Automated Verification of Mobile Code

by

H. Dan Lambright

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

UNIVERSITY OF ARIZONA

November 3, 1997

Abstract

In this thesis, we introduce a new technique to automate the verification of mobile code. Using dataflow analysis techniques, the verifier can check whether data passed between trusted software components is illegally modified by untrusted mobile code. We show that this analysis is powerful enough to make significant guarantees about whether the program will access system resources safely. Furthermore, by rendering the verification transparent to the user, the security system is not vulnerable to human error, or dependent on the user's technical abilities. Other verification techniques do not share these advantages. We describe what requirements enable this analysis, explore its limitations, and present prototype software that implements the idea.

Contents

Chapter 1

Introduction: Mobile Code and Security

1.1 Motivation

In recent years, the growth of the Internet and the World Wide Web has enabled new ways of distributing computer programs. This software, *mobile code*, is specially designed such that it is closely coupled with computer networks. For example, because a great variety of machines are connected to the Internet, some mobile code systems are *architecture neutral*, i.e. they are executable on any platform.

Mobile code proponents envision a plethora of exciting new applications: users will be able to download anything from real-money casino games to customized tax preparation programs. Large-scale distributed computing may soon be possible, in which mobile code is embedded in everything from set-top-boxes to PDAs [36] [20]. Ultimately, many companies are banking that mobile code will make practical the long awaited *network computer* [4], thereby driving down the cost of maintaining a PC.

In reality, mobile code technologies have been widely deployed in Internet browsers for several years. But despite their promising applications and availability, they have not yet become widely employed by users. This slow acceptance is due to a variety of problems, notably lackluster performance and incomplete standardization. The major problem with mobile code, however, is its security risks.

Each time a user executes software that has originated from somewhere on the Internet, he is betting that the foreign code is not *malicious*. Code is considered malicious if it has an undesired side effect. Examples include software which irrevocably damages the computer, makes public private information, or renders the computer inoperable. Whether the software does its damage on purpose or as a result of an unforeseen bug is irrelevant: either way the outcome is undesirable and must be prevented. For most users, these risks are too great to run mobile code.

Arguably, a user is taking the same risks in buying shrinkwrapped software from a store as he does by downloading it off of the Internet. From the user's perspective, however, storebought software is safer: the manufacturer is clearly identified on the box's label, and he is given a tangible receipt by a salesperson. In contrast, on the Internet the customer is not given any such psychological or tangible reassurances. No human salesperson is on hand to ease the user's fears, and no box is unwrapped at home containing a warranty card. A purchase is made by simply clicking on a webpage link.

Proponents of Internet commerce might point out that a user could acquire the software off of a well known vendor's website, and be assured of the code's safety by the company's good reputation. Unfortunately, that is still too risky: how does one know that the site actually is the vendor's site it claims to be? On the Internet, there is the potential for a server to pretend to be something it is not, and trick the client [8]. In a recent incident, Microsoft corporation took legal action to prevent a website called `www.ms.com` from operating. The operators of the site were taking advantage of the fact that many users would access their webpages thinking it was the real Microsoft site.

To protect client machines from malicious mobile code, designers are devising *security policies*. These are strategies that allow the user to download code safely by ensuring that the user is completely protected from undesired side effects should the code prove to be malicious. Security policies work by *verifying* the downloaded software. The purpose of the verifier is to check that the code adheres to the security policy. It uses some strategy or algorithm to prove that the code is not malicious before it is allowed to execute at the client. Once the verifier signals that the code is safe, the client can grant the code some degree of access to its *system resources*, i.e. the client's disk, graphics, network, and core memory.

The purpose of this thesis is to confront the security problems which exist in the Java language. Java is one of the most widely used mobile code languages, and its security problems are a timely topic of debate. We introduce a new security policy, *dataflow verification*, which has significant advantages over policies that are either currently used by Java, or that have been proposed for it. In a nutshell, these advantages are

1. It fills a significant hole in Java's security policy: the ability of mobile code to safely access system resources on the user's computer, such as the disk.

2. It minimizes the amount of responsibility and technical intelligence required by the end user who runs the mobile code. In doing so, it retains the simplicity of the Internet and keeps the promise of mobile code available to the largest number of people.

1.2 Definitions

Throughout this thesis the following definitions are used.

Applet: An applet is a program that is mobile (transferable across computer networks) and also architecture neutral.

Client: The client is the machine which runs the downloaded applet.

User: The user is the human party that controls the client.

Server: The server is the machine that provided the applet to the client.

Developer: The developer is the human party who wrote the applet.

Browser: In this thesis, a browser is a computer program run by the user that both accesses the World Wide Web *and* downloads and executes applets.

System Resources: A system resource is some component of the client which is protected by the verifier. These components are allocated a finite quanta of the computer's resources. The resouces may be time (CPU cycles) or memory. Example system resouces include process ids and the disk drive.

Safe: A body of code is considered to be safe if it is known by the user to not have any unexpected or undesired effects on the client when it is executed.

1.3 Conventional Security Policies

Despite the fact that Java was designed with security concerns in mind, its default policy has proven to be inadequate. While it does prevent applets from corrupting the client's volatile memory, it does not have a safe way to access the rest of the client's system resources, such as the disk and network. Currently, Java users must either grant the downloaded applet unrestricted access to the disk or no access at all. The major problems with this is that the burden of deciding whether to grant access to system resources falls to the user, who will be forced to develop some criteria to make the judgment. If he makes the wrong decision, he may put his computer in jeopardy. Most users are unwilling to take on that responsibility.

Because of this weakness, Java applets have thus far largely been limited to simple animations that do not require access to system resources, such as newstickers and arcade games. These programs are a far cry from the mission critical applications mobile code proponents envision.

Many researchers in industry and academia are working hard to improve Java's current state of affairs. Broadly, three solutions are being considered. They may be characterized as:

Digital Signatures. In this scenario, the user's assurance in running the code derives from the proof that it was obtained from a trusted vendor. A *digital signature* is built by encrypting the software. Only the vendor is capable of making the signature, but anyone can recognize it.

Runtime Monitors. In this strategy, the user has fine grained control over access to system resources. The user customizes the access granted to the applet based on some criteria, such as the applet's identity. For example, a user may configure his runtime monitor to allow only Microsoft applets access to the local disk.

Formal Verification. A formal verifier shows that the code is not going to harm system resources by using provable, mathematical logic.

We will show that each of these solutions is flawed: digital signatures are vulnerable to human errors, runtime monitors require too much technical ability by the end user, and formal verification has not been shown to be possible. A better approach must be found.

1.4 Dataflow Verification

This thesis introduces a new security architecture which transparently checks whether downloaded applets are safe or not, without user intervention. The advantages to performing the verification transparently are that no technical abilities are required of the user, and the verifier is protected from human error.

The verification works by *specifying* how critical API routines (those that access system resources) may be invoked. The specification explicitly defines what parameters may be used for those routines. This specification is provided to applet developers; any developer who wishes to make his code verifiable must follow the specification.

When a user downloads an applet, the verifier performs dataflow analysis on the parameters sent to critical API routines. This is done to check compliance with the specification. If all of the API routines have been invoked in a way that agrees with the specification, the code will be deemed safe, and the user can be sure that the code is not malicious. Otherwise, the applet will be rejected.

Thus, the responsibility of building safe code rests on the developer's shoulders, and the client-side verifier automatically tests whether those responsibilities have been met.[1] The end-user is not involved in the process. Even if the developer made a mistake in coding the applet such that it could adversely

[1] We assume the verifier is bug free.

affect the client, this will be caught by the verifier because only safe programs are allowed in the specification.

1.5 Thesis Organization

This thesis will first examine the security issues with mobile code in general (chapter 2). We will look at the security policies proposed and examine their deficiencies. Then, in chapter 3 we will look at how Java protects the user from malicious code, and where it comes up short. We will then introduce *dataflow verification* in chapter 4, and show its advantages over the proposed policies. Chapter 5 shows the details of a prototype verifier which implements a working dataflow verifier. Chapter 6 will look at security policies implemented in other mobile code systems. Finally, chapter 7 concludes with a brief discussion of future work.

Chapter 2

Secure Mobile Code

This chapter summarizes the threats posed by mobile code and the current strategies employed to counteract them. We will first describe the *threat model* posed by mobile code, i.e. the methods malicious applets employ to gain unrestricted access to the client, and the damaging actions they may take. The threat model will show that capitalizing on human mistakes is the most likely route by which a malicious entity will attack. Given this background, we will then describe the design principles and overall goals of a security policy. There are a wide variety of architectural options of varying complexity. Finally, we will critically examine the current or proposed approaches being taken to verification, and discuss how well they meet the goals.

2.1 Problem: Mobile Code is Unsafe

Applets are usually loaded onto the client via a World Wide Web browser connected to the Internet. Browsers load applets onto the machine in response to a user input; they will not begin downloading applets on their own accord. This is in contrast to other forms of content delivery, such as electronic mail, which can be "pushed" onto unwitting users in the form of "junk mail."

Thus, the job of a hostile server that wishes to attack a user is to somehow trick the user into loading a malicious applet. Note that on the Web, this is simplified by a peculiarity with HTML: no visible distinction is made between a link to an ordinary webpage and a link to an applet. Because of this, a user may not realize that he has loaded an applet until after he has clicked on the link.

Three closely related attacks whose strategy is to deceive the user are:

1. The *luring attack*, which works by putting a link on a webpage that somehow encourages the user to access it. For example, the link may promise a free gift.

