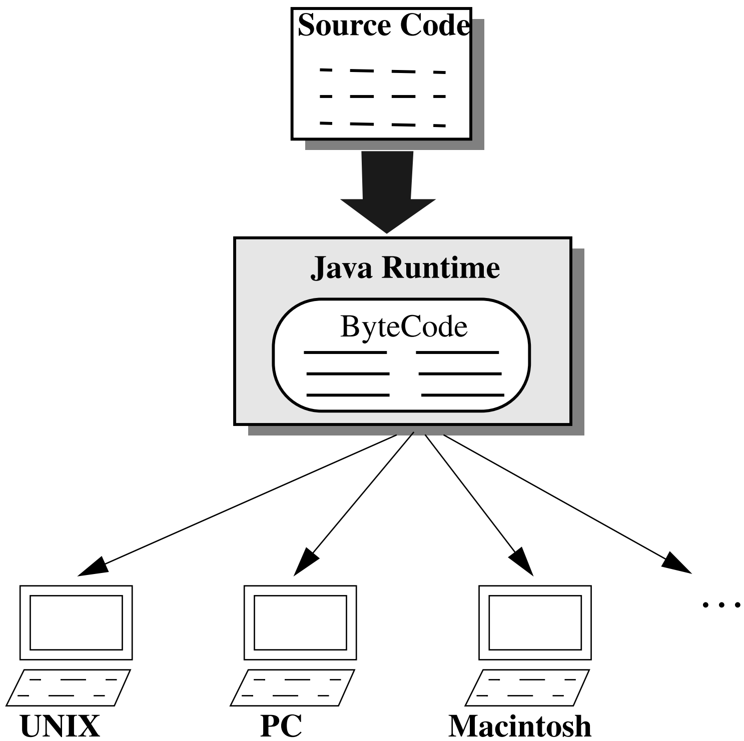

Figure 7. Java Virtual Machine Environment

3.2.1 Language

The safety of the Java language stems from its goal to keep programming simple and straightforward. This avoids the unnecessary common design and programming errors, and equally importantly, prevents malicious programs from accessing and modifying critical parts of the system. [22, 9]

3.2.1.1 Simple Rules

Java has simple rules and features in its language which make it easy to follow and more difficult to make the mistakes often made in other languages. For example, it supports only single inheritance class hierarchy, but allows multiple inheritance of *interfaces*. It allows a class to implement the behavior of an *interface* without needing to be a subclass of the *interface*. This eliminates the need for multiple inheritance of classes, without causing the problems associated with multiple inheritance. [22] Further, Java includes the notion of *packages* for organizing class files. This *package* system allows the Java compiler to handle the operations of *make* automatically. [22] It also has a simple way to manage concurrent threads. The keyword *synchronized* can be associated with a method to designate it for serialized access within the object. The run-time system then ensures that only one *synchronized* method is run simultaneously. [22]

3.2.1.2 No Naughty Pointers

Unlike some other programming languages, Java prevents the programmer from having direct access to the address-space of the program. Java is strictly object-oriented, so that memory can only be allocated and accessed through objects. There are no pointers (but references), no global variables, and no notion of memory 'records' or 'structures'. Since there are no pointers, there can be no pointer arithmetic or pointer de-referencing, forcing programs to access only those objects they create themselves. [23, 34]

Arrays in Java are true arrays in that they are bound and limited to their size. A program that tries to access data outside of the array's range faces a run-time error. Strings also have similar high-level support in Java. Another safety measure is to prevent uninitialized objects from being used or accessed. This restricts programs to stay within their own memory and prevents them from peeking into unauthorized locations. [22]

3.2.1.3 Dynamic Memory Management

In addition to not having ad-hoc pointers, Java objects are automatically created and deleted in the heap by the system's memory manager. The Java run-time system keeps an account of all references to an object and removes it from memory when it is no longer needed. This use of a garbage collector protects programs from having dangling pointers (accidentally freeing memory that is still referenced) and memory leaks (failing to free memory once it is no longer needed).

3.2.1.4 Statically Typed, Late-binding

Every object in Java has a well-defined type that is known at compile time. The Java compiler prevents objects that are assigned to the wrong type or that call nonexistent methods within it. The Java run-time system keeps track of all objects and knows their types during execution. This allows the system to prevent illegal casting of objects from one type to another. [22]

Java is a late-binding language, which means that it locates the definitions of methods dynamically at run-time. This allows the run-time system to determine which method to run in a hierarchical object structure where a subclass can override its superclass' methods. Although this reduces the performance of the program, it makes the behavior of the system straightforward. [22]

3.2.1.5 Encapsulation

The Java language comes with four access modifiers for variables and methods within classes to assert the security permissions on their visibility.

This allows users to encapsulate and protect their data. The permission can be *public* (accessible by anyone), *protected* (accessible only by subclasses and classes within its package), *private protected* (accessible only by subclasses), *private* (accessible only within the class itself), or the default (accessible only within its package). The runtime and compiler checks ensure that these permissions are followed by all objects.

3.2.2 Architecture

Along with the safety provided by the Java language, the Java security model provides three layers of protection around all classes as shown in Figure 8. This is necessary especially if Java binary code is to be downloaded from (trusted and untrusted) sources in the Internet. At the inner level, the *verifier* guarantees the integrity of incoming classes and that the code uses components as they are intended. This allows the *class loader* to guarantee

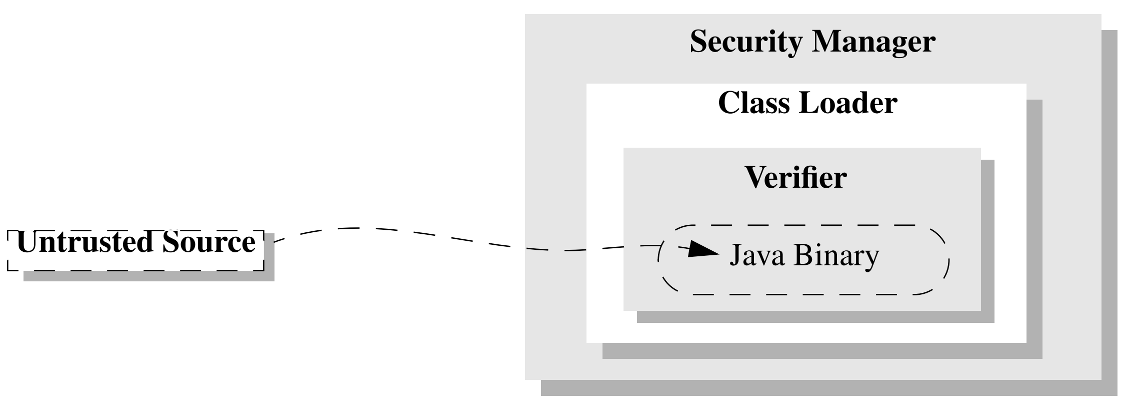

Figure 8. The Java Security Model [22]

that an application is using the core Java system classes and that these classes are the only means of accessing basic system resources. With these restrictions in place, the *security manager* can control access over these system resources. [22]

3.2.2.1 Byte-Code Verifier

Even though the compiler performs thorough type checking, the *verifier* can catch deliberately malicious assembled code not produced by a trusted Java compiler or code damaged via the network. The *verifier* ensures that the class file has the correct format. The byte-code is verified using a simple theorem prover which establishes a set of structural constraints on the bytecode. [34]

The *verifier* also improves the performance of the interpreter since certain expensive run-time checks already statically performed by the *verifier* can be eliminated. These constraints include [34]:

- No stack overflows or underflows
- No illegal register accesses and stores
- No illegal data conversion (via casting)
- No violation of access permissions on objects
- No illegal parameters

3.2.2.2 Class Loader

Bytecode loaded from the system library on the local file system can be downloaded by the Java runtime system. However, to load code from other sources in the network, the *class loader* is used to provide the Java runtime with a downloaded class. Classes are transported across the network as byte streams and restructured into classes by the *class loader*. [7]

The class loader establishes a name space hierarchy. It guarantees that unique name spaces exist for classes that come from the local file system and for those that come from the network. When a class is loaded, it is placed in a private name space associated with its origin and remains assigned to its *class loader*. This partitioning ensures that 1. classes loaded from different network sources are restricted from interacting with other classes, and 2. the system classes are not overridden or replaced by versions loaded from other (trusted or untrusted) sources. [15]

3.2.2.3 Security Manager

The Java *security manager* is responsible for making application-level security decisions. A *security manager* can be installed by an application after which it cannot be replaced or modified. The Java system classes which provide access to system resources always consult the *security manager* before permitting access. The *class loader* together with the *verifier* guarantee that these system classes are the only means by which the bytecode can access the system resources. This permits the *security manager* confidently and solely to grant or deny access to all system resources using any simple or complex policy. This lets the application impose an effective level of trust on the *security manager*.

There are a fixed number of resources to which an application can have access. The *security manager* contains methods for each of these accesses as shown in Figure 9. These methods are called by the system classes to check if permissions to the resources should be given. These resources include the file system, network ports, external processes, system properties, and the windowing environment. If permission is denied, a security exception is raised; oth-

File	
checkRead	Read a file?
checkWrite	Write a file?
checkDelete	Delete a file?
Network	
checkConnect	Connect a socket to this host?
checkAccept	Accept a connection from this host
checkListen	Listen for connections on this port?
Processes	
checkExec	Execute this system process?
checkExit	Execute a system exit?
checkAccess	Access this thread?
Other	
checkProperty(ies)Access	Access this system property(ies)?
checkTopLevelWindow	Create this new top-level window?
checkCreateClassLoader	Create a new class loader?
checkPackageAccess	Access this package?
checkPackageDefinition	Define this package?
checkLink	Link to this dynamic library?

Figure 9. Security Manager Methods

erwise, control is given back to the calling procedure. Additional resources and accesses are, and further can be, included in future Java Development Kits (JDK). For example, Sun's JDK1.1 now also includes *checkAwtEventQueueAccess*, *checkMemberAccess*, *checkMulticast*, *checkPrintJobAccess*, *checkSystemClipboardAccess*, and *checkSecurityAccess*. [29]

An uninstalled *security manager* grants all requests so that the Java virtual environment can perform any activity that is under the user's authorization. However, an application that requires the downloading of untrusted source code, such as a Java-enabled Web browser needing to download applets, should install a *security manager* as one of its first actions. This would ensure that permissions are granted according to that security manager. By default, the base *security manager* simply denies access to all resources. In order to allow applets any access to resources, the methods listed in Figure 9 need to be overridden in a subclass of the *security manager*. [15, 22]

The *security manager* provides a central powerhouse for setting access rules. In making these rules, it can recognize for whom the access is being requested by checking whose *class loader* is currently on the stack. Since each package has its own *class loader*, the *security manager* can recognize which applet, if any, is requesting the permission. This allows the capability of placing a more complex, but fine-grained policy.