2. The *trojan horse* applet, which overtly performs acceptable functions while invisibly performing hostile acts. An applet that plays a game in the foreground while spawning off a thread to delete the client's files in the background would be an example.

3. In the *spoofing attack* the server providing the applet pretends to be a trusted entity that it is not. For example, if the client makes a request for an applet from a trusted company, and the malicious applet does a good job at pretending to be that company, the user may be tricked into giving the applet access to his computer.

Currently, mobile code usage is dominated by the server-web browser model. In the near future, however, mobile code may begin to be used in embedded controllers [36]. In that scenario, a server may have the ability to push malicious code onto client hardware. Today's mobile code systems must be robust enough to adapt to tomorrow's threat models in which the attacks cannot be as easily foreseen.

Malicious code that has somehow managed to penetrate the client may be further characterized by the severity of the damage that it inflicts.

The most damaging attack is one that causes irrecoverable damage on the client's machine. This may be called a *write-attack* applet. It can occur only if the applet has "write" capacity on the client's permanent storage. Examples include applets that delete files, format hard drives, rename files, or create directories. Worse, a write attack applet that finds a way to write data to unauthorized memory may be able to change attributes in a process control block, or even affect periphial devices, if they are linked to the client by memory mapped I/O.

Malicious applets that only have read access to the client's permanent storage fall under a slightly less severe category. They can be called *read-attack* applets. The damage they inflict is not physical or permanent, but may be just as serious because they could make public highly sensitive information, such as a password file or a list of credit card numbers.

The weakest category of malicious applets has neither read nor write access to system resources. They cause no permanent damage and cannot leak private information. *Hostile* applets deliberately abuse resources. This can be done by aquiring an inordinate percentage of CPU cycles, thereby denying the user access to the computer. Creating too many threads, using up all of the system's available file descriptors, or opening too many windows, are all examples of hostile applets [15].

In the worst case a hostile applet is written in a way such that the only recourse the user has is to reboot his machine. Hostile applets come in enormous varieties and are extremely difficult to detect and prevent [15].

What follows is a taxonomy of the principal components of a computer system that must be protected.

Disk The client's permanent storage must be protected. The user may allow some subset of his storage, such as a directory, to be accessible to the applet.

Network An applet must not be allowed to open unauthorized network connections. This is to prevent private data from being leaked to untrusted hosts. Authorized network connections should be allowed for such applets as teleconferencing applications or multi-user games.

Graphics Engine Clearly, some access by the applet to the graphics engine is desirable in order to generate an effective user interface. However, giving an applet free reign over every aspect of graphics may introduce problems. For example, keyboard events could be monitored while the user is typing in a password, or a mouse could be manipulated into dragging a file icon into the delete bin.

CPU and RAM Ideally, an upper bound is given to the size of the run-time environment granted to an applet (size of volatile memory, and the number of CPU cycles). Such real-time limitations may be difficult to implement in practice.

2.2 Solution: Design a Security Policy

Once the threat model is well understood, a security policy can be formulated. For the purposes of this thesis, the security policy is adequate if it can faithfully detect and prevent read and write attack applets from damaging the client. Code accepted by such a verifier is considered safe. Note that by this definition, the security policy may not be able to detect hostile applets. Hostile applets are beyond the scope of this work.

We assume the client's execution environment, including the virtual machine, API functions, and verifier, are completely free of implementation errors. If the security policy has an implementation bug, the entire system may become vulnerable to attack [3] [8].

In the next subsections we will lay down the design goals of the security in greater detail, and broadly discuss the architectural options.

2.2.1 Goals

Several sets of criteria have been developed to evaluate and compare security policies [38] [24]. This thesis will focus on the following subset of the criteria developed in those papers.

1. The verifier must have *fail-safe defaults*. This means that the default level of protection on the system gives the applet no access to system resources. Any level of access must be explicitly granted by the user.

   By fail-safe, we mean there is no way for a malicious applet to circumvent the verifier. Even a single flaw in the verifier might be enough to compromise the integrity of the entire system [15].

2. The verifier should be *psychologically acceptable* to users accustomed to the simplicity of the World Wide Web. Ideally, the verifier should be a "black box" which accepts an input applet from the user, and outputs whether or not the applet is safe or not. Maintenance and day-to-day usage of the verifier should incur no burdens on the user. For example, it would be unacceptable for the verifier to query the user in a dialog box to ask for permission each time an access was made to a system resource. The user may grow weary of answering those queries.

   The user should retain complete freedom in exploring the Internet. A security policy should in no way rely on how "street savvy" a user is on the Internet. Educating users about "safe ways" to use the Internet, and telling them to be "careful what websites they browse" is insufficient. The paradigm of World Wide Web is that it is a limitless labyrinth of hypertext on which users can browse anywhere they like. This is the liberal model users have come to expect and it cannot be changed.

   Finally, the system's performance should not be impacted negatively by the verifier to the extent that it hurts the user's ability to perform work.

3. The verifier should be *compatible* with the target language's current applications and run-time system (e.g. the browser). While it is usually possible to rebuild certain components of the run-time system to make them more secure, doing so could confuse users who would be confronted with multiple versions (e.g. both the legacy and rebuilt systems).

4. The verifier should be *extensible*. Security policies vary application to application. A flexible security policy allows the user to incrementally grant access to portions of the client's system resources to the applet. For example, the user may desire to give different system access to a game program than to a tax preparation program.

2.2.2 Design Spectrum

Once the goals of a security policy have been defined, some very basic architectural issues must be addressed that decide *where* and *when* the verification will be performed. These decisions will impact the policy's performance, robustness, and maintainability.

A verifier could be configured either on the client (see Figure 2.1.b and 2.1.c) or on a third machine connected to the client over the network (see Figure 2.1.a.) The most common model is to perform

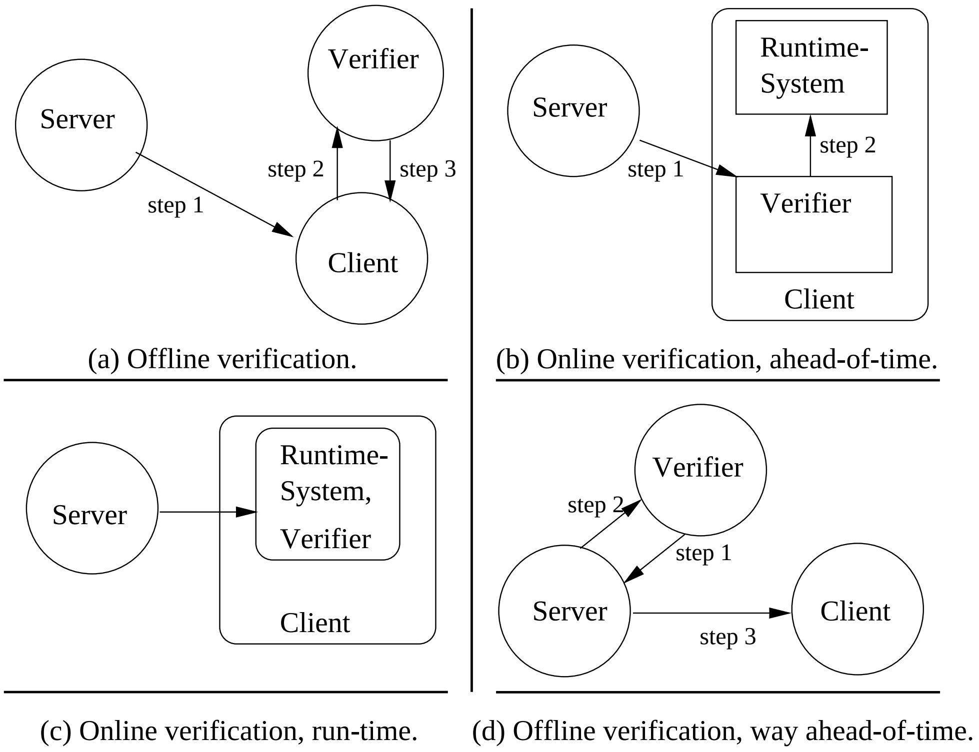


Figure 2.1: Different Architectures of the Verifier.

the verification at the client. However, researchers are investigating offline verification, such as at the University of Washington's Kimera project [5]. The advantages are that it centralizes the information needed to perform verification into one place. For example, the third machine may contain a master list of applets which are known to be safe. Additionally, it reduces the size of the code needed to be maintained at the client. The verification code could be very large and ensuring that all clients in an organization are using the most up-to-date versions can be cumbersome. Reducing the client's code size is also important for embedded applications because of their limited amount of memory.

On the other hand, there are several disadvantages to moving the verification away from the client. One is that it creates new opportunities for spoofing attacks. The client must be certain that the decision made by the offline verifier reaches the client untampered over the network. While the mechanics of doing so are well understood [27], it introduces another degree of complication. Secondly, latency is introduced. There is additional overhead incurred by the extra exchanges over the network between the client and verifier. Lastly, a centralized verifier is a single point of failure: if either it or the network connected to it become inoperable, no applets can be verified.

If the verification takes place at the client, another design decision must be made: should the applet be verified while it is being interpreted (at *run-time*), or before it is run (*ahead-of-time*). The advantage to run-time verification is that there is no intermediate verification stage between downloading the applet and executing it. However, run-time execution may necessitate modifying the virtual machine (VM), so that it checks the validity of instructions immediately before they are executed. Users who use unmodified VMs would not be able to detect malicious applets.

The final option is to perform the verification *way ahead-of-time* (see figure 2.1.d). In this case the server has the verification performed *before* it sends the applet to the client. The verification is done at some location that is known and trusted by the client, and proof of the verification is given to the client by the server. The advantage to this scheme is that it removes the verification latency. Only one transaction between the client and the server is required to download the applet. The disadvantage to way ahead-of-time verification is that the verifier that was used by the server may not be recognized by the client.

An interesting possibility enabled by offline verification is that a fee may be charged to perform the work. In this scenario, clients would send applets to a verification agency, called an *underwriter*. For a price, the underwriter would then tell the client whether the applet is safe to execute or not.

2.3 Approaches to Protecting Memory

Protecting memory is a well understood problem. The goal is to prevent the code from making unauthorized accesses to memory outside of the domain of the applet's data section.

One way this can be done is by using a language that is *type safe*. Such a language prevents variables from being assigned to each other in ways that are illegal. For example, an arbitrary integer cannot be stored into a pointer, and the index of a cell accessed in an array must be within the bounds defined when the array was created. The Spin project at the University of Washington relies in part on Modula-3's type safety for security [26] [6]. Java's basic security also depends on type safety; this will be further discussed in chapter 3.

Type safe languages may be unfamiliar to system programmers used to C, or incompatible with existing software systems written in C. For situations where a low level language such as C or assembly is used, a technique called *Software fault isolation* [37] (SFI) can be employed. SFI is the process of inserting checks around memory accesses in the code. These checks ensure that the accesses to memory are safe. The disadvantage to SFI is that the checks incur a performance penalty [34].

Finally, *virtual memory address domains* can also be used to protect memory [1]. In this strategy, the address space visible to programs is controlled by a protected kernel-level memory manager. Changing the address space can only be done via protected system calls. The disadvantage to this idea is that the system is not as portable; only machines the apropriate kernel-level support can implement it. SFI and type-safety do not require any privileged operations.

2.4 Approaches to Protecting System Resources

The following subsections will explore the different techniques that have been developed to protect a client's system resources (other then memory) from malicious applets. The subjects of the first two sections, reference monitors and digital signatures, are well understood and have been deployed for commerical use in mobile code systems. The third section, formal verification, is being actively researched.

2.4.1 Reference Monitor

A reference monitor is a verifier which checks the validity of accesses to system resources made by the code while it is running. Recently, Sun has indicated that they will incorporate reference monitors into Java's security policy, and Netscape has announced that version 4.0 of their browser will use the technique.

In the language used in the security literature, the *principal* is the entity making the request, and the *target* is the system resource that the principal desires to access [32]. When an applet accesses a system resource, a reference monitor receives a request from a principal for a target. To determine

```
COM.javasoft file read path
COM.javasoft net connect remote_IP:port
COM.javasoft awt maketoplevelwindow number
```

Figure 2.2: Example Configuration File.

whether the request may be accepted or not, the monitor must refer to a set of criteria. Typically, this criteria is represented within an *access matrix* [17] which binds principals to targets.

In the reference monitor model, there must exist a mechanism by which users can maintain and update access matrixes. Sun microsystems accomplishes this using a *configuration file*, as in Figure 2.2. In both Sun and Netscape's implementations the principal is the network address of the server from which the invoking class was downloaded. Thus, the configuration file binds network addresses to system resources.

A problem with this solution is that maintaining the configuration file is a burden on the user. If the user wishes to run an applet that performs disk accesses, and that applet is not recorded in the configuration file, the user will have to manually insert the applet into the configuration himself. That operation may be beyond the abilities of many users. Maintenance of a configuration file may be acceptable to large corporations with support staffs, but smaller corporations may balk at having to keep track of yet another level of infrastructure in their computer systems.

Version 4.0 of Netscape's browser will also implement a reference monitor. In order to support run-time monitoring, Netscape extensively modified its virtual machine [38]. In the new VM, the call stack is *annotated* with the the invoking class's privileges. For example, suppose the user configured the monitor such that IBM applets had disk access. When a method in an IBM applet was entered, the VM would record the disk privileges onto the callstack. Whenever the reference monitor detects that an access has been made from a principal to a target, it searches back in the callstack for an annotation giving privileges to the target. If an IBM method attempted to access the network, the VM would check the callstack, and see the no network access had been given.

The problem with Netscape's solution is that it is unlikely that the changes that the company made to its virtual machine will be adapted by its rivals, such as Microsoft. Therefore, this security policy will work only with Netscape's products. This lack of interoperability violates the spirit of Java, which was devised to work universally across different computers. It will also confuse enterprises which use both Internet Explorer (Microsoft's browser) and Netscape. A malicious applet rejected by Netscape may be undetected by Internet Explorer.

Another complication is that code must be prevented from spoofing trusted principals. This means that the principal has to bear proof that it is who it claims to be. This problem is discussed further in the next section.

2.4.2 Digital Signatures

A *public key encryption* (PKE) scheme allows a server to digitally encrypt information using a "private key." The client can then decrypt the information using the corresponding "public key" [14]. As long as the private key is not made public, it is computationally hard for a malicious entity to guess the private key [32].

Public key encryption is the underlying technology behind many security solutions used for networks, including secure communications (SSL), electronic transactions (SET), and email encoding (S/MIME). PKE can also be used to construct *digital signatures*. A digital signature is a unique, recognizable proof that data originated from a particular source. The server can send a digital signature with an applet, allowing the applet to be *authenticated*: the recipient can be assured that the sender is really who he claims to be (assuming the client knows the server's public key).

Authentication between the client and server is insufficient if the server is unknown and untrusted. How does a client know that a particular company is safe or not? In and of itself, digitally signed code says nothing about the code's safety; bad code can be encrypted as easily as good code.

The solution has been to use a third machine called a *certification authority* (CA.) The CA is a trusted machine that makes a determination about the safety of the code according to some criteria. If the code is found to be safe, the CA digitally signs the code. Any client that receives the applet can then check whether the code has been signed by a CA.

There are several problems with the CA:

1. The criteria used by certification authorities to determine safety may not take into account the quality of the code. For example, the criteria used by the CA for Microsoft's ActiveX components is merely the fiscal reputation of the company which provided the component [22]. The idea is that if an accounting firm has certified that a company is financially responsible, the client will be able to trust the provider.

   The problem is that this is analogous to allowing car manufacturers to build unsafe cars simply because they are fiscally responsible. If the car breaks, you can sue the manufacturer, but that will not bring back the person who died in the accident. Similarly, if an applet destroys data on the client's hard drive, there is nothing the provider can do to recover the lost data. Encryption does not detect malicious code.

2. Certification authorities are under pressure to not burden developers, while at the same time give legitimate reassurances to the customer. This dilemna can lead to an dangerous situation if the CA does not have the customer's interests as a first priority.

   Verisign, the certification authority used by Microsoft, doesn't make it hard to obtain a certificate. All the CA requires is a rating from the accounting firm Dun & Bradstreet. This rating is based on minimum requirements of the financial stability of the requesting company and has no relation to the quality of the mobile code in question. If a software publisher has released a financial statement or paid taxes, it probably has a D&B rating.

3. The *complexity* of public key encryption may burden the user. For example, Internet Explorer has multiple levels of access to the system. The access levels in part depend on whether the applet's provider is on-site, an individual, or a software publisher. It is the user's job to sort out these different sources.

   Additionally, there may be multiple CAs. They may exist locally on the intranet (Local Registration Agencies), or globally on the Internet. Currently, Verisign, GTE, and Microsoft have all registered to become CAs. It may not always be clear which to use. Currently, the CAs only operate nationally. If the program comes from a company outside of the USA, it will not be recognized by an American CA.

4. It is not unreasonable to consider what would happen if a company's *private key was stolen*. Acquisition of the private keys would allow any entity to spoof the company's applets. Furthermore, massively parallel techniques and high speed computers may be beginning to make the factoring problem computationally feasible using brute force methods [11]. Recently RSA-129 was broken in just one year [8].

Pure authentication is the name given to the security policies which use authentication and no other form of security. ActiveX [22] and Inferno [7] are examples of the pure authentication model. These systems will be further discussed in section 6.

2.4.3 Formal Verification

The third verification technique is *formal verification.* In this strategy the downloaded applet is scanned by another program before it is run. This scanner verifies that the applet is not malicious based on a set of provable theorems to which the code and language must adhere. This technique can also be called *self-verifying code* because it does not require outside human opinion.

Although the Java language and virtual machine do not currently have a formal specification, they probably will in the near future. Once they do, formal verification may be researched further.

The basic problem would be simulating on the program all inputs to show that the code executes in a safe way. However, in general it is not possible for an arbitrary program to be verified because this reduces to the halting problem [35].

"Proof carrying code" (PCC) could facilitate formal specification of applets [25]. Over the network, along with the applet, extra data is sent representing the "proof" that the code is safe. This data is analogous to the "witness" in an NP problem: it is used to prove that the data is safe. Unfortunately, it is very difficult to build the proof. No automated method has yet been devised to do so. Building the proof by hand is a laborious process which may be extremely difficult for large programs. Currently, PCC has only been shown to be practical for very small programs, such as packet filters.

Formal verification is a holy grail. It does not incur any burden on the user because the verifier could be run automatically in the background whenever code is downloaded off of the net. No configuration file needs to be maintained. Unlike the public key encryption schemes, no third party (certification authority) is necessary.

2.5 Summary

This chapter has shown how mobile code can be vulnerable to malicious entities. The goal of a malicious applet is to fool or trick the user into giving it access to the client. It relies on human mistakes to gain access.