3.3 Security Concerns with Java

When granting Java applets access to system resources, the security of the local system must be carefully scrutinized.

3.3.1 Java Security Holes

The Java language was designed to be safe so that secure applications can be built using it. However, as it is extremely difficult to implement bug-free and entirely secure systems, Java also has security holes that are still being discovered by various research teams scrutinizing the system. Since the Java platform has been available to the public eye since the very beginning, its security can be carefully scrutinized, implemented, and repaired.

Examples of security-related bugs include undermining the Java type system, letting applets create their own *class loaders*, loading illegal byte code, using false DNS servers to make arbitrary network connections, etc. [15, 7] These bugs have been promptly corrected in later Java Development Kits and Java-enabled Web browsers once they were discovered. Since Java provides a safe foundation for developing secure network applications, as long as it remains under public scrutiny, it can provide a potentially secure platform. Therefore, in this paper, we assume that such security holes will be corrected as they are uncovered allowing us to think beyond these holes and to create more useful applications.

3.3.2 Respecting Other Applications

When a common set of resources is to be shared by multiple applications and applets, common courtesy should be enforced to respect others. An applet should not affect the execution of other applets and other applications. This also includes the prevention of denial of service attacks which *UNIX* has no safeguard against as described in Section 3.2.1.2. Current Java implementations do not monitor or control resource consumption by applets.

3.3.3 Gossiping Applets - Inter-Applet Communication

If Java applets were given the power to access protected information, can they by forced to keep it to themselves? We can perhaps create a policy where applets are forbidden to connect to the network after reading protected information on the system. However, there are covert channels which applets can use to communicate this information to other applets who would then be able to release this information to the network. In other words, as shown in Figure 10, if Applet A had accessed Protected file F, then knowing this, the *security manager* can prevent applet A from connecting to the network. However, using a covert channel, Applet A can release this information to Applet

B. Not being aware of this transaction, the *security manager* can mistakenly allow Applet B to then transmit this protected information to the network.

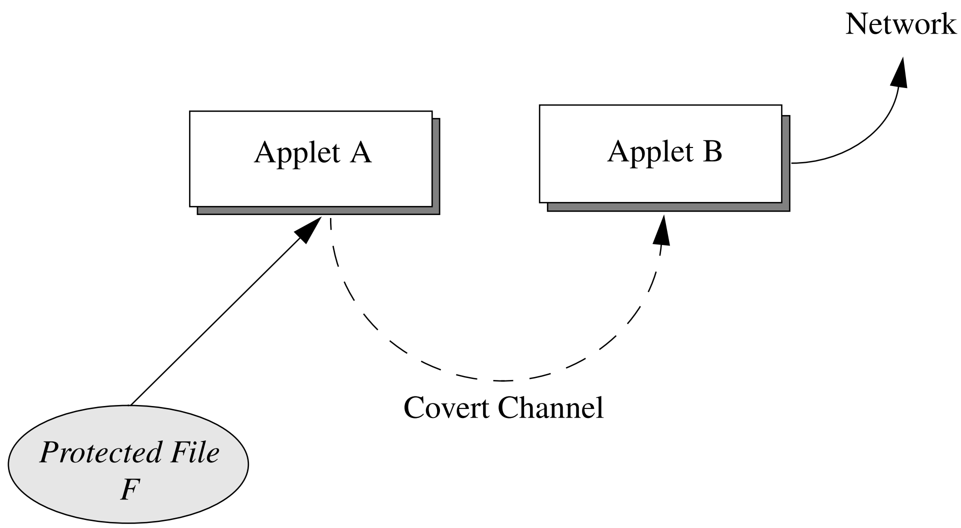

Figure 10. Applet Communication

Such inter-applet communication makes it difficult to keep track of information flow.

3.3.3.1 Via The File System

If applets are allowed to write to the file system, an applet can easily transmit information to another applet by writing that information to a file readable by the other applet. The following scenario depicts this transaction:

File F $\xrightarrow{\text{Information I}}$ Applet A

Applet A $\xrightarrow{\text{Information I}}$ File G

File G $\xrightarrow{\text{Information I}}$ Applet B

Additionally, intra-applet communication can also be achieved using the file system. An applet can store information in the file system which it can later retrieve during the same execution or when it restarts. The system must be aware of the possibility of it accessing and storing protected information for later use.

3.3.3.2 Via The Document

An applet can transmit information to other applets within its document by passing the information within method arguments. Java allows an applet to ask its context (the Web browser or the appletviewer) for the list of applets in its document using the *java.applet.AppletContext.getApplets* procedure or for a specific applet in its document using the *java.applet.AppletContext.getApplet* method. [29] Since an applet has access to other applets, it can invoke other applets' methods and pass any argument it wishes to those methods. For example, the following Java code in Applet A sends protected information to Applet B if Applet A and Applet B are from the same document:

```
AppletB appB = (AppletB) getAppletContext().getApplet(“AppletB”);
appB.covertChannel(“Here is some protected information“, <information>);
```

3.3.3.3 Via Newly Spawned Applets

Java includes a feature whereby applets can request the browser or appletviewer to show a given Web page. Using the *java.applet.AppletContext.showDocument* method, it can replace the current web page with a given URL, show the URL within a certain frame, or show it in a new top-level window. [29] If applets are able to write to the file system, however, this capability enables them to spawn new applets. For example, the applet can write a

Web document (*NewApplet.html*) with an embedded applet and then request the browser to show the document (*file:/<dir>/NewApplet.html*). This, however, opens up another channel through which applets can transfer information to other applets. Particularly, the new class file written by the parent applet can include protected information known by the parent which would then be transferred to the newly spawned applet.

3.3.3.4 Via Selective Resource Usage

A stealthy way for applets to transfer information is by sending signals using system resources. For example, a malicious applet can selectively name the files that it creates so another applet can retrieve information by reading the names of the files. An applet that maintains a private list of your bookmarks file can name some temporary files that have the names of the listed URLs. An accomplice applet can read the contents of the directory and retrieve the names of those URLs from the names of the files. As another example, two applets can communicate using some previously agreed code by the opening and closing of certain network ports.

Such selective naming and usage is difficult to control. However, such communication may be possible to track if applets' operations are audited and suspicious activity is recognized. For example, if odd file names appear or an uncanny number of network ports are opened, one's suspicions would be raised.

3.4 Summary

Java's safe design can provide a solid foundation for creating useful mobile code. However, when exposing system resources to the public domain, security is of great concern. The UNIX platform provides ways to secure one's account and files; however, precautions need to be taken when making the system resources more accessible to outsiders.

Chapter 4

Overall Design

This chapter will give an overall picture of our system and describe the functionality of each module. The details of the implementation will be described in subsequent chapters.

In order to provide a secure, straightforward, flexible, and configurable policy for Java applets, we have designed and implemented a *System* where ordinary users can specify their restrictions using a Simple Constraint Language (SCL). The user specifies his/her policy in SCL and stores it in an *Applet Rules File* (ARF). When an applet needs access to a certain resource, the *Applet Security Manager* (ASM) allows or denies permission to that resource according to the policy set by the user in the ARF. This resolution of permissions is managed by a separate module called *Applet Access Checker*. Further, logs recording applets' past accesses (*Applet Access Logs*) and a log

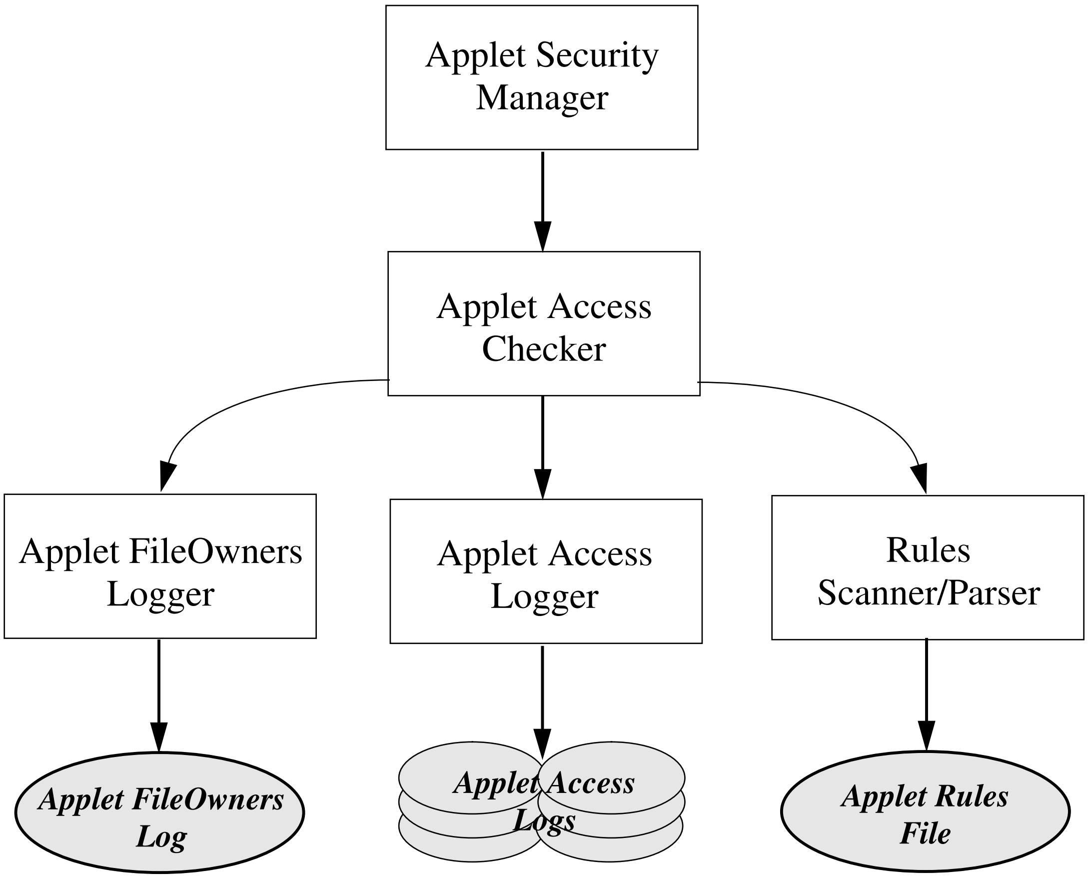

Figure 11. Design Architecture

storing the applet owners of files (*Applet FileOwners Log*) are maintained by the two loggers, *Applet Access Logger* and *Applet FileOwners Logger*. The following sections will describe and explain the need for the elements of the *System* as depicted in Figure 11.

4.1 Applet Rules

The *Applet Rules File* stores the policy that the user wishes to place on applets downloaded onto his/her system. The *Rules* module processes this file and provides information about the rules to the *Applet Access Checker*.

SCL also provides a way to label applets according to their state. This allows users to categorically state their policies by identifying applets by these security labels in addition to their names and origins. For example, an applet can be labelled suspicious if it accesses more than 25 files, or becomes private after accessing private files. Further rules would state that suspicious applets can no longer access private resources, and that private applets can no longer connect to the network.

More details about SCL, the *Rules* module, and the labelling scheme are described in Chapter 5.

4.2 Applet Access Logs - Applets' Karmic State

The user's policy is a function of the applet's identity and its past accesses. This is dissimilar from the Access Control List or capabilities technique used by most operating systems. The permission of an applet depends not only on its identity but also on its current state; its current state is the sum of all its past accesses. Keeping a record of applets' past accesses requires keeping *Applet Access Log (AAL)* files which are maintained by an *Applet Access Logger*.

AAL files allow users to create much more useful policies. For example, when determining whether an applet should be allowed to connect back to its host, user Joe would be unwilling to permit this only if the applet would be compromising his privacy. The integrity and reliability of his system would not be damaged by this connection assuming the network connection does not drastically slow down his system. In order to assess safely whether the applet would compromise his privacy, he would need to determine whether it has accessed any of his private information. (This is better than having Joe blindly trust the applet if it came from a trusted source.) Providing an auditing mechanism on applets as is done by our *Applet Access Logger*, allows Joe to peruse the applet's past history and to see whether any of his private information were accessed. (Having Joe specify this policy in his *Applet Rules File* does not require him to repeatedly reassess his policy). Of course, Joe is also free to disallow all access to his private information if he so wishes. However, this logging feature provides Joe a way to be less restrictive.

Each applet merits its own AAL file in order to search efficiently through an applet's log. An entry in the log will state the resource used, the access made, the name of the resource, and the number of times this access was made. For example: [File, Read, "/Public/FileName, 9] or [Host, Connect.To, web.mit.edu, 5]. This information sufficiently provides the information needed to decide future permissions.

4.3 Applet FileOwners Log - Mute Applets

In order to address the issue of applet communication described in Section 3.3.3, the *System* entirely prevents applets from communicating with one another *(via the File System)*. In the cases where it cannot prevent transmission of information *(via the Document)*, it considers those gossiping applets to be in the same category of secrecy. In the case where there is only a one-time exchange *(via Spawned Applets)*, it updates the culprit's log file to account for this transfer.

4.3.1 Via the File System

Since the *System* is fully aware of which applets have accessed which files, the *System* is capable of disallowing applets from accessing files written by other applets. This provides a way to create individual storage space for applets without risking the possibility of privileged applets revealing information to non-privileged ones. (A possible future extension that allows file sharing among applets is described in Section 7.1.3.) An applet becomes an owner of a *File F* once it has written to a pre-existing, non-previously-owned *File F* or has created a new *File F*. Other applets cannot read, write, or delete *File F* since it is not owned by them. This ownership will last through the stopping and restarting of the applet, and through the exiting and reexecuting of the Web browser. The ownership will last as long as the applet's stored information remains accessible. The applet will therefore continue to be the owner of *File F* until *File F* is deleted.

One can argue that it would be safe for non-owners to delete *File F* without allowing them to read from or write to it since there would be no transfer of information. However, allowing the deletion of another applet's files would encourage malicious applets wanting to prevent another Web site's applets from accessing their own (temporary) files. Therefore, we have also disallowed the deletion of files owned by other applets.

Since current operating systems do not support the logging of which executables have modified which files, the *System* keeps a log of this information in the *Applet FileOwner s Log (AFL)*, maintained by *Applet FileOwner s Logger*. The AFL is indexed by the file's name and associates each file with the identity of the owner applet and the time the file was last accessed. The *Applet Access Checker* uses this file through the *Applet FileOwner s Logger* in order to determine file accesses for all applets. A file's entry in AFL is deleted when that file no longer exists, that is, when the file is deleted by the owner applet or directly by the user via the operating system.

4.3.2 Via the Document

Since Web browsers allow applets to pass information to cousin applets from the same document (as described in Section 3.3.3.2), we need to account for the possibility of information leaking through this manner. If *Applet A* has been given access to *Information I*, we must be aware that *Applet B* from *Applet A*'s document, can also access *Information I* from *Applet A*. Similarly, *Applet A* can access information from *Applet B*. So both *Applet A* and *Applet B* can be considered to have the same state of combined information.

Using this argument, when determining whether a permission should be granted for *Applet A*, we make sure the same permission should be granted for *Applet B*. If there are N applets in the document, then the permission is checked for all N applets. Using this protocol, we account for the information exchange between the cousin applets.

Another approach would have been to give permissions and to log past accesses based on the identity of the Web documents instead of the identity of the applets. Rather than the individual applets, each Web document would be held accountable for the actions by all the applets within it. However, this limits an applet residing in multiple documents from accessing its own temporary files. The files would need to be associated with one of those documents instead of the applet.

4.3.3 Via Spawned Applets

We also need to account for the possibility of information exchange to spawned applets. As described in Section 3.3.3.3, this communication is a one-way, one-time transmission. When a new applet is spawned, it can carry with it the latest state of the parent applet. In other words, the total information that the parent knows at the point of birth can be transferred to the newly-born applet. Since we consider the state of an applet to be stored in its *Applet Access Log* (AAL), we can simply view this information exchange as the spawned applet inheriting a copy of this log. So the spawned applet must also bare the burden of and be accountable for its parent's past history. However, afterwards, the parent and child go their separate ways without any further communication, allowing them to make their own history keeping their separate AALs.

If spawned applets are loaded via the `file` protocol, detecting them is trivial since we log the applet ownership of files. When the applet is downloaded, we can check in *Applet FileOwners Log* whether the file in the URL is owned by another applet; the owner would be it's parent. If there is a parent, the spawned applet's AAL is initialized to the parent applet's AAL.

If spawned applets are loaded via any other protocol such as `http`, it is more difficult to detect them. However, it is also more difficult to create such spawned applets since this requires 1. the parent applet to be allowed to write to document space on a Web server and 2. the parent applet to have knowledge of the Web server's file structure. The latter can be easily achieved by snooping around the server, however, it is strongly suggested that precaution be taken to avoid the former.

4.4 Applet Security Manager

As was detailed in Section 3.2.2.3, the Java Security Manager is responsible for granting and denying access to resources. The *Applet Security Manager* (ASM), the security manager for our *System*, delegates the responsibility of determining the permissions for applets to the *Applet Access Checker* (described in Section 4.5). The ASM asks the *Applet Access Checker* whether to grant the applet a particular access (i.e. read) to a particular resource (i.e. a file) giving additional information about the resource (i.e. the file's name). The *Applet Access Checker* responds in the affirmative or the negative. If negative, the *Applet Security Manager* will signal a security exception, otherwise, it does nothing.

4.5 Applet Access Checker

The *Applet Access Checker* manages the entire *System*. It stores in memory the applet policy parsed and scanned by *Rules*, and makes sure the two loggers, *Applet Access Logger* and *Applet FileOwners Logger*, are updated with the latest information. Using these three subordinates it responds to the *Applet Security Manager*'s inquiries about applet permissions.

4.5.1 Logic Flow

In order to determine whether applet P can be granted access A to resource R named N, the following steps need to be taken as shown in Figure 12. First, if R is a file, then verify that N is not owned by another applet using the *Applet FileOwners Logger*. If not, verify that N is not one of the *System*'s file (i.e. *Access Log Files*, *FileOwners Log File,* or *Applet Rules File*). If not, verify that it is not one of Java's system files (i.e. in `java.class.path`). If all of the above is fine, or if R is not a file, then determine which rules in *Rules* affect access A for resource R. (Chapter 5 will go into more detail about the backward searching of rules and how conflicting rules are resolved.) Both the applet and its cousins (other applets from the same document as described in Section 4.3.2) must pass these rules. For the rules that require knowledge of the applets past history, the *Applet Access Logger* is questioned..

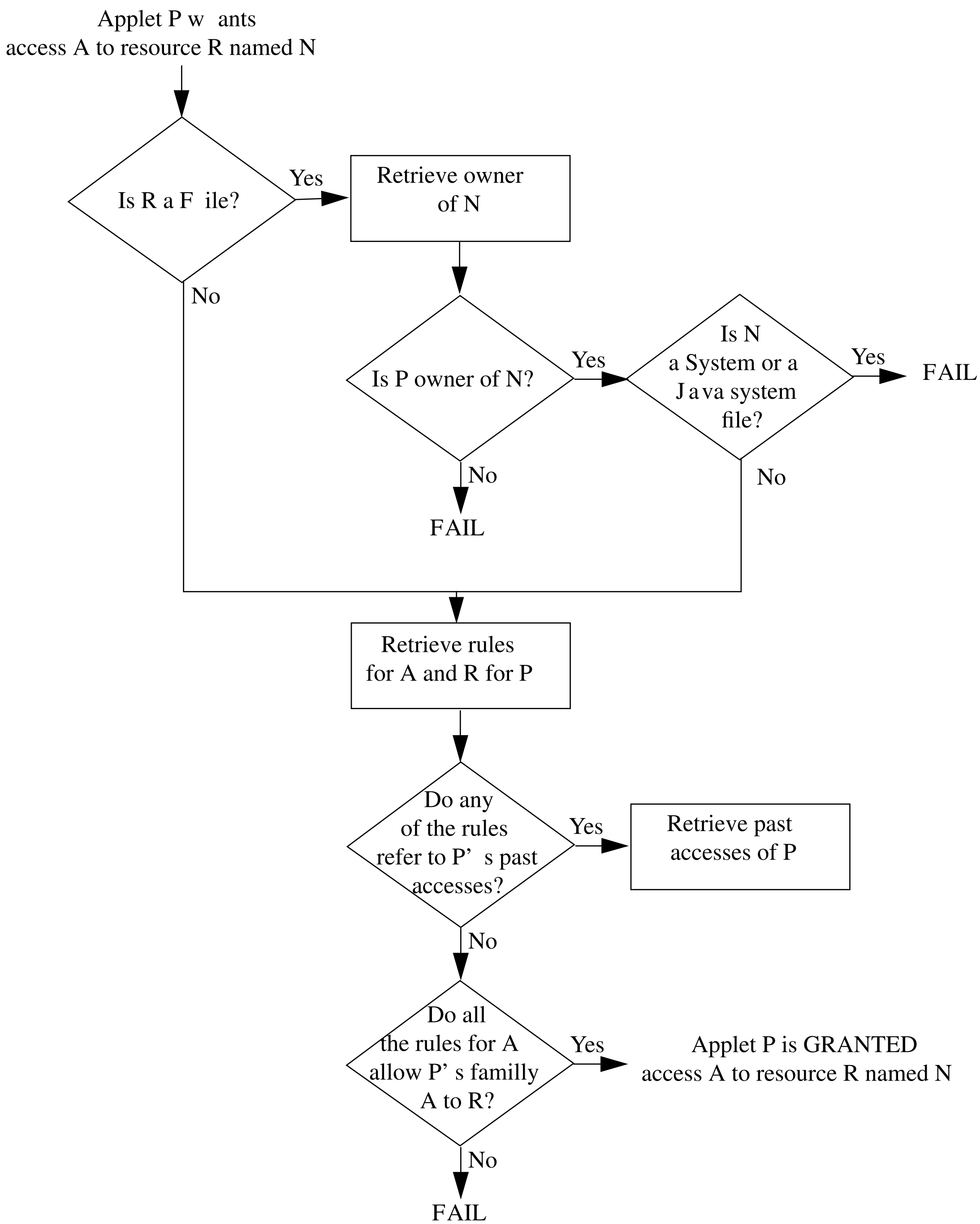

Figure 12. Logic Flow

4.6 Loggers

The two loggers, *Applet FileOwners Logger* and *Applet Access Logger*, are alike in their functionality although unique in their usage. They have therefore been designed similarly but with striking differences.