A security policy must be employed by the client to prevent applets from damaging the machine. The goal of a policy is to perform its job without incurring any burden on the user. Most significantly, there should be no required technical background to use the policy. The simplicity of using the Web must be retained.

The principal problem with the security policies currenty being put in place are that they are not simple for the user to operate and are vulnerable to human error. Digital signatures are granted based on human judgments which may be fallible, and runtime monitors require the installation of complex configuration files. Formal verification is in principle the simplest to use but has not yet been shown to be feasible.

Chapter 3

Java's Basic Security Policy

This chapter describes the *basic security policy* built into Java's language specification.

To achieve mobility, Java is compiled into a distribution format called a *class file*, which contains information about the classes, methods, and code that make up the program. The instructions which make up the code are represented by *bytecodes*, which can be executed on any computer that implements the Java Virtual Machine. Java's compiler translates source code into class files, which can in turn be sent over the network in an applet.

Java's language inherently contains properties that make it secure. Because Java is type safe (see section 2.3), pointers to invalid memory addresses cannot be forged. This means one cannot modify a pointer so that it points to an illegal address.

Unfortunately, the client cannot rely on the compiler to enforce the language's rules. This is because a malicious applet could have been generated by a compiler that purposely broke Java's rules. Conceptually, there are two ways that the client could protect itself from malicious compilers. The first would be to have known compilers digitally sign the applet which they generate in a way that is recognizable to the client. Alternatively, the client can re-verify the applet in its entirety, to check that the code adheres to the rules. Java's adaptation of the latter approach is called the *bytecode verifier*.

Note that a significant security feature of Java is the language's *openness*. The degree to which Java's specifications and internals have been made available to researchers in academia and elsewhere has had a positive impact on the robustness of Java's security. Sun microsystems and Netscape provide the source code to their implementations to researchers who wish to study and improve its security properties [8] [5]. This feedback has resulted in numerous bug-fixes and many suggestions for improvements.

3.1 Bytecode Level Security: The Bytecode Verifier

Java's *bytecode verifier* checks the code for the following problems.[1] The verifier is run on the client immediately after an applet is downloaded.

1. The stack is kept to a fixed size at every point in a Java program. This prevents loops from overflowing or underflowing the stack.

2. The code is verified that it is type-safe. Any assignment and data conversions in the code are checked.

3. Any branches must be made to legal locations (not outside of a method or to the middle of an instruction.)

[1] This list is not exaustive. For a complete description of the bytecode verifier, see [19].

4. Any registers that are used must be "live," i.e. they have had values assigned to them earlier in the program.

5. Parameters sent to opcodes and methods must be of the correct type and number.

These verification steps prevent Java applets from accessing memory outside of the *sandbox*, i.e. the execution environment legally accessible to the applet.

3.2 Dynamic Linking: The Classloader

Java is *dynamically linked*. Dynamic linking allows applications to share code by enabling them to link to shared objects at runtime. This is in contrast to static linking which binds object only at link time. This simplifies development and maintenance because it removes the link stage, and allows programs to be built up from multiple points on the network [10]. The disadvantage to dynamic linking is that it introduces new security problems.

One problem with dynamically loading classes is that they must be prevented from overwriting system classes. For example, the run-time environment must prevent a hostile applet from dynamically overloading the disk API routines such as the `FileInputStream` constructor.

The subsystem of the runtime environment which prevents this is called the *classloader*. The classloader checks all dynamically loaded classes to be sure they are not illegally overwriting important system classes.

3.3 Runtime Monitor: The Security Manager

The classloader and bytecode verifier keep a Java program constrained within the sandbox. They do not address how a Java program performs operations which must be done *outside* of the sandbox, such as disk accesses. A legal Java program that meets the bytecode verifier's requirements could still format the client's hard disk unless prevented from doing so.

In Java's basic security model, the *security manager's* task is to protect the client from such applets. Java's API routines invoke the security manager prior to doing any work on a system resource, such as the disk. The security manager returns whether the request may be performed or not. By default, this decision is based on whether the applet is local (i.e. originated on the user's hard disk), or is remote (i.e. was downloaded off of the network.) Local applets are given access to the entire system. Remote applets are denied access to the disk or the network (with the exception that connections may be made to the server which provided the applet in the first place).

The browser software can customize the security manager and implement a more detailed policy. Ideally, this would give vendors the ability to implement their own remote monitors.

The major problem with the security manager is that the user must be sure that the API code calls the security manager before it invokes any potentially damaging accesses. Thus, Java's runtime monitor is lightweight in the sense that there is no guarantee that the API routines or other code will invoke the security manager at the right time, and no practical way to check that it does.

3.4 Future Security Features

Sun Microsystems is in the process of improving Java's security model [12]. As described in Section 2.4.1, one new feature is a reference monitor that will allow users to customize the type of accesses to system resources they wish to give applets. The access will be granted based on where the applet came from, e.g. a domain name authenticated using a digital signature. Users will have a high degree of control over what the applet can do. For example, the user could specify the ability to open new windows, access particular directories, or create network connections.

Newer security features will incorporate *protection domains* [33]. This will make explicit the distinctions between local systems modules and remote mobile code, as well as protect different applets from each other. Sun's implementation will allow a thread to enter multiple protection domains during the course of its execution. Note that Sun's implementation of protection domains *do not* isolate one domain's resouces from another (i.e. the JVM will still be vulnerable to resouce denial attacks from hostile applets.)

Support for the secure sockets layer (SSL) and other cryptographic protocols (MD5, DES, etc.) will be supported. This will allow safe network connections, which will enable applications such as authenticated password logging and secure remote method invocation.

3.5 Summary

Java's security policies are multi-layered and complex. Newer versions of Java will incorporate even more features. This complexity may result in many points of failure and make maintenance difficult. Fortunately, Java's openness ensures that problems with the policies are found quickly.

In summary, Java's language and bytecode verifier do a very good job at protecting the client's memory, but its security manager is not a good solution to protecting system resources. Because of security manager's deficiencies, users are reluctant to give applets access to the client's system resources.

Chapter 4

Dataflow Verification

The verification technique proposed in this thesis has many of the advantages of formal verification, yet is computationally feasible. Although dataflow verification cannot prove the correctness of an applet, it can show that a program uses the client's resources safely. For example, if the applet to be verified was an ftp program, dataflow verification could not show that the ftp protocol had been implemented correctly, or that the data being transferred had not been corrupted by the applet. However, dataflow verification could show that the file being transfered and the network connection being opened are the ones specified by the user. In other words, the system resources being accessed by the applet can be shown to be safe.

This is accomplished by greatly simplifying the analysis done by the verifier. Instead of building a general verifier which can decide the safety of arbitrary applets, a verifier is built which can only make decisions on a very restricted set of applets. This set contains all programs which access system resources in a well-defined way. If the program attempts to access a system resource in a manner which is not well-defined, the verifier will reject it.

```
public class sample1 {
  public static void main(String[] args) {
    if (args.length == 0) {
      String filename = getfilename();
    else
      String filename = "/tmp/foo";

    FileInputStream fd = openFile(filename);
  }
}
```

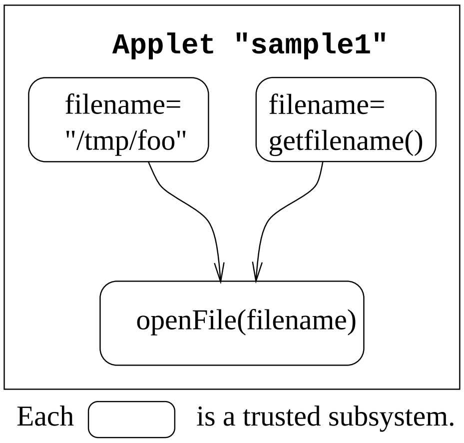


Figure 4.1: Example Flow Diagram and Corresponding Code.

In the context of dataflow verification, a program is well-defined if the verifier can prove that each of the parameters sent to particular API call contains values that make the routine execute safely. For example, if an API call to open a file is called, the verifier must be able to prove that the parameter specifying the filename is safe. This does not mean that the verifier will attempt to determine what value the filename will be during execution. There is no way that the verifier would be able to know whether an arbitrary filename was safe or not. Rather, the verifier checks that the filename parameter *originated* from a body of code which is known to be safe.

This safe body of code that derives the parameter is called a *trusted subsystem*. An example trusted subsystem could be a dialog box that queried the user for a file name. Presumably, the file is safe to open if the user specified its name. In Figure 4.1, this would correspond to line 4, `getfilename()`. Alternatively, a recognizable filename could be embedded in the program. In this instance, the trusted subsystems would be a constant. In Figure 4.1, this corresponds to line 6.

Dataflow analysis is used to check that the parameters originated in a trusted subsystem. The verification must be able to prove that the code outside of the trusted subsystems (i.e. the *untrusted* code) does not modify the data at any point in the path.

Given this framework, a security policy can be devised which regulates how applets can access system resources. The author of the policy (hereafter referred to as the *policy architect*, or PA,) defines a specification which states exactly how resources may be accessed, i.e. what trusted subsystems can derive a given API's parameters. The specification is distributed to software developers, who must write their software so that it adheres to the policy. Compliance is checked at the client when the end user downloads an applet. Together, these steps are called *dataflow verification*.

The following list shows how dataflow verification meets the goals espoused in Section 2.2.1.

1. Dataflow verification has fail-safe defaults: we will show that there is no way for malicious code to circumvent the verifier. By default, the verifier will reject all applets that access system resources.