4.6.1 HashCache

Extensive reading and writing to and from the file system requires many time-consuming I/O operations. Since both loggers need to store their information in files, an efficient design is required to account for the multiple updates and read requests to and from these files. Therefore, a caching scheme is used to store and retrieve the data. In order to make the data retrieval even more efficient, the data is stored in a hashtable indexed by the file’s name for the *Applet FileOwners Logger* and by the applet’s name in the *Applet Access Logger*. This combination of using a hashtable and a cache is termed HashCache.

4.6.2 Log Cleanup

The practice of keeping a log naturally leads to the question of when/how to garbage collect the entries to prevent a forever growing collection on limited disk space.

This question is easily answered for the *Applet FileOwners Logger* which has an entry for each file owned by an applet. The number of entries cannot forever increase because of the limitation on the number of files placed by operating systems. As the applets' files are created and deleted, the entries are also created and deleted respectively.

However, garbage collecting the *Applet Access Logger* is a bit more tricky since logs can grow endlessly with time. As applets do more and more things, their history will forever increase; so when can one safely delete an applet's log? An applet's log is no longer needed if we know the applet will never be executed again. However, since the future cannot be predicted, another alternative is needed. The *System* safely deletes an applet's history once the applet no longer owns a file and has exited. The reasoning is that if the applet is no longer running and it does not leave any information behind, then there is no way that it can store any previously accessed protected data. Since it has no remains in file storage, it is safe to start it off with a clean slate the next time it runs. This logic allows us periodically to clean up applets' logs.

4.6.3 Log Location

The logs cannot be modified by applets. They are stored in a private location which may be configured by the user. In addition, the *Applet Access Checker* makes sure that applets do not access these files.

4.7 Summary

This chapter delineated the overall design of the *System* excluding the *Rules* module which will be described in the next chapter. The *System* gives a solution to the need-for-fine-grained-control-over-applets problem introduced in Chapter 1 while addressing the security issues described in Section 3.3. The solution is to allow users to specify their policies using a simple constraint language and to allow the categorical labelling of applets . The *System* does this while ensuring the applets are mute and are accountable for their past history . It makes use of an intelligent *Applet Access Chec ker* for managing the processes involved and Loggers for recording information about the applets .

Chapter 5

Rules

Our system allows users to specify their policies on applets by declaring rules that applets must follow. In order to specify these rules in a straightforward manner, we have designed a *Simple Constraint Language* (SCL) in which users describe the rules in their policy. These rules are then used to determine permission for a particular access to a particular system resource for a particular applet.

5.1 Simple Constraint Language (SCL)

SCL is a strongly typed, case-insensitive language. It includes four major constructs: *constants*, *variables*, *primitive procedures*, *and statements*. All four are used to create comprehensible rules.

5.1.1 Types

All expressions in SCL are strongly typed to catch mismatched type errors during the scanning of the rules. They can be either of type **string**, **digit**, or **boolean**. *Variables* of type *T* can be assigned only values or expressions of type *T*. Expressions of type T can be compared only to other expressions of type T. A list of type T contains only expressions of type T. **Boolean** operators expect only expressions of type **boolean**.

5.1.2 Constants

There are three types of constant values. **String** literals are denoted by surrounding the text with quotation marks (i.e. *"www.mit.edu"*, *"/usr/bin"*). **Digit** values are positive or negative integers (i.e. *5000*, *-10*). **Boolean** values are either *true* or *false*.

5.1.3 Variables

SCL supplies a fixed number of *variables* that can be used to obtain information about the applet (*applet variables*) and the resource (*resource variables*), and to give a specific access to that resource (*access variables*). As in Java, *variables* are specified with a hierarchical syntax using dots ("."), providing comprehensibility and easy extensibility for future additions.

5.1.3.1 Applet Information *(Applet Variables)*

Four kinds of *applet variables* are provided to identify the applet: *Name*, *CodeBase*, *Document*, and *Category*. The first three provide readable information about the origin of the applet. These can be easily extended to include information about the digital signers and endorsers of the applet and document in future models. The *Category* of the applet provides a modifiable label which can be used to flexibly group applets according to some measure. A label can be set once an applet has done something in its history or can be based simply on its origin. For example, an user can assign a label for *applets-that-have-read-protected-files* and for *applets-that-are-trusted-to-access-the-mailbox*. However, in order to allow an ordinal ranking to the labels, these labels are implemented as integers. An applet's label is 0 by default.

Figure 13 lists the *applet variables*, including their types. For the examples included, assume the user has downloaded the applet *http://www.applets.com/scheduler/addressbook/main.class* from the document *http://web.mit.edu:80/applets/schedulers.html.*

5.1.3.2 Resource Information *(Resource Variables)*

Variables are provided to identify system resources that are available to applets. These include *File*, *Directory*, *Host*, *Command*, *Thread*, and *Property*. With JDK1.1, Print and Dynamic Linking can also be included. Figure 14 lists the read-only *variables* that provide information about the resources.

5.1.3.3 Access Permission *(Access Variables)*

Each resource has associated with it accesses that can be permitted to it. The **boolean** *variables* in Figure 15 can be set to *true* or *false* specifying whether the access should be permitted. These *variables* can be assigned but not read. Otherwise, if they could be read, then the possibility of infinite loop-

Variable	**Type**	**Description**
Name: the applet		
Applet.Name	string	The full package name of the applet i.e. *scheduler.addressbook.main*
CodeBase: the applet's origin		
Applet.CodeBase.Name	string	The full URL of the applet i.e. *http://www.applets.com/scheduler/addressbook/main.class*
Applet.CodeBase.Host.Name	string	The DNS name of the applet's server i.e. *www.applets.com*
Applet.CodeBase.Host.IP	string	The IP address of the applet's server i.e. *18.249.0.37*
Document: the applet's document		
Applet.Document.Name	string	The full URL of the document i.e. *http://web.mit.edu:80/applets/schedulers.html*
Applet.Document.Host.Name	string	The DNS name of the document's server i.e. *web.mit.edu*
Applet.Document.Host.IP	string	The IP address of the document's server i.e. *18.249.0.37*
Category: the applet's state (label)		
Applet.Category	digit	The label placed on the applet i.e. *Trusted(7)*, *Suspicious*(3), *Untrusted (0)*

Figure 13. Variables providing Applet Information

ing would arise in the rules. Specifically, if one wanted permission to make a *File.read* access and a rule that granted *File.read* access needed to know the permission for a *File.write* access, then the permission for *File.write* access would need to be evaluated. If that in turn needed to know the permission for *File.read* access, then a runtime looping error would occur. In order to avoid such complexity, these access variables are writable only.

Variable	Type	Description
File.Name	string	The name of the file. e.g. *hosts*
File.Path	string	The path of the file which may or may not be absolute. e.g. *etc/hosts*
File.AbsPath	string	The absolute path of the file. e.g. */etc/hosts*
File.Parent	string	The parent directory of the file. e.g. */etc*
File.Size	digit	The file's size. e.g. *2000* (for 2Kilobytes)
Directory.Name	string	The name of the directory. e.g. *inbox*
Directory.Path	string	The path of the directory. e.g *Mail/inbox*
Directory.AbsPath	string	The absolute path of the directory. e.g. */user/nimisha/homes/Mail/inbox*
Directory.Parent	string	The parent directory of the directory. e.g. */user/nimisha/homes/Mail*
Directory.Size	digit	The length of the directory. e.g. *34816*
Host.Name	string	The name of the host. e.g. *www.sun.com*
Thread.Name	string	The name of the Thread.
Thread.Group	string	The name of the Thread's group.
Command.Name	string	The name of the command. e.g. *ls*
Property.Name	string	The name of the property. e.g. *user.home*

Figure 14. V ariables identifying System Resources

Resource	*Access Variable*	Description
File	*File.read*	Whether the file can be read.
	File.write	Whether the file can be modified.
	File.delete	Whether the file can be deleted.
Directory	*Directory.read*	Whether the directory can be read.
	Directory.write	Whether the directory can be modified
	Directory.delete	Whether the directory can be deleted.
Host	*Host.Connect.To*	Whether the host can be connected to.
	Host.Connect.From	Whether connections can be accepted from the host.
Thread	*Thread.Access*	Whether the thread can be accessed.
Command	*Command.Exec*	Whether the command can be executed.
Property	*Property.Read*	Whether the property can be read.
	Property.Write	Whether the property can be modified.
Window	*Window.Create*	Whether the window can be created.

Figure 15. V ariables giving Accesses to System Resources

5.1.4 Primitive Procedures

Primitive procedures allow a way to determine useful information during run-time using *constants* and *variables*. Each procedure has at least one argument on which it operates and returns one value. Both the arguments and the return value are strongly typed. Except for the *Past* procedure, procedures are syntactically written in prefix notation.