2. Dataflow verification is extensible: different levels of access to system resources may be defined. Refering to Figure 4.1, a policy suitable for games may allow files containing high scores to be read and written too while denying access to the rest of the file system. A different policy for financial software could allow a range of files within a particular directory to be accessed.

3. Dataflow verification is psychologically acceptable. The only decision that users may need to make is in choosing which policy to use (game policy, financial software policy) to validate their code.

4. Dataflow verification is compatible with current Java systems and virtual machines. It is simply another layer through which an applet must pass before it can be executed.

   However, the developer must construct the applet in a way that conforms to the policy's specification. This specification may be rigid enough so that it is significantly different from the developer's coding habits. Were the specification not so rigid, the dataflow analysis could not be performed.

   Therefore, this strategy may not work for existing applets because any applet which is not written in a recognizable way is assumed to be unsafe. These defaults preclude the browser from running existing applets which access system resources.

   We believe this is an acceptable tradeoff because the developer, unlike the user, has the technical resources to do the work.

4.1 Major Components of Dataflow Verification

This section explains the major concepts which make up the dataflow verification process: critical data, trusted subsystems, dataflow analysis, and the registry.

4.1.1 Critical Data

Critical Data stands for the information passed from the untrusted code to the trusted API routine. The data is considered to be *critical* if it has the potential to make the API routine impact the client in a way that is unsafe. In Figure 4.1, the filename can cause the user damage, such as `/etc/passwd`, therefore it is considered critical.

Not all parameters to API calls are critical. For example, the API routine `fwrite` takes two parameters, `int fp` and `char *buf` (the file descripter and a pointer to data). The policy architect may know that the second parameter could not do anything harmful to the computer no matter what its contents were. In that case, the second parameter is not critical and would be ignored by the verifier.

The verifier's job is to check whether critical data conforms to the policy's specifications in the input applet. This is done using dataflow analysis, described below.

4.1.2 Trusted Subsystems

A *trusted subsystem* is a body of code which is known by both the developer and the policy architect to be safe. In the context of dataflow verification, trusted subsystems are the sources of critical data. In Figure 4.1, the trusted subsystems are `fopen()`, `getfilename()`, and the assignment `filename="/tmp/foo"`. It is the job of the PA to define what the trusted subsystems are.

An obvious candidate for a trusted subsystem is the local code that resides on the user's permanent storage. This definition would encompos *all* of the code which resides on the client, including not only Java's supporting API routines, but also potentially untrusted code that had somehow been stored on the user's local disk. A more restrictive definition would limit trusted subsystems just to the standard set of API routines.

There are advantages to limiting trusted subsystems to the standard system routines. Standard calls are part of the mobile code system, and we assume there are no bugs in the implementation. However, the disadvantage to limiting trusted subsystems is that doing so may make the developer's job very difficult. This is because the API routine's functionality is typically very primitive: they express actions on the system resources in a narrow and precise way. The developer, on the other hand, will probably desire a richer and more flexible body of routines from which to derive critical data. Therefore, the policy architect may desire to write his own trusted subsystens.

As an example, suppose the PA creates a rule stating that applets may only open files if the filename had been derived from a dialog box which queried the user. In Figure 4.1, this dialog box might correspond to the `getfilename()` routine. In addition, the PA might want the `getfilename()` routine to filter out filenames that contain illegal directories. In the Java Development Kit there is no single API routine that performs those functions. The PA is forced to obtain the code to do that outside of the standard API.

The difficulty with defining new trusted subsystems is that the PA is essentially inventing a *new* standard API routine and requesting that it be adapted by clients. This raises compatibility problems: one PA's trusted subsystems may be different than another's. Additionally, the client must somehow be convinced that the PA's new trusted subsystems are safe.

The PA may also define a trusted subsystem to be a known constant value that has been hardcoded into the applet. As Figure 4.1 shows, this could be used to express in the policy that only one particular filename may be opened by the applet. Section 4.2.1 will show how constants are important for other reasons and elaborate on the idea.

An important point to make about the design of trusted subsystems is that they cannot contain any callbacks. Callbacks are common in AWT routines. For example, the configuration of a button widget typically specifies a procedure that should be called when a button is pressed. The problem with callbacks is that control could jump from a trusted subsystem into untrusted code. Conceivably, dataflow verification could check for callbacks within trusted subsystems. However, trusted subsystems are by definition not supposed to have any unintended consequences. The verifier should not have to check them for such problems.

4.1.3 Dataflow Analysis

In dataflow verification, *dataflow analysis* is a process used for two purposes:

1. To identify where in the applet that critical data is transferred between trusted subsystems. This flow is called a *sensitive path*.
2. To verify that critical data has not been tampered by untrusted code in the sensitive path.

Example 4.2 graphically shows each of the components of dataflow verification. The dashed line represents the sensitive path through the code in which the critical data `fname` is *live* (e.g. the variable has had a value assigned to it which will be used later). The trusted subsystems are represented by the rectangles.

A principle problem involved with the verification's design is choosing how sophisticated the dataflow analysis will be. This directly impacts how the developer is allowed code up invocations to critical API calls. Essentially, the verifier imposes *constraints* on how the developer can write his code.

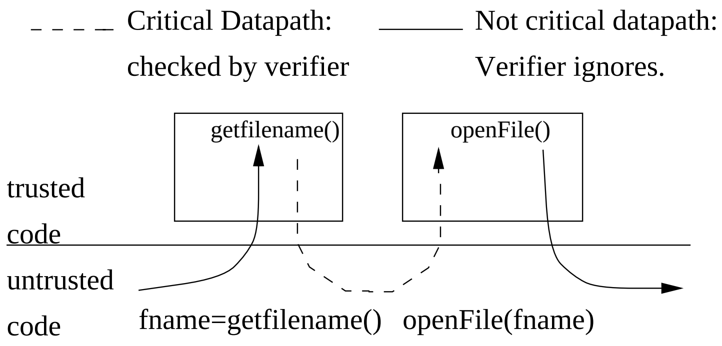


Figure 4.2: Diagram of Dataflow Analysis, Trusted Subsystems, and Sensitive Paths

There are advantages and disadvantages to making the dataflow analysis more sophisticated. If the dataflow analysis was sophisticated, the developer would have more choices in how he decided to code the flow of critical data between trusted subsystems. This flexibility would make his code more modular; e.g. bug-free and easier to maintain. It may also ease the transition between converting code which does not meet the requirements of the verifier to code which does.

The disadvantage to complex analysis, however, is that those algorithms are difficult to implement, and can be buggy. Complex security systems have more points of failure; recall that this is already a frequent complaint raised against Java [8]. In addition, complex dataflow analysis may incur performance penalities; for example, it is time consuming to compute dataflow structures such as gen and kill sets across method boundaries [2]. Coupling this analysis with just-in-time compilation could result in a visible delay to the user.

4.1.4 The Registry

The job of the PA is to write a specification that explicitly states the sensitive parameters to trusted subsystems, and the valid trusted subsystems from which those parameters may be derived. The specification is expressed in a special configuration file called a *registry.*

The complete textual representation of a registry corresponding to Figure 4.1 is below. The lines in the file which begin with the word `illegal` represent instances of flowpaths of data between trusted subsystems that cannot exist anywhere in the applet. In Figure 4.1, line 4 expresses that it is illegal to open a file using any API routine except by the `FileInputStream` or `File` constructors.

```
// Legal functions
legal openFile(getfilename());
legal openFile("/tmp/foo");

// Disallowed functions
illegal java/io/File/<init>(*);
illegal java/io/FileInputStream(*);
```

Lines in the registry that begin with the word `legal` represent instances of critical datapaths which may exist in the applet. In Figure 4.1, line 1 states that the `openFile()` API may be called *if* the parameter sent to it was derived from `getfilename()`. Line 2 states that `openFile` may be called if the file to be opened is `/tmp/foo`.

In the registry, a trusted subsystem is represented by giving its name. In place of parameters, the trusted subsystems which derive the parameter are given. Therefore the line:

```
legal openFile(getfilename());
```

Means that the parameter for `openFile()` must have been derived from the getFileName() trusted subsystem. There is an implicit sensitive path between the getFileName() routine and the `openFile()` routine.

The parameters are critical data, unless an * is used. If the PA were to liberalize his system to allow *any* filename to be opened using `getfilename`, he would change the registry to read:

```
legal openFile(*);
```

The problem now is that a developer might choose to open a file using a different API routine than `openFile()`. To prevent this, the PA must explicitly use the `illegal` keyword to express that this is not allowed. The PA must be careful that all possible means of accessing the client's system resources are blocked except via known methods.

```
legal openFile(*);
illegal java/io/FileInputStream(*);
illegal java/io/File/<init>(*);
```

The registry is not to be confused with the specification files used in runtime monitors, such as Sun's specification file shown in section 2.4.1. They are fundamentally different. Firstly, the registry's contents are transparent to the user; the low level details that it contains are only of interest to developers and PAs. In contrast, a runtime monitor is written by the user, who must thereby assume responsibility for any errors it contains.

Potentially, the user could be asked to choose from a range of registries to use. This is because different registries represent different levels of access to system resources, and some registries may be more suitable for certain applications than others. However, giving the user too much control over this process may burden the user in the same way that runtime monitor's configuration files do. A balance needs to be made between the goals of extensibility (allowing the user to use multiple policies) and complexity (the user should have to take on as little responsibility as possible). This problem will be discussed further in section 7.1.

4.2 Dataflow Verification's Limitations

This section will describe dataflow verification's limitations. By way of illustration, this section will consider the registry required to implement the *file transfer protocol* (ftp) [30].