5.1.4.1 Boolean Operations

((boolean, boolean) | (boolean)) ==> boolean

The *Boolean Operations* are *And*, *Or*, and *Not*. They all take in **boolean** arguments and return a **boolean** value. *And* and *Or* are binary operators, while *Not* is a unary operator. *And* returns *true* only if both of its arguments have the value *true*; otherwise, it returns *false*. *Or* returns *false* only if both of its arguments have the value *false*; otherwise, it returns *true*. *Not* returns *false* if its argument has the value *true*; otherwise, it returns *true*. This is illustrated below:

```
( And true ( And true false ) )  ==> false
( Or  ( Not true ) true ) ==> true
```

5.1.4.2 Comparison Operations

((string|digit), (string|digit)) ==> boolean

The *Comparison Operations* are equal ("="), not equal ("!="), greater than (">"), less than ("<"), greater than or equal to (">="), and less than or equal to ("<="). They are all binary operations which take in **string** or **digit** arguments and return a **boolean** value. Both arguments must be of the same type. If the arguments are of type **digit**, a numerical comparison is done. If the arguments are of type **string**, a lexicographic comparison is done. The following provide some examples:

```
( < 5 2 ) ==> false
( < "hello" "hi" ) ==> true
( = Applet.Category "Trusted" ) ==> run-time comparison
```

5.1.4.3 Match Procedure
(string, string) ==> boolean

The *Match* procedure provides a way to do pattern matching on strings. It is a binary procedure that takes in two **strings** with the first argument being the *test string* and the second argument the *pattern string*. A **boolean** specifying whether the *test string* matches the *pattern string* is returned. The *pattern string* is a limited regular expression in that it may have one or more asterisks ("*") as wild card characters which can match any string, including the null string (as is similarly done in Unix shells). The following examples illustrate its functionality:

```
( Match "hello there" "*ello*" ) ==> true
( Match "abc" "*cd" ) ==> false
( Match "web.mit.edu/nimisha/www/index.html" "web.mit.edu/*" ) ==> true
( Match File.Path "/Public/*" ) ==> run-time match
```

5.1.4.4 OneOf Procedure
((string|digit), (string|digit)+) ==> boolean

The *OneOf* procedure is useful in determining whether an element belongs to a list. It is a binary operator that takes in a *test element* and a *list* of elements. The type of the *test element* must be the same as the type of the elements in *list*. That is, if the test element is a **string**, *list* must be a list of **strings**. The *OneOf* procedure returns a **boolean** specifying whether the *test* *element* is included in the *list*. If the *list* is a list of **strings** that include *patterns*, then the *Match* procedure is used to determine whether the *test element* matches one of the *patterns* in the list. Examples follow:

(OneOf 9 (1 5 9)) ==> *true*
(OneOf "hi" ("hola" "hello" "hi" "namaste")) ==> *true*
(OneOf "hi" ("h*" "namaste")) ==> *true*
(OneOf File.Name ("ls" "csh" "more")) ==> *run-time check*

5.1.4.5 Count and CountAll Procedures

boolean variable ==> digit

The *Count* and *CountAll* procedures are provided for totalling the number of accesses to a particular resource or for all resources, respectively. They take in an *access variable* as their argument and return a **digit** specifying the total access. For example, (Count file.read) will return the total number of times the file in question was read by this applet, and (CountAll file.read) will return the total number of times all files were read by this applet. The following constraint is always kept:

(>= (CountAll Command.Exec) (Count Command.Exec)) ==> *true*

5.1.4.6 Past Procedure

(Any *identifier* in Past *X predicate*)
(All *identifier* in Past *X predicate*)

The *Past* procedure is a powerful tool in determining the applet's state inclusive of its past actions. The *predicate* is evaluated over the applet's past usage of resource (or access) *X*. The purpose is to find out the truth of the *predicate* in the applet's past history with *X*.

Two operators are provided, *All* and *Any*, which correspond to the mathematical notations ∀ (for every) and ∃ (there exists). *All* is used to verify the truth of the *predicate* for every *X* used in the past. With *All*, the *predicate* will be true if and only if all *X* satisfied the *predicate*. (The null set is considered to have satisfied the *predicate*.) *Any* is used to confirm the truth of the *predicate* for at least one *X* used in the past. With *Any*, the *predicate* will be true if and only if there existed at least one *X* that satisfied the *predicate*. (The null set is therefore considered to not satisfy the *predicate*.)

X can be either of two things: 1. a resource name or 2. a resource's *access variable*. For example, if *X* is "File", then it corresponds to all the files that the applet had accessed in the past. However, if it is "File.Read", then it corresponds to only the files that the applet had read in the past.

The *identifier* is used to provide a nomenclature to identify the particular past resource inside the *predicate*. When the identifier appears in the predicate, it assumes the role of the resource in *X*. Information about the resource can be inquired within the *predicate* by using the resource information variables described in Section 5.1.3.2, but replacing the resource name with the *identifier*. For example, if *X* is "File" or "File.Read", and the *identifier* is "f", then when f.name and f.parent appear in the *predicate*, they refer to information about the current file in question. In other words:

Any: If ($\exists_{\text{identifier}}$ Σ *PastX* s.t. *predicate*), then true, else false
All: $\forall_{\text{identifier}}$ Σ *PastX* , if (*predicate*), then true, else false

The following examples illustrate their functionality:

```
( Any f in PastFile ( OneOf f.path ProtectedDirs ) )
```

```
( Any f in PastFile.Write ( > f.size 3000 ) )
( All h in PastHost ( = h.name Applet.codebase.host.name ) )
( All c in PastCommand.Exec ( and ( <= (count c.exec ) 1 )
                                    ( OneOf c.name SafeCommands )))
```

5.1.5 Statements

The aforementioned *constants*, *variables*, and *primitive procedures* are combined and used in statements to describe the rules of the applet policy. *Define* statements provide a mechanism for declaring global constant values. *Assignments* allow a way to modify variables. *If-Else* statements allow a way to set preconditions to assignments. *Begin* statements allow the grouping of statements. The syntax of these statements is derived from the Lisp language.

5.1.5.1 Define Statements

(define *identifier value*)

The *Define* statement includes the keyword "Define", the *identifier* being defined, and the *value* to be assigned. The *value* can be either a list of elements (encapsulated within parentheses) or a single element of any type: **string**, **digit**, or **boolean**. The elements can even be *identifier*s defined in previous *Define* statements. *Define* statements are used to provide global *identifier*s for constant values. These values therefore cannot be *variables* or *primitive procedures*. The following examples illustrate the syntactic representation:

```
( Define WriteableDirs ( "/user/nimisha/AppletTmp/*" ) )
( Define ReadOnlyDirs ( "/ftp/pub/*" "/user/nimisha/Public/*" ) )
( Define ReadableDirs ( WriteableDirs ReadOnlyDirs ) )
( Define FileSizeLimit 5000 )
```

5.1.5.2 Assignments

(*variable* = *value*)

An *Assignment* includes a writeable *variable*, the assignment operator "=", and the *value* to be assigned. The types of the *variable* and the *value* must be the same. *Assignments* can be globally declared or placed within *If-Else* statements as will be described later. Contrary to *Define* statements, *Assignments* can have more complex expressions including *variables* and *primitive procedures* as their *values* as shown below:

```
( Host.Connect.To = false )
( File.Write = ( and ( Match File.Path WriteableDirs ) )
                     ( < File.Size FileSizeLimit ) ) )
```

5.1.5.3 If-Else Statements

(If *predicate* *consequent* (Else *alternative*))

An *If-Else* statement provides a straightforward way to describe a rule. It sets preconditions on the assignment of variables. It consists of a *predicate* which must have a **boolean** return value, a *consequent* statement which is evaluated if the *predicate* is true, and an *alternative* statement which is evaluated otherwise. The *consequent* and *alternative* statements are local statements (i.e. not *Define* statements). For example, the *Assignment* above can be rephrased as:

```
( If ( and ( Match File.Path WriteableDirs )
           ( < File.Size FileSizeLimit ) )
        ( File.Write = true )
 ( Else ( File.Write = false ) ) )
```

5.1.5.4 Begin Statements

(Begin *local-statement* 1 *local-statement* 2 ...)

Begin statements provide a way to group local statements together including *If* statements, *Assignments*, and other *Begin* statements. *Define* statements are not included since they are global. The following is an example of its usage:

```
( Begin
      ( File.Write = false )
      ( File.Read = true )
      ( File.Delete = false ) )
```

5.1.6 Example

Figure 16 shows an example of an applet policy constructed using SCL. SCL allows the use of "/*", "*/", and "//" to form comments as is done in Java.

5.2 Rules Resolution

SCL provides a syntactic language to specify the rules. However, a further mechanism is provided to evaluate these rules during run-time and to conclude the permissions that are set.

5.2.1 Static Simplifications

The rules are analyzed and simplified statically using copy propagation, constant folding, and typing to efficiently decrease the run-time evaluation of the policy. These simplifications are done during the parsing of the rules.

5.2.1.1 Copy Propagation

The constant values declared in the *Define* statements are statically substituted within the rules in which the definition's *identifier* are found. This reduces the need for an extra lookup table and allows further simplifications of the constant values. Table 1 gives an example.

```
/* Declare Readable Directories. */
( Define  ReadableDirs   ( "/user/nimisha/Public/*"   "/www/"" ) )

/* Declare Writeable Directories. */
( Define  WriteableDirs  ( "/user/nimisha/AppletTmp" ) )

/* Allow read access to Readable directories. */
( If  ( OneOf  File.Path  ReadableDirs )
        ( File.Read = true )  )

/* Allow read, write, and delete access to Writeable directories. */
( If  ( OneOf  File.Path WriteableDirs )
        ( Begin
            ( File.Write = true )
            ( File.Read = true )
            ( File.Delete = true )  )  )

/* Limit usage of total file space for all applets. */
( If  ( >  ( Countall File.Size )  20000 )
         ( File.Write = false )  )

/* Declare Protected Directories. */
( Define  ProtectedDirs  ( "/user/nimisha/Mail/" ) )

/* Allow read access to the Mail directory to mail organizer applets. */
( If  ( And ( Match File.Path  "/user/nimisha/Mail/*" )
          ( Match Applet.CodeBase.Name  "web.mit.edu/applets/Mail"  )  )
        ( Begin
            ( File.Read = true )
            ( File.Write = true )  )  )

/* Keep Protected information inside. */
( If  ( Any f in PastFile  ( OneOf  f.path  ProtectedDirs ) )
        ( Host.Connect.To = false  )
( Else  ( Host.Connect.To = true  ) )  )

/* Allow creation of windows by default. */
( Window.Create = true )

/* Limit the number of windows created. */
( If  ( > ( Countall Window.Create )  50 )
       ( Winocw.Create = false ) )
```

Figure 16. Example of an Applet P olicy

(Define X 1) (A = X)	===>	(A = 1)

TABLE 1. Copy Propagation from Define Statements

5.2.1.2 Constant Folding

If there are any constant **boolean** values in a *Boolean Operation*, the expression is easily reduced to a single value. The following table summarizes the reductions:

(And X false) ==> false	(And false X) ==> false
(And X true) ==> X	(And true X) ==> X
(Or X false) ==> X	(Or false X) ==> X
(Or X true) ==> true	(Or true X) ==> true

TABLE 2. Constant Folding in Boolean Operations

5.2.1.3 Typing

Since SCL is strongly typed, all variables, values, and expressions are typed statically to prevent re-parsing during run-time. Mismatched typed errors are caught during the parsing and signalled to the user.

5.2.2 Backward Searching

When a permission is to be checked for a particular resource, the user's applet policy is referenced. Instead of verifying all the rules each time an access is to be granted, only the rules that affect this access are verified. The other rules are considered as dead-code at that time. However, since the applet's label can change at any time, rules that affect *Applet.Category* are always re-evaluated. For example, if permission to read a system property is requested, then only global *Assignments* and *If-Else* statements that affect *Property.Read* and/or *Applet.Category* are evaluated, while the other rules are discarded. This is done for efficiency reasons only.

5.2.3 No Redundancies

To avoid confusion and to keep the user's policy clear, the system disallows global re-definitions and global re-assignments. So there can be at most one *Define* statement for each *identifier*, and at most one global *assignment* for each *variable*. Any redundancies are detected statically during the parsing of the rules and signalled to the user. However, redundancies among local assignments in *If-Else* statements are still allowed.

5.2.4 Resolving Access Conflicts

Since *access variables* can be assigned in global *Assignments* and in multiple *If-Else* statements, conflicts may arise from these plural assignments. For example, one rule may assign *File.Read* to false, while another assigns it to true. A safe evaluation scheme is provided to resolve such conflicts.

The system uses a straightforward logic to evaluate the rules for a given access. By default, the access is false. So if there are no rules affecting this access, permission is not granted [Φ ==> false]. If there are rules, the final permission is the "and" of all the permissions set by the rules. So, if there is at least one rule that denies access, the permission is denied [$(T)^*(F)^+$ ==> false]. In other words, in order for permission to be granted, <u>all</u> the rules must permit access [$(T)^+$ ==> true].

5.2.5 Resolving Labelling Conflicts

Similarly, since the category can be modified by any rule, conflicts may arise from multiple assignments to an applet's category. For example, consider the policy written below. It labels all applets Untrusted by default, Trusts those applets that come from *www.trust.org*, and regards those applets that accessed any protected files as Internal. Since more than one of these rules can be fired for a particular applet at a particular state, a safe resolution scheme is provided.

```
/* Set the Category to Untrusted by default. */
( Applet.Category  =  Untrusted  )

/ * Declare the applets that we trust. */
( If  ( =  Applet.CodeBase.Host  "www.trust.org"  )
        ( Applet.Category  = Trusted  ) )

/* Declare the applets that should be internal. */
( If  ( Any f in PastFile ( OneOf  f.Path  ProtectedDirs )  )
        ( Applet.Category =  Internal )  )

/* Allow trusted applets to read from the file system. */
( If  ( >= Applet.Category  Trusted )
        ( File.Read = true ) )

/* Disallow Internal applets from connecting outside. */
( If  ( <  ( Applet.Category Internal )  )
        ( Host.Connect.To = false )
( Else ( Host.Connect.To = true ) )  )
```

The system enforces the labels to be implemented as type **digit** in order to specify a way to order the security levels of each category. So for the above policy, the following needs to be declared:

```
( Define  Untrusted  1 )
( Define  Internal     5 )
( Define  Trusted    10 )
```

The least secure label should have the least value and the most secure should have the highest. Conflicts in assigning labels are then safely resolved by conservatively setting the label to the minimum of the values . By default, the label is set to 0 for all applets .

5.3 Summary

SCL was designed as a straightforward, comprehensible , and flexible language for specifying the policy . The *variables* provide a common vocabulary for receiving information and setting permissions . The *primitive procedures* provide tools for computing useful information about the applets . The *statements* allow a way to formulate the rules in a logical structure . These rules are then evaluated and resolved at run-time .

Chapter 6

Implementation Notes

This chapter describes the pertinent details of the implementation of the system. The system is developed on the Sun SPARC platform using JDK1.0.2. It modifies the appletviewer's *Security Manager* that comes with JDK1.0.2. The Java source code is retrieved from the class files using Java Mocha Decompiler [31]. Neither the Java Virtual Machine nor the system classes are modified.

6.1 Applet Security Manager

The *Applet Security Manager* (ASM) is questioned by the Java system classes when access to a system resource is requested. When one of the *checkX* methods listed in Figure 9 is called, the ASM must determine whether the request comes for an applet or for the application. It does this by checking whether a *class loader* is on the execution stack. Since *class loaders* are only used for loading non-system classes, having a *class loader* on the stack implies that an applet was loaded.

Being a centralized authority, the ASM is responsible for responding to all the queries in a multi-threaded environment. Therefore, all the *checkX* methods in the ASM are synchronized assuring that only one thread is accessing the ASM at one time. Also, since the ASM itself may need to access system resources in determining the permissions, a flag (*InCheck*) exists to signal when a security check is in progress. (For example, the ASM may need to access the logs to determine ownership of a file.) If *InCheck* is true, then permission is unquestionably granted since the request is coming from the ASM, otherwise, the request is evaluated after setting *InCheck* to true. Once the request is determined, *InCheck* is reset to false before returning from the procedure.

As the overseer, the ASM makes sure the system performs its closing tasks before the appletviewer exits. It does this by initially setting the *java.System.runFinalizersOnExit* to true. This ensures that the loggers can write back their data to the files and all information is properly recorded upon exit.

Not all the *checkX* methods in JDK1.0.2's appletviewer's *security manager* were modified. *checkCreateClassLoader* and *checkExit* still throw security exceptions for applets since they would otherwise introduce major security hazards. Letting applets create their own class loaders would imbalance the safe foundation set by the lower level security in Java; allowing applets to halt the Java Virtual Machine seems unnecessary. In addition, *checkLink* (throws a security exception) is not modified in the current system; however, more flexibility can naturally be provided in the future.

6.2 Rules

The rules are scanned and parsed using Sun's Java Compiler Compiler (JavaCC) [28]. Given lexical and grammar specifications, JavaCC generates Java code that can parse the files. The files must conform to the grammar specified, otherwise a parsing error would be raised. Other errors including mismatched types, global redefinitions, assignments to read-only variables, illegal identifiers, negative applet categories, and so on, are caught during parsing.

In the implementation, *Past* procedures are not allowed to be nested since nested *Past* expressions can be simplified to one *Past* expression. Otherwise, it makes the implementation more difficult and makes the rule more complex than is needed. The parser signals an error if nested *Past* procedures exist.

The parser has been implemented so that it stores the rules in two tables: one for global *Assignments* and another for the global *If-Else* statements. The subcomponents of the rules are stored in their corresponding data structures as shown in Figure 17. The data structures form a hierarchy based on their types. Namely, all the **boolean** typed structures are subclassed from a common boolean expression structure (**BoolExpStruct**), and

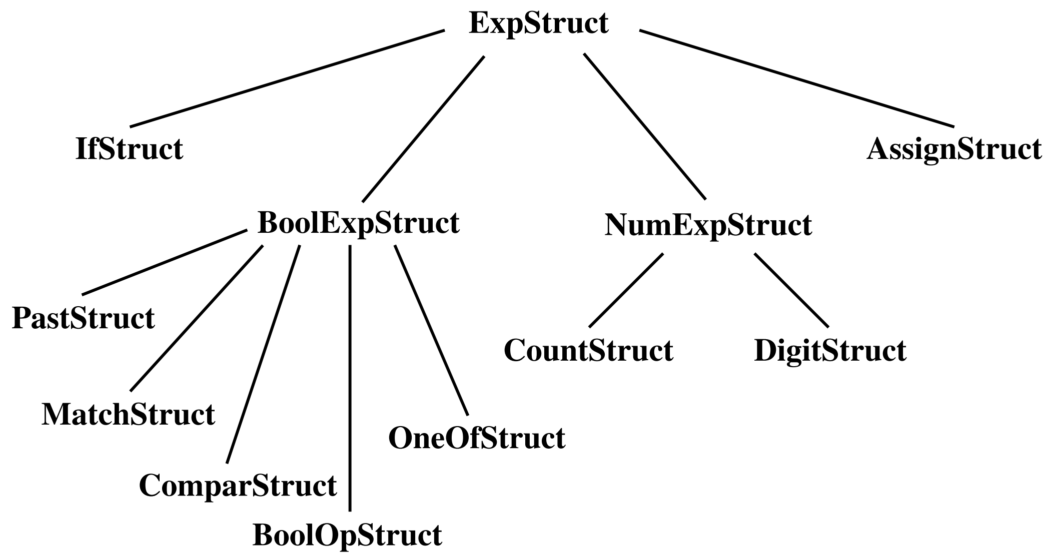

Figure 17. Hierarchy of Data Structures for Rule Subcomponents

the **digit** typed structures are subclassed from a common numerical expression structure (**NumExpStruct**).

6.3 Applet Access Checker

The Applet Access Checker (AAC) needs to evaluate the rules during runtime . It makes extensive use of Java's runtime type checking feature in order to do this . Since *Define* Statements are copy propagated, only *If-Else* statements and *Assignments* are evaluated during execution. These statements are recognized during runtime by verifying the type of the data structures: **IfStruct** for If-*Else* Statements and **AssignStruct** for *Assignments* . As the rules are decomposed for evaluation, various ***evalX*** methods are called to evaluate the objects:

evalIf (**IfStruct** ***) :*** *Vector<AssignStruct>*
evalAssign (**AssignStruct** ***) :*** *AssignStruct*
evalBool (**BoolExpStruct** ***) :*** *Boolean* or *boolean[]*
evalNum (**NumExpStruct** ***) :*** *int*

***evalVar (* **VarStruct** *) :* *Object*
***evalList (* **Vector<Object>** *) :* *Vector<Object>*
***evalObject (* **Object** *) :* *Object*

The evaluation of the *Past* procedure is interesting in that the resource or access under question has a list of past values that need to be evaluated instead of a single value. This list of values needs to be evaluated within the *Past* procedure's predicate. For example, in

```
(Any f in PastFiles ( Or  ( match  f.path  "/etc/*" )
                          ( >  ( count  f.write )  0 ) ) )
```

the predicate (Or ...) needs to be evaluated for the whole list of files accessed in the past. Within the predicate, the suboperations also need to be evaluated for a list and then combined as a list. In order to achieve this, ***evalBool*** is allowed to return an array of booleans (*boolean[]*) for the list of values in addition to a single *Boolean* value. In this way, both match and > return a *boolean[]* (*m[]* and *g[]*) which is used by Or. Or then evaluates the *or* operation on the corresponding values in the array [*(or m[0] g[0])*, *(or m[1] g[1])*, ...]. Finally, the array of booleans is reduced to a single *Boolean* value depending on whether the values are to be *and'*ed (for the *All* procedure) or *or'*ed (for the *Any* procedure). The suboperations within boolean operations in *Past* procedures may also return a list of values of other types (e.g. (count f.write) returns a list of *Integer*s). These lists will be maintained until a final *Boolean* value is attained.