In the ftp problem, the verifier must ensure that the applet only sends *allowed* data out onto the network, and no other data. Only the file indicated by the user may be made public. The simplified registry for ftp might do the following:

1. Create a socket connection to a trusted host.
2. Open a trusted file on the client.
3. Read data from the file.
4. Send the data out onto the destination socket.

Such a registry is shown below.[1] The origin of the `datagram` parameter is not shown. It will be described in the next section.

```
// simplified registry for ftp.
legal socket=openSocket(gethostname());
legal fp=fileOpen(getfilename());
legal fileWrite(socket,datagram);
```

The next two subsections will show the issues that arise in implementing this registry.

4.2.1 Manipulation of Critical Data

It is very difficult for a verifier to distinguish valid operations on critical data from invalid ones. This is a problem because critical data must be manipulated in order to implement ftp. The ftp protocol encapsulates the data sent to the recipient in a formatted structure (a packet) so that it will be understood by the recipient.

For example, to send a buffer, a programming construct must be built which concatenates the `SEND` command with a buffer, as in the source code below:

```
String data=readFile(fp);
String datagram="SEND "+data;
fileWrite(fp,datagram);
```

Suppose the lines below was inserted into the source. In order to prove that ftp was implemented correctly, the verifier would have to be capable of determining that the following two expressions are invalid.

```
String datagram=data+"SEND ";
String datagram="WRONG "+data;
```

How can the verifier tell the difference between a valid operation on the critical data from an invalid operation? Below are two suggestions which might help resolve this dilemma:

1. The PA can allow *known constants* to be used in expressions. These constants could be hardcoded in the registry. For example, `SEND` is a constant string that is known to be valid.

[1] The `illegal` constructs are omitted for simplicity.

```
// Simplified registry for FTP.
legal socket=openSocket(gethostname());
legal fp=openFile(getfilename());
legal data=readFile(fp);
legal datagram= "SEND "+datagram;
legal writeSocket(socket,datagram);
```

2. The policy architect could *encapsulate* the code that builds up the ftp packets in a trusted subsystem. This trusted subsystem could be provided by the PA to developers.

   For example, encapsulation would be useful in implementing more complex fields constructed in protocols which cannot use constants, such as sequence numbers. A trusted system could be provided that generated a correct sequence number and inserted it into critical data.

Both encapsulation and known constants suffer from the same drawbacks of not being general enough. While configuring the registry using a known constant might allow the developer to write an ftp applet, it would not help him implement a different protocol: what constants would have to be hardcoded if the developer decided to write MIME? It is difficult for the policy architect to anticipate what constants the developer will need. Similarly, what trusted subsystems would the policy architect have to encapsulate?

Unless one of the above steps are taken, dataflow verification cannot show that ftp has been implemented correctly. However, dataflow verification can prove that the client's system resouces are being used safely: the user can verify that the applet only opens a safe file (i.e. no private data has been leaked) and a safe network connection (i.e. the data has not been sent to an untrusted host).

4.2.2 Covert channels

A covert channel is a *valid* connection from the client onto the network that *covertly* sends information onto the connection that it is not supposed to send. For example, a covert channel might send a credit card number by secretly embedding it in a jpeg image. The user would think only a picture was being sent. Or, an applet could spawn off a thread which loaded the CPU in a way that the *rate* at which data was sent was changed. If the rate were cleverly controlled it could become a Morse code which could encode private information.

Essentially, there are an unlimited number of ways that covert channels can be implemented [18], and this property makes it extremely difficult for a verifier to detect them in arbitrary code. Many sophisticated techniques have been devised to prevent covert channels [28]. But despite these efforts, no general solution to the problem has been found [23].

Any applet which has both a network connection and access to private data risks a covert channel. For example, if the ftp applet had opened a file that was meant to be private, and also sent a public file using ftp, there could have been a covert channel.

Although dataflow verification cannot detect covert channels, the policy architect could take one of the following steps to prevent them:

1. The PA can ensure that the applet will only open files that have been made *public*. This could be done using known constants, by ensuring only a particular directory, such as `/pub`, was used on the client.

2. Known constants could also be used by the PA to ensure that only connections to particular hosts were made. It may be that the PA knows what connections the client may contact legally, and these could be hardcoded into the policy.

3. The PA can *prevent* a network connection from being made in applets that have access to private data. Such applets would be called *inside* applets [29] because the registry could give them unrestricted read access to the client but no access to the network.

   Alternatively, dataflow verification could be used to disallow any access to the private disk if the applet is detected to make a network connection. This would be an *outside* applet.

4.3 Summary

In this chapter, we have introduced a new security policy, dataflow verification, which meets the requirments of Section 2.2.1. The policy works by forcing the developer to comply to a specification (the registry) built by a policy architect. Applets are verified automatically by the client. The details of the verification process were explained, showing that it is essentially a dataflow analysis problem.

We showed that dataflow verification is different than formal verification: it cannot prove a program's correctness. We showed that dataflow verification's major advantage is that it can prove that the system resources available to the applet are safe.

Because the verification is completely automated, the user does not need to have any knowledge of how it works. However, the user may use different registries to give varying levels of access to different applets if he desires.

Chapter 5

Prototype

5.1 Organization

We have constructed prototype software to test the validity of dataflow verification. The prototype takes as input a registry and the applet to be checked. The software outputs true if the applet meets the registry's specifications, and false otherwise.

The software was written entirely in Java using JDK 1.1 over a three month period. The software is standalone- i.e. it is not integrated into a browser. Its architecture most closely resembles "ahead-of-time online verification" in Figure 2.1. Significant portions of the verifier were derived from pre-existing projects such as Sumatra [31] and Bali [16].

The major components of the prototype are described below.

registry compiler (391 lines) This is a small compiler that converts the textual representation of a security policy (the registry) into a table which can be read by the dataflow analysis subsection of the verifier. This translation is intended to be performed ahead-of-time.

Conceptually, *just-in-time security policies*, in which a PA dynamically changes the client's security policy, could be implemented. This would require the registry compiler to be moved into the verifier.

classfile decomposition (6291 lines) The classfile decomposition software was built by the Sumatra group [31]. The decomposer takes as input a classfile and makes visible to the programmer all the information that the classfile contains. The information used by the verifier includes the class's methods, code, the number of local variables used in a method, and exceptions [19].

code analysis (1431 lines) Large parts of the code analysis subsection of the verifier were taken from a Java bytecode optimizer, Bali [16]. This software was originally written in C and was a converted to Java to interoperate with the prototype. Bali takes the raw data from the class decomposer (code and variables) and identifies structures within it that can be later used to perform dataflow analysis. Two of the most important computations executed during the phase are:

Determining the Control Flow Graph (CFG) The control flow graph depicts all of the posible paths by which execution could flow within a method [2]. When performing dataflow verification, the graph is used to trace backwards in the code from the point that a variable is used to each point where it may have been defined.

Performing Stack Simulation The Java bytecode instruction set is stack-based. It also has the property that at any point in the code the contents of the stack can be statically determined.

In the prototype, the *stack simulation* algorithm determines the contents of the stack at every instruction in the code [16]. This information is used to perform dataflow analysis.

Note that code which was built by a malicious compiler would be caught by the byte-code verifier explained in Section 3.1. In a production verifier it may be more efficient to combine the dataflow and bytecode verification info a single system.

dataflow analysis (485 lines) This section checks the input applet for the validity of critical dataflow paths as defined by the registry. In the prototype, the dataflow analysis techniques were chosen to be very simple because the overriding goal was only to demonstrate the concept. In a production verifier, more complex analysis could be used. The dataflow analysis algorithm will be explained in section 5.2.

When the dataflow analysis software detects an unsafe construct in the applet it outputs a textual description of the problem. The description shows the class, method, bytecode offset of the offending code, and the rule in the registry that has been violated.

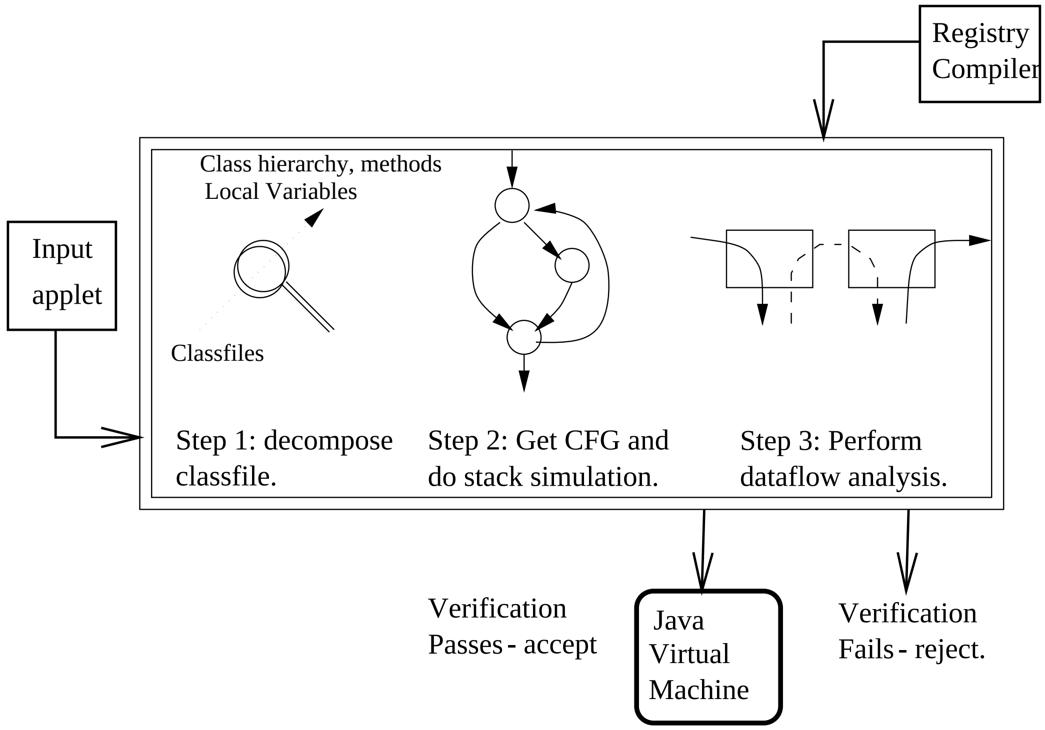


Figure 5.1: Prototype Organization

The interaction between the verifier's components is graphically depicted in Figure 5.1. Note that double framed rectangle in the picture represents a "black box"; everything inside of it is invisible to the user.