6.4 Logs

The *Applet FileOwners Log* (AFL) and the *Applet Access Logs* (AALs) are implemented using cached hashtables (HashCache) as described in Section 4.6.1. The size of the two caches are specified as constants in the logger classes (*AppletFileOwnerLog.CACHE_SIZE* and *AppletAccessLog.CACHE_SIZE*) which can be easily modified. For now, they are arbitrarily set to size 16.

The caches use a Least Recently Used (LRU) replacement policy. The *HashCache* class is implemented as a subclass of *java.util.Hashtable*. It maintains an LRU vector which is updated whenever an object is accessed from the table. In order to do this, *HashCache* overrides the *put*, *containsKey*, *get*, *remove*, and *clear* methods from *java.util.Hashtable*. It also adds a *get_all* method that can be used if one is retrieving all the objects in the table and does not want the LRU vector to be updated.

In order to account for failure in the system, all the data in the cache is written back to the log after a certain number of events or minutes. In case the system fails or is exited abnormally, these regular writebacks will prevent major loss of information. The loggers include *WRITEBACK_EVENTS* and *WRITEBACK_TIMEOUT_MNS* constants to specify the minimum number of events between writebacks and the minimum number of minutes between writebacks, respectively. Counters are kept for both events and minutes. When a writeback occurs, both counters are reset. If the application exits normally, writebacks are also done upon the exit through their *finalize* methods.

6.5 Security Notes

In the implementation of the system, certain security issues needed to be addressed. As in Java, a safe design is only a support structure for a secure implementation.

6.5.1 Error Resolution

How does one resolve various errors? Errors include I/O errors, parser errors, applet security errors, other runtime errors (i.e. out of bounds, unknown host), and other unexpected system errors. Since this system involves the maintenance of multiple simultaneous running applets, we need to classify these errors into *fatal system errors* and *applet-specific errors*. The first class of *fatal system errors* should include those errors where the entire system should be halted, since otherwise, security violations would occur. If the system were inside a Web browser, then the browser should halt all its Java operations when encountering a fatal class error. The latter class of *applet-specific errors* should include those errors where only a particular applet's execution should be halted. These errors that affect only one applet need not and should not halt the entire system. If the entire system were halted from such an error, then that would allow an applet to affect the execution of other applets by simply causing those errors (denial of service attack).

It then seems apparent that errors which affect all applets and the security of the entire system should be included in the *fatal system* class. This would include parser errors and I/O errors from the Rules file, and formatting and I/O errors from the *Applet FileOwner Log*. On the other hand, a security violation by a single applet and a formatting or I/O error in a single applet's *Applet Access Log* should not affect the entire system. Instead, these errors should be signalled for the benefit of the user.

6.5.2 Applet Identification

How does one identify applets? In the appletviewer and in Web browsers, the host of the applet's document is most commonly specified using its DNS host name. In so doing, the system simply identifies the applet by the host name of its codebase. The accesses in the *Applet Access Log* and the files in the *Applet FileOwner Log* are all associated with the DNS host name of the applet's codebase. Since a server sometimes uses multiple machines (and thus perhaps multiple IP addresses) to reduce its load, the system should not distinguish the same applet on the different machines. Identifying applets by their host name allows the system to associate the applet with the server and not the specific machine permitting the applet to later access its files.

However, this leaves room for DNS spoofing attacks, in which incorrect entries in a DNS server lead to incorrect identification of applets. [7] If the DNS server becomes infiltrated since the last execution, then the correct identity of an applet changes. This introduces a vulnerability to an external source: the DNS server.

On the other hand, the support for digitally signing applets can address this. Instead of IP addresses or DNS host names, the signature on the applets would provide a secure mechanism for identification. This functionality has now been included in JDK1.1's *java.security* package. However, one may want to consider an applet that is updated to a newer version to be considered as the same identity as its older version so that it may access its old files. This would require a slight variant to digital signing where the identity of the applet does not change with some modification to its code if the author and the origin are the same. But as the theorists may know, this is currently a difficult problem.

6.5.3 CheckX methods

The *checkX* methods in the *Applet Security Manager* are called when system classes want to verify whether the current thread (applet) has the authority to access a certain resource. These methods are provided to check access. However, in calling a *checkX* method, it does not necessarily mean that the given applet has performed that action, but only that the action was requested. Despite this, the current implementation logs it as if the action was performed. For example, the *java.io.File.canRead* method uses the *checkRead* method to just check whether the read permission should be allowed, although the file may not even be read.

Current Java development kits also give applets access to the *Security Manager*, enabling applets to inquire about their permissions using the *Security Manager's checkX* methods. This feature allows applet authors to write more robust and useful code. However, with the current implementation of the system, the *checkX* methods will log these inquiries as actual accesses.

The outcome is that applet's accesses would be limited by these extra loggings if the applet policy includes rules that limit future accesses based on past ones. For example, if a rules states that "an applet cannot access more than 8 files", the applet would be able to access only 4 files since the checks prior to each access would also be counted. This does not introduce any security holes assuming past accesses only **limit** future accesses. If this were not the case, then an applet could merely call the *checkX* methods without actually accessing the resource but instead extending its permissions. An example would be a rule that states, "an applet can access this protected file only if it has read this copyright." The applet could simply call the *checkRead* method on the copyright, but not read it. However, such rules would not be very useful.

To address this properly however, the *Security Manager* should have two types of methods: one for checking access (*checkX*) and another for both checking and logging the access (*checklogX*). The former can be used by applets who want to know their permissions, and the latter can continue to be used by the system classes.

6.5.4 Unix File System

To be consistent with the capabilities on the UNIX platform, giving write access to an applet does not necessarily mean it has read access. Writing is not a superset of reading. In order to allow an applet to read and write, both permissions must be assigned to true in the rule.

On another note, the current Java API is limited in that the file permissions on the UNIX system are inaccessible. Therefore, there is no way of setting a file's ACLs or discovering whether a file's **SUID** bit is set. This limits the breadth of the policy one can place on an applet's access to the file system.

Further, the files that are newly created by the Java runtime are given ACL permissions based on the user's current *umask* value. If the *umask* value does not prevent the creation of world-readable files, then all new files would be world-readable. The file permissions cannot be changed after its creation because of the limitation in the Java API. Consequently, the log files created by the system and files created by applets would be dependent upon the *umask* value. Users must be aware of this and set their *umasks* values accordingly.

6.6 Summary

The implementation of the system addressed issues of efficiency, concurrency, and security. Algorithms have been discovered for the first two as computer science progressed. However, the last, security, still needs to be carefully considered. The current implementation has attempted to address, in a practical manner, the security issues that have so far arisen. But more has yet to be done.

Chapter 7

Conclusion

The *System* has been designed and implemented to propose a solution for providing fine-grained control over Java applets while ensuring the requirements set in Section 1.4: configurability, flexibility, straightforwardness, and security. In doing this, a *Simple Constraint Language* was created to be used for writing a **configurable** applet policy. The language includes nomenclature for variables that identify resources, accesses, and applets. The syntax of these variables was intentionally chosen to be hierarchical in order to provide a **flexible** naming scheme that can be easily extended. Primitive procedures and structural constructs are also supplied to ease specification of constraints and rules.

The rules are powerful in that they can reference an applet's past accesses in order to determine its future accesses. This goes a step beyond the commonly used capabilities and access control lists found in current operating systems. Further, a labelling scheme is included for grouping like applets into a single category and for writing rules based on the category. The labelling can also be based on the applets past accesses in addition to their origin and identity. This makes writing policies even more **straightforward**.

In designing and implementing the *System*, the **security** of the client system was always kept in mind. Therefore, measures were taken to prevent or track applet communication, to limit resource usage, to audit applet behavior, and to maintain privacy, integrity, and reliability of the system.

All this was implemented without the need for modifying Java's core system classes. However, in order to build a commercial application of the *System*, the Java API needs to be extended to include more UNIX file system specific control, resource usage control, and applet permission verification. These additions would then more confidently assure that the security of the UNIX system is preserved. A sample policy for UNIX users is provided in Appendix A which addresses the security issues of the UNIX system. To be most secure, the *System* should not be run as the *super user*.

7.1 Future Direction

Further work in this area can prove to yield an even less restrictive and even more secure, straightforward, configurable, and flexible system.

7.1.1 Simple, but Smarter, Constraint Language

The language should be extended to include the possibility of identifying applets using their digital signatures. This will allow users to authenticate applets and to place more trust based upon their origin. However, to create a sound policy, the rules should still be combined with those that give authorization based upon the applets' trustworthiness and past actions. Policies based solely on the origin of applets would otherwise create a rigid caste system.

In addition, a more intelligent policy reader could perhaps be developed to catch obvious errors in the rules. If artificial intelligence, specifically an expert system on security policies, were included, the policy reader might be able to statically determine the security level of the policy. For example, if a rule grants write permission to certain files, but no read permission, then that may be an error on the part of the user which can be easily detected by an intelligent policy reader. Or if the policy reader has knowledge of the vocabulary used for security, it can detect a possible error in a rule that allows "suspicious" applets to connect to the network. Such intelligence could prevent security blunders made in the policy.

A feature that is currently lacking in the language is the ability to define procedures. Such a capability would make the policy more powerful and the language even more flexible. For example, currently in the language, there is no way for the user to check if the name of the file under question matches one of the files in the *bin* directory (to catch Trojan horses more effectively). A primitive procedure that lists the contents of a directory or an additional attribute to the directory resource variable can be added to the language. However, that would require modification of the System. Instead, if the user could develop his own procedure to list the contents of the directory, then that would be more flexible allowing even further extensions. A possible mechanism of easily implementing this is to allow users to write their procedures in Java using Java's API. Their procedures would simply then be executed during runtime. Care must be taken, however, to make these procedures bug-free and to test them fully since the applets' policy would be based upon their return values. Keeping them short and simple would aid in this.