5.2 Dataflow Verification Algorithm

This section explains how dataflow verification was implemented in the prototype. The following registry will be used as an example.

```
legal openFile(getfilename());
legal openFile("/tmp/foo");
```

The dataflow analysis must check that the parameters to every `openFile()` invokation in the applet are valid. Suppose the code in Figure 4.1 was to be verified. The Java compiler would translate class `sample1` into the following bytecode: [1]

```
 0 aload_0
 1 arraylength
 2 ifne 12
 5 invokestatic #7 <Method test.getfilename()Ljava/lang/String;>
 8 astore_1
 9 goto 15
12 ldc #2 <String "/tmp/foo">
14 astore_1
15 aload_1
16 invokestatic #6 <Method test.openFile(Ljava/lang/String;)V>
```

First, the verifier will scan the applet for calls to `openFile()`. It will find one at offset 16 in the bytecode sequence.

The algorithm will then use dataflow analysis to trace backwords from that point to find where the parameters to `openFile` were defined:

1. To locate what the critical data is, the algorithm looks at the stack at the point where `openFile()` is invoked. In the example, the stack contains a reference to an object (in the source, this corresponds to the `fname` string).

2. The algorithm then uses the control flow graph to look at preceding instructions to find where the critical data was loaded onto the stack. There will frequently be more then one predecessor to a given instruction, but at offset 16 the only predecessor is the instruction at offset 15: `aload_1`. This instruction tells the verifier that the critical data is stored in reference variable 1.

3. The control flow graph shows that offset 15 has two predecessors (offset 9 and offset 14). Both of these execution paths (i.e. sensitive paths) must be traced.

   The algorithm will trace backwards in the code along each sensitive path. It will stop tracing a sensitive path when it detects one of the following events:

   (a) If a store is made to critical data, the verifier will check where the instruction got the data it is storing. For example, at offset 14, the verifier would see that the `ldc` instruction loaded `"/tmp/foo"`. In the example, this is where the critical data's value was generated. Because the registry shows that this is a valid value, the tracing on that sensitive path will halt. The verifier will go on to trace the rest of the sensitive paths.

   (b) If an operation (such as an arithmatic or string operation) is made on the data, the prototype will abort. This is because the prototype does not have a way of determining whether the manipulation of critical data is valid or not (see Section 4.2.1).[2]

[1] The first column corresponds to the byte offset from the beginning of the method.
[2] The prototype did not implement known constants.

(c) If an invokation is made to a static method, and that method is a function which *returned* critical data, the prototype will trace the sensitive path into the method. If the invokation is to a virtual function, the prototype will reject the applet (see Section 5.3.2).

(d) If the beginning of the method is reached, the prototype will reject the applet. This is because the code analysis phase does not determine all the places where the method could have been called from in the applet. This is a limitation of the prototype that could be easily removed.

5.3 Prototype Limitations

This section describes the prototype's limitations. These limitations are not inherent with dataflow verification; they exist because they simplified the implementation. A production verifier could remove these limitations.

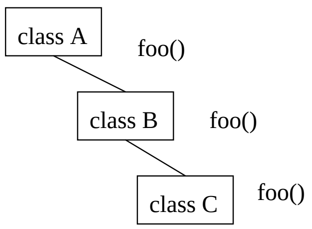
5.3.1 No Dynamic Linking.

In the prototype, every class that is accessible by the code must also be accessible by the verifier. Clearly, code within the sensitive path must be visible to the verifier. But classes that are not within the sensitive path must also be accessible. This is because malicious code could dynamically reload a class that lies in the sensitive path. If the class name to be loaded was built up from an expression, the verifier would not be able to determine whether it contained code in the sensitive path or not. To avoid this problem, all of the classes in the applet must be *statically* determinable.

An alternative implementation could integrate the verifier with the classloader. This would check code that was dynamically loaded. The drawback to this idea is that it would make the verification scheme incompatible with older browsers.

5.3.2 Don't Allow Virtual Functions

If a parameter sent into a critical API has been derived from the return value of a function, then the function itself must be traced. In Figure 5.2 below, the dataflow analysis must check method `z.foo()`.




```
static void dummy(A var1) {
   fname=var1.foo();
   openFile(fname);
}
```

Figure 5.2: Example of Virtual Functions

Virtual function calls are problematic because they can be overloaded. In Figure 5.2, `foo()` could refer to class A, B, or C, depending on the value of `var1`. The verifier must analyze all the functions, because it is impossible using static analysis to determine which one in the hierarchy is being called.

One simplification is to require function calls to be final. But a method which has been declared final may still itself be overloading a method of which it is a child. To get around these complications, the prototype requires that procedures which carry critical data be static. This constraint could be averted if the verifier performed dataflow analysis on each of the virtual functions in the class hierarchy.

5.3.3 Restrict Critical Data to Local Variables

A local variable in Java is a variable whose scope only includes the method in which it is used. Because of their limited scope it is not difficult to search for all store operations to that variable.

Java does not have global variables in the same sense that languages such as C do. In Java, a variable that is not local must be defined in a class. Such variables are difficult to analyze because there may be many references throughout the code to a class's instance, and each reference may potentially perform an illegal store operation onto the class's variable. Thus, in contrast to local variables, variables defined in classes require a very complex analysis. For this reason it was decided that in the prototype, critical data may only be stored in local variables.

5.3.4 Don't Allow Asynchronous Exceptions

The following code throws an exception.[3] In the exception handler, critical data is changed within the throw clause to an invalid value.

```
String fname;
try {
        int z[];
        z=new int[10];
        fname=trusted.getfilename();
        z[11]=1;              // ** throw exception **
        FileOutputStream fp=trusted.openFile(fname);
}
catch (Exception e) {
        fname="Illegalfile";  // Change critical data.
}
```

In the prototype, a constraint was imposed stipulating that critical datapaths must contain throw clauses. Note that the clause must catch *all* exceptions. The only code in the throw clause is a call to a trusted exception handler. Thus, the example above must be modified to become:

```
String fname;
try {
        int z[];
        z=new int[10];
        fname=trusted.getfilename();
        z[11]=1;                 // ** throw exception **
        FileOutputStream fp=trusted.openFile(fname);
}
catch (Exception e) {            // Catch all exceptions.
        trusted.TrustedCatch(e); // Call trusted handler.
}
```

This solution is problematic because the `TrustedCatch()` routine must be standardized somehow. It would be difficult for the PA to devise a procedure that would be flexible enough to meet all of the desires of the developer. The routine would most likely simply terminate the program on an exception.

[3] For demonstration purposes, the `getfilename` and `openFile` methods are assumed to be part of a class named "trusted".

A more flexible analysis not implemented in the prototype is to check the validity of all of the exception handlers within the critical datapath. The analysis has to check whether the handler makes any stores to critical data. The analysis is complicated because all of the exception handlers in the call stack in the sensitive path must be traced.

For example, if an "end of file" exception was thrown in function A, and function A did not have a throw clause to catch that exception, the run time system would look for a handler in the routine that called function A. Suppose that was function B. If the sensitive path extended into function B, the handler in B would have to be analyzed by the verifier. Sorting out these dependencies is complex.

5.4 Summary

This chapter described a prototype verifier which implemented dataflow verification. The verifier was built only to show that the idea works, and therefore the sophistication of the dataflow analysis is limited. These limitations are not inherent to dataflow verification.

Chapter 6

Related Work

This chapter describes how three other popular mobile code systems protect the client from malicious applets, and briefly looks at dataflow verification's applicability to other languages.

6.1 ActiveX

ActiveX is binary code signed using public key encryption and certification authorities. Certification is done by companies independent of Microsoft, such as Verisign and GTE. A company is allowed to become a CA if it has been deemed acceptable by an umbrella organization.

The certification authorities digitally sign ActiveX components in a way that corresponds to the degree of safety that the code is *presumed* to have. That decision is based on the type of organization that requested the signature. The simplest and cheapest certificate is available to individual vendors. More expensive (i.e. safer) certificates require agencies such as Dun and Bradstreet to scrutinize the vendor.

6.2 Inferno

Inferno is different from ActiveX in that it compiles to an architecture neutral bytecode (DIS), just like Java does. However, unlike Java, Inferno's designers desired the language to have full access to the client hardware and not be constrained within a sandbox. Thus, security is not inherently built into DIS. Security is provided by a public key encryption strategy that is similar to ActiveX. Inferno is different from Java and ActiveX in that its clients are intended to be embedded systems or network interconnection devices, as opposed to mainstream Internet users. Therefore, security controls may not have to be as strict because the client will be presumably more technically adept and, therefore, be able to take on security responsibilities.

6.3 Safe-Tcl

Tcl is an interpreted scripting language that is popular for GUI development. In a recent paper [29] the Safe-Tcl model was introduced. In Safe-Tcl two types of interpretors are used. Safe interpretors have total access to system resources- i.e. they can use any of the API routines to open files, etc. Unsafe (i.e. mobile) code use a *separate* interpretor which does not have access to system resources. If an unsafe routine desires to access a system resource it is required to invoke an alias to a safe interpretor. The analogy is that the safe interpretor is like "kernel" mode, and the restricted interpretor is like "user" mode. The invocation between the two is akin to a system call.

The major advantage to Safe-Tcl is its simplicity in implementation compared with runtime checks bolted onto the Java Virtual Machine. The interpretors themselves do not need to be modified. However, this simplification is passed on to the developer, not the user. The user still needs to set up configuration files and thus Safe-Tcl suffers from the problems described in Section 2.4.1.

6.4 Applicability to Other Languages

There is nothing unique about that Java language that prevents dataflow verification from being implemented in other languages. The principal requirement is that the language be type-safe, so that critical data can be traced. Modula-3, Oberon, or Inferno's Limbo could all be amenable to dataflow verification. [1]

Weakly typed languages such as C++ or assembly cannot be checked with dataflow verification. Note that because ActiveX components are in binary code, they cannot be verified.

6.5 Summary

The most widely adapted security strategy used to protect mobile code is digital signatures. This is probably because digital signatures can be used with any language, even machine code. The ability to use machine code is important for systems which desire complete access to the client's hardware. Mobile code that has been implemented using higher level languages which are type safe can employ stronger security policies, such as dataflow verification.

[1] This assumes it was known that a trusted compiler had generated the binary, see Chapter 3.

Chapter 7

Conclusion

7.1 Discussion and Future Work

As alluded to in Chapter 4, the user could conceivably be asked to choose from a range of registries to use on the browser. One problem is that this might enable a malicious applet to trick the user into choosing a policy that gave it more access to the machine than it should have. On the other hand, the ability to choose from a range of policies was one of the principal goals of a verification strategy (e.g. extensibility, in Section 2.2.1). Furthermore, covert channels may require multiple registries: one for inside applets and another for outside applets.

Potentially, these conflicting needs could be reconciled by implementing a verifier that *automatically* choose a policy. In this scenario, a verifier could scan the applet to determine whether it was an inside or outside applet, and choose the correct policy accordingly. This same idea could be applied to game applets or financial applets. If the type of applet could not be determined by scanning the applet, a restrictive default policy could be used. Experimenting with this idea is an area of future work.

Trusted subsystems are problematic because they might have to be coded and provided by the PA. The argument against this was not unlike that against code signed by certification authorities. The PA may unwittingly insert a bug in a trusted subsystem that could be used by a malicious entity. Also, Section 4.2.1 pointed out that it is difficult for a policy architect to anticipate what trusted subsystems clients and developers would need.

One partial solution to this dilemma may be to allow *local policy architects*, who are the system administrators of a protected intranet, to define trusted subsystems. Presumably, those individuals are knowledgeable of the needs within their administrative boundary.

Finally, an additional area of future work may include experimenting with dataflow analysis techniques to detect hostile applets and covert channels.

7.2 Close

History has shown that the weakest link in the security chain is the human component. Human error caused not only most of Java's security holes [8], but are to blame for other security systems breaches, such as those in ATM machines [3]. The luring, trojan horses, and spoofing attacks all rely on deceiving the user. All a malicious applet has to do to circumvent a browser guarded by digital signatures is fool the certification authority. If the client is protected by a runtime monitor, human vulnerabilities could still lead to disaster, were the user to make a mistake in the configuration of the monitor.

In this thesis we have presented a system which minimizes the amount of responsibility left to humans, by automating as much of the security apparatus as possible. For decisions which must be made by human beings, we have chosen to give that responsibility to someone other than the user. The user is least likely to have the technical expertise required to make good decisions.

Someday, formal verification may become practical. Until that day comes, we believe automated security policies such as dataflow verification are the best way to protect the client from hostile mobile code.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. "Mach: A New Kernel Foundation for UNIX Development". *Proceedings of the Summer 1986 USENIX Conference.* July, 1986. pages. 93-112.

[2] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] R. Anderson. "Why Cryptosystems Fail." *1st Conference on Computer and Communication Security.* November, 1993.

[4] D. Bass. "The Inpact of the Network Computer". *http://www2.netdoor.com/ davebass/NCSHIM.HTM.* April 1997.

[5] B. Bershad, G. S. Emin, S. McDirmid. Kimera Architecture. `http://kimera.cs.washington.edu/overview.html`

[6] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, S. Eggers. "Extensibility, Safety, and Performance in the SPIN Operating System." *Proceedings of the 15th Symposium on Operating Systems Principles."*

[7] P. David. "Inferno Security." *Proceedings of IEEE COMPCON 1997* pages 97-102. February 1997.

[8] D. Dean, E. Felten, D. Wallach. "Java Security: From HotJava to Netscape and Beyond." *IEEE Symposium on Security and Privacy.* May 1996.

[9] E. W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. "Web Spoofing: An Internet Con Game." Technical Report 540-96. Department of Computer Science, Princeton University. December 1996.

[10] D. Flanagan. "Java in a Nutshell." O'Reilly & Associates, Inc.

[11] G. Fox, W. Furmanski. "Computing on the Web New Approaches to Parallel Processing Petaop and Exaop Performance in the Year 2007." Technical Report SCCS-784, Syracuse University, NPAC. January 1997.

[12] L. Gong. "New Security Architectural Directions for Java." *Proceedings of IEEE COMPCON 1997* pages 97-102. February 1997.

[13] J. Hartman, U. Manber, L. Peterson, T. Proebsting. "Liquid Software: A New Paradigm for Networked Systems." Technical Report 96-11. Dept. of Computer Science, The University of Arizona.

[14] C. Kaufman, R. Perlman, M. Speciner. "Network Security: Private Communication in a PUBLIC World." Prentice Hall, 1995.

[15] M. Ladue. A Collection of Increasingly Hostile Applets. URL: http://www.math.gatech.edu/ mladue/HostileApplets.html

[16] H. D. Lambright. "Java Bytecode Optimizations." *Proceedings of IEEE COMPCON 1997*, pages 206-210, February 1997.

[17] B. W. Lampson. "Protection." *Proceedings of the 5th Priceton Symposium on Information Sciences and Systems.* Princeton University, March 1971. page 437-443. Reprinted in *Operating Systems Review*, 8(1):18-24, January 1974.

[18] B. W. Lampson. "A note on the confinement problem." *Communications of the ACM 16,* 10 pages 613-615. October 1973.

[19] T. Lindholdm, F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[20] P. Madany. "JavaOS(tm): A Standalone Java Environment". http://www.javasoft.com/docs/white/index.html

[21] G. McGraw, Edward W. Felten. "Java Security: Hostile Applets, Holes and Antidotes." John Wiley and Sons, New York, 1996.

[22] Microsoft Corporation. *Proposal for Authenticating Code over the Internet* April 1996. http://www.microsoft.com/intdev/security/authcode

[23] I. Moskowitz, A.R. Miller. "Covert Channels - Here to Stay?". *Proceedings of the IEEE Symposium on Research in Security and Privacy.* Oakland, CA. 1994.

[24] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria (The Orange Book).* 1985.

[25] G. Necula. "Proof-Carrying Code." *Proceedings of the 24th Symposium on Principles and Programming Languages.* January, 1997.

[26] G. Nelson. *Systems Programming in Modula-3.* Prentice Hall. 1991.

[27] Netscape Corporation. *Secure Sockets Layer.* 1997. http://www.netscape.com/info/security-doc.html

[28] N. Ogurtsov, H. Orman, R. Schroeppel, S. O'Malley. "Covert Channel Elimination Protocols." TR96-14. The University of Arizona.

[29] J. Ousterhout, J. Levy, B. Welch. "The Safe-Tcl Security Model." May 1997. http://www.sunlabs.com/techrep/1997/abstract-60.html

[30] J. Postel, J. Reynolds. "File Transfer Protocol". *Request for Comments 959.* October, 1985.

[31] Proebsting, Todd. "The Sumatra Project". http://www.cs.arizona.edu/sumatra

[32] R. Rivest. "Cryptography." *Handbook of Theoretical Computer Science.* Chapter 13. Elsevier Science Publishers. 1990.

[33] J. Saltzer. "Protection and the Control of Information Sharing in Multics". *Communications of the ACM.* 17(7):388-402. July 1974.

[34] M. Seltzer, Y. Endo, C. Small, K. Smith. “Dealing With Disaster: Surviving Misbehaved Kernel Extensions”. *Proceedings of the 1996 Symposium on Operating System Design and Implementation (OSDI II)*

[35] M. Sipser. “Introduction to the Theory of Computation”. PWS Publishing Company. 1997.

[36] D. L. Tennenhouse, J. M. Smith, W. Sincoskie, D. J. Wetherall, G. J. Minde. “A Survey of Active Network Research” *IEEE Communications Magazine.* Vol. 35, No. 1, pp.80-86. January 1997.

[37] R. Wahbe, S. Lucco, T.E. Anderson, S. Graham. "Efficient Software-Based Fault Isolation." *Proceedings of the Fourteenth Symposium on Operating Systems Principles.* 1993.

[38] D. S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. "Extensible Security Architectures for Java." Technical Report 546-97, Department of Computer Science, Princeton University. April 1997.