7.1.2 Applet Durability

Applets are associated with their codebase in the *System*. An applet can access its past files only if its codebase matches the codebase associated with the files. However, an outcome of this practice is that if the applet's location is modified, then it can no longer access its past files on the user's system. The usage of Uniform Resource Names (URNs) however, would easily remedy this. If applets were associated with their URNs instead of their location, the mobility of the applets would not affect their ability to access their files. [26]

7.1.3 File Sharing

The *System*'s goal of providing more freedom to applets was however limited by the restraint of not allowing applets to share files. This confinement was placed in order to retain security by preventing inter-applet communication. However, from the applets' point of view, file sharing would be a useful capability in creating cooperative applets that work together in performing powerful operations. For example, as one applet organizes and arranges the user's schedule, another could graphically present the scheduler, while another communicates with other agents to make appointments. These teamplayer applets would perhaps need to communicate with each other and to share the common schedule files.

A possible way of implementing file sharing while ensuring security, would be to attach a security label with each file and with each applet. The label on the file would denote the security level of its contents. i.e. Secret-from-All-Applets, Public-to-Network, Non-Network, etc. The label on the applet would correspond to the highest level of security of the information that it had accessed so far. Then applets with the same current security level should be able to access the same files without compromising the local system. This way, in case applet communication occurs through the shared files, it would not matter since both applets have accrued the same security level of information. This assumes that the policy set by the user is in accord with these security labels. A simple and straightforward way of implementing such rules is necessary.

7.1.4 Centralized Policy Decisions

The *System* provides a way for ordinary users to specify a fine-grained control policy. However, some users may still not be capable of or have the time to set the rules. To address this, default policies for different operating systems should be supplied for these users. If they wish, they may modify the rules. But the default policy should be comprehensive and easily configurable to set appropriate values (i.e. user's home directory, etc.) to those of the user's system.

Network administrators, on the other hand, may want to place a centralized policy for all system users. This can be done by configuring the location of the *Applet Rules File* (ARF) to a directory that is writable only by the *super user*. Users can, however, still download their own version of the *System* and configure their ARFs differently ignoring the administrator's policy.

Corporations also need to place certain common limitations on applets downloaded by all their employees. This is usually done by using a firewall. A policy can be placed at the firewall which must be obeyed by all applets entering the site. In order to implement this, the *System* would need to be integrated into the firewall filtering mechanism. The filtering mechanism would take the place of the *Applet Security Manager*. Further, with this scheme, users inside the firewall can create their own policy but it cannot override the site's policy. In other words, the accesses granted to an applet by the user can only be a subset of the accesses granted by the site, not a superset.

7.1.5 Other Languages

Can the *System* be modified to work with languages other than Java? The security of the *System* stems from the secure foundation of Java's low level architecture, its safe language, and its interpretative feature. The safety of the language is needed to ensure the applet's program does what it says. The secure architecture is needed to ensure the applet is obedient. The interpretative language is needed to check and restrict its behavior during run-time. Another language that can provide these three things would be suitable.

7.2 Final Note

The *System* has potential for a less restrictive policy on applets, while providing a secure, straightforward, configurable, and flexible system. Integrating with UNIX-specific Java API, file sharing, centralized policy decisions, digital signatures, and URNs would make it even more powerful.

Appendix A: Sample Policy for UNIX Users

```
/************ UNIX Specific Rules ************/
/** Catch Trojan Horses **/
( If  ( OneOf  File.Name  ( “ls” “find” “more” “diff” “ps” “cat” “chmod” “chown” “chgrp”
                            “cp” “mv” “grep” “sh” “csh” “ksh” “man” “ln” “su” )  )
          ( File.Write = false ) )

/** Protect the system directories **/
( If  ( OneOf  File.Path  ( “/bin/*” “/etc/*” “/usr/*” “/var/*” “/dev/*” “/adm/*” “/sys/*” ) )
          ( File.Write = false )
          ( File.Delete = false ) )

/** Disallow the creation and modification of dotfiles ( includes .rhosts files ) **/
( If  ( Match File.Name “.*” )
          ( File.Write = false )
          ( File.Delete = false ) )

/** Disallow the execution of protected progams. **/
( If  ( OneOf Command.Name ( “ps” “who” “w” “lastlog” “lastcomm” ) )
          ( Command.Exec = false ) )

/************ Prevention of Denial of Service ************/
/* Limit number of file pointers created. */
(If  ( > ( Countall File.Write ) 20 )
          ( File.Write = false ) )

/* Limit usage of total file space. */
( If ( > ( Countall File.Size ) 500000 )
          ( File.Write = false ) )

/* Limit number of windows created. */
( If ( > ( Countall Window.Create ) 100 )
          ( Window.Create = false ) )

/************ Additional Personal Rules ************/
( Define WriteableDirectories ( “/user/nimisha/AppletTempSpace/*” ) )
( If ( OneOf File.Path WriteableDirectories )
          ( File.Read = true )
          ( File.Write = true )
          ( File.Delete = true ) )

/* Allow trusted sources access to protected directories and network connection. */
( Define ProtectedDirectories ( “/user/nimisha/Mail/*” “/user/nimisha/Papers/*” “/user/nimisha/Diary/*” ) )
( Define Trusted Sources ( “web.mit.edu” “www.sun.com” “ana.lcs.mit.edu” ) )
( If ( And ( OneOf File.Path ProtectedDirectories )
        ( OneOf Applet.CodeBase.Host.Name TrustedSources ) )
          ( File.Read = true )
          ( File.Write = true )
          ( Host.Connect.To = true ) )

/* Keep protected information inside. */
( If ( Any f in PastFile ( OneOf f.path ProtectedDirectories ) )
          ( Host.Connect.To = false ) )
```

References

[1] E. M. Bacic, *UNIX^{TM} & Security*, 24th Annual DECUS Canada Symposium, Canada, February 1991.

[2] W. Belgers, *UNIX Password Security*, December 6, 1993. http://www.het.brown.edu/guide/UNIX-password-security.txt.

[3] Business Week*, Business Week/Harris Poll: A Census in Cyberspace,* May 5th, 1997. http://www.businessweek.com/1997/18/970505.htm.

[4] F. J. Cooper, et al. *Implementing Internet Security*, New Riders Publishing, Indianapolis, Indiana, 1995.

[5] D. Curry, *Improving the Security of Your UNIX System*, Final Report, SRI International, CA, April 1990.

[6] Cylink Corporation, *Digital Signatures and Certificates*, 1997. http://www.cylink.com/products/security/digsig/.

[7] D. Dean, E. W. Felten, D. S. Wallach, *Java Security: From HotJava to Netscape and Beyond*, IEEE Symposium on Security and Privacy, Oakland, CA, May 6-8, 1996.

[8] D. Farmer, W. Venema. *Improving the Security of Your Site by Breaking Into It*, 1993. ftp://ftp.win.tue.nl/pub/security/admin-guide-to-cracking.Z.

[9] J. S. Fritzinger, M. Mueller. *Java Security*, Sun Microsystems, Inc, 1996. http://java.sun.com/security/whitepaper.ps.

[10] S. Garfinkel, *Security Article Extracts Legalities*, 1987. http://www.cs.purdue.edu/coast/archive/.

[11] F.T. Grampp, R. H. Morris. *UNIX Operating System Security*, AT&T Bell Labs Technical Journal, Vol. 63, No. 8, October 1984.

[12] M. Gray. *Web Growth Summary*, 1996. http://www.mit.edu:8001/people/mkgray/net/web-growth-summary.html.

[13] C. Hare, et al. *Inside UNIX*, Second Edition, New Riders Publishing, Indianapolis, Indiana, 1996.

[14] T. Jaeger, A. D. Rubin, A. Prakash, *Building Systems that Flexibly Control Downloaded Executable Content*, Proceedings of the 6th USENIX Secuirty Symposium, San Jose, CA, July 1996, pp 131-148.

[15] JavaSoft, Sun Microsystems, Inc., *Frequently Asked Questions - Applet Security,* April 1997. http://www.javasoft.com:/sfaq/index.html.

[16] JavaSoft, Sun Microsystems, Inc., *HotJava: The Security Story*, May 1995. http://www.javasoft.com:/sfaq/may95/security.html.

[17] C. Kehoe and J. Pitkow, *GVU's 6th WWW User Survey,* Georgia Tech Research Corporation, 1996. http://www.cc.gatech.edu/gvu/user_surveys/survey-10-1996/.

[18] M. McKusick, et al. *A Fast File System for UNIX*, ACM Transactions on Computer Systems, Vol 2, No 3, August 1984, pp 181-197.

[19] Microsoft Corporation, *Microsoft Security Advisor*, April 1997. http://www .microsoft.com/security/ .

[20] G.Necula, P. Lee. *Proof-Carrying Code*, Technical Report, CMU-CS-96-165, November 1996. http://www.cs.cmu.edu/~necula/papers.hml.

[21] Netscape Communications Corporation, *Netscape Security Solutions*, 1997. http://www.netscape.com/assist/security/index.html.

[22] P. Niemeyer and J. Peck, **Exploring Java**, O'Reilly & Associates, Inc., May 1996.

[23] The Open Group Research Institute, *Java Mobile Code: An OSF White Paper*, January 1996. http://www.gr.osf.org/java/papers/whit_pap.htm.

[24] O'Reilly & Associates/Trish Information Services*, Defining the Internet Opportunity: Internet Usage*, 1995. http://www.ora.com/research/users/index.html.

[25] D. M. Ritchie, *On the Security of UNIX*, UNIX Seventh Edition Manual, Volume 2, Bell Telephone Laboratories, 1979.

[26] K.R. Sollins, *Functional Requirements for Uniform Resource Names*, Network Working Group RFC 1737, February 1995. ftp://ds.internic.net/rfc/1737.txt.

[27] Sun Micosystems, Inc., HotJava Browser 1.0 Users Guide. doc:/UsersGuide/applet_security.html.

[28] Sun Microsystems, Inc., Java Compiler Compiler, Version 0.6(Beta), 1997. http://www.suntest.com/JavaCC/.

[29] Sun Microsystems, Inc., *Java Platform 1.1.1 Documentation*, 1997. http://www.javasoft.come/prodcuts/jdk/1.1/docs/.

[30] Sun Microsystems, *Secure Computing with Java: Now and the Future*, April 28, 1997. http://java.sun.com/marketing/collateral/security.html.

[31] H. van Viet, Mocha Decompiler, Version 1(Beta), 1996. http://web.inter.nl.net/users/H.P.van.Vliet/mocha.html

[32] The World Wide Web Consortium, *Digital Signature Initiative,* August 1996. http://www.w3.org/pub/WWW/Security/DSig/.

[33] The World Wide Web Consortium, *Platform for Internet Content Selection,* September 1995. http://www.w3.org/pub/WWW/PICS/.

[34] F. Yellin, *Low Level Security in Java*, WWW4 Conference, JavaSoft, December, 1995. http://www.javasoft.com:/sfaq/verifier.html